Bio-inspired Implicit Communication and Control for Quadcopter Drone Fault-Tolerant Formation Flight

BY

LEONARDO CERRUTI
B.S, Politecnico di Torino, Turin, Italy, 2022

THESIS

Submitted as partial fulfillment of the requirements
for the degree of double Master of Science in Electrical and Computer Engineering
in the Graduate College of the
Politecnico di Torino and
University of Illinois at Chicago, 2024

Chicago, Illinois

Defense Committee:

Igor Paprotny, Chair and Advisor
Zhao Zhang
Alessandro Rizzo, Politecnico di Torino

(Politecnico di Torino version)

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to all those who provided me with the possibility to complete this thesis. A special gratitude I give to my family, who sustained me in this project, and to my roommates for tolerating me.

Furthermore, I would also like to acknowledge with appreciation my advisor Igor Paprotny, who provided materials to complete the task and the initial idea. A special thanks goes to my other advisor, Alessandro Rizzo, who helped me by distance with suggestions throughout the project and to Zhao Zhang for his participation.

I also really appreciated the guidance of John Sabino, PhD student in my same lab for his suggestions and help overall.

I have to appreciate the guidance given by the Top-UIC project representatives, in particular Jenna Stephens.

L C

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

FIGURE **PAGE**

LIST OF FIGURES (continued)

FIGURE **PAGE**

LIST OF ABBREVIATIONS

UIC	University of Illinois at Chicago
DOF	Degrees Of Freedom

SUMMARY

The first main objective of this Thesis research project, that is the final step for completing the double master's degree program between Politecnico di Torino and UIC, is to develop a drone following algorithm based on the study of drones' interaction signals. This algorithm must be able to run on drones with low computational power as the ones used to develop it.

The idea behind this algorithm stands in the geese' formation flight, where such birds are able to fly in an optimal wing vortex point to save energy. This results in their typical V-shaped swarm formations [16].

Similarly, the used drones must be able to detect the wind produced by other's rotors to engage a following procedure.

This research work is divided into 3 main parts as 3 main actions were completed.

At first, the drone interaction model had to be built by performing some real-world tests and by analyzing the results. After that, a solution was found to the problem using a multiple detection approach: the drone that follows would approach the leader multiple times and register the necessary information to estimate its speed. This algorithm also takes into account all cases of wrong speed detection by periodically forcing a new leader search.

Later, it was noticed that the drone-following algorithm could be used to actually transmit data from the leader to the follower; in fact, at any point, the leader could perform a specific speed pattern that will be interpreted by the follower as a command. Here there is the second part of this thesis title: this functionality can be used in case a direct communication channel

between drones fails. A working leader, in that case, would still be able to give commands to the faulty elements, such as "go back home" or "land".

Finally, it was noticed that the available drones weren't able to fly closely as their ultrasonic sensors' beam angles were big enough to cause altitude control disturbances. As a solution to this problem, a new altitude estimation algorithm was designed exploiting available elements in the system, such as the camera and rotors' power output. This algorithm consisted of completely redesigning and optimizing the Image processing system for a better data rate and also redesigning the state estimator block.

The obtained result exploits the image signal as validation and the thrust signal as instant values because of its higher data rate. However, it was demonstrated that using a different platform with optical sensors might be a much easier solution (less computationally expensive).

CHAPTER 1

INTRODUCTION

1.1 Unnamed Aerial Vehicles (UAV) history overview

Since ancient times, humans have been admiring the birds' capability to fly, leading to a growing desire to imitate them. Following the history timeline described by Konstantinos Dalamagkidis [17] I'll briefly list the main human achievements.

The first attempts date back to 450-400 BC, when the Greek philosopher Archytas the Tarantine (from the city of Taranto in southern Italy, part of the Magna Graecia, a region of Greece at that time) was able to build the so-called pigeon. This wooden bird was able to fly using pressurized air (Figure 1.1).

Figure 1.1: Reconstruction of the pigeon, from Marco Mellace [1]

Around that same period, in China, there were more attempts: together with more wooden birds, the first documented vertical flight device was a stick with feathers on top. Spinning this stick by hand made it possible to get some kind of lift for a short flight.

Much later, in 1483, Leonardo Da Vinci designed what could be seen as the first helicopter, the aerial screw (Figure 1.2). He also designed a mechanical bird able to flap its wings in the exact way of the real animals. Unfortunately, he could never find a way to give them enough energy to fly as modern engines didn't yet exist.

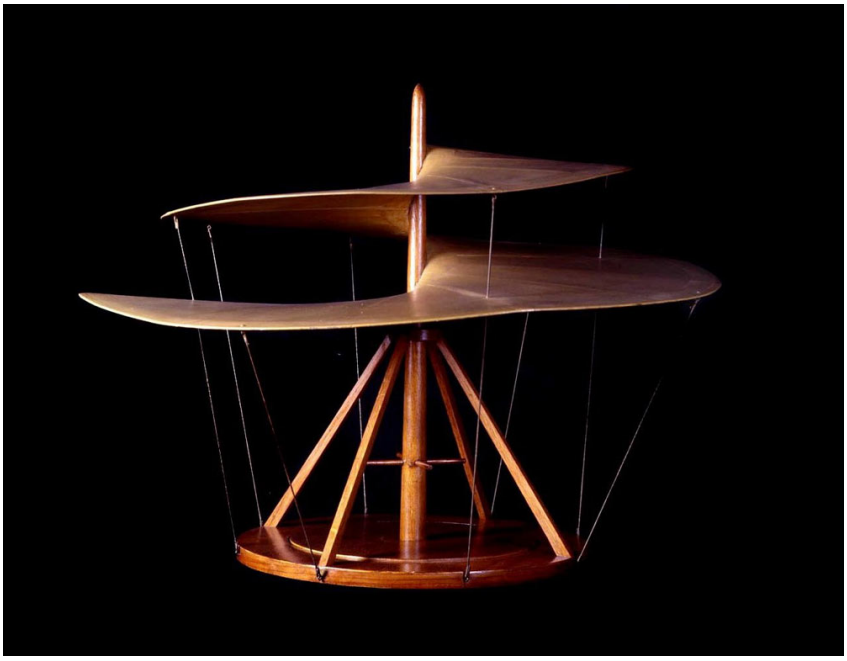

Figure 1.2: Model of Leonardo's aerial screw [2]

It was only in 1783 when the Montgolfier brothers were able to actually build a flying vehicle able to lift heavier weights like the human body, with hot air balloons. This means that anything that came before could be considered a rudimental UAV. Indeed, their definition is

autonomous flying vehicles without any man-composed crew, controlled via onboard computers or remotely.

In more recent times, the engines' development made it possible to build a wider variety of vehicles such as airplanes and helicopters.

From World War I, the technological development of modern drones was accelerated seeing the appearance of the first torpedoes. At that time, the main concern was reducing human life loss during attack operations, but since then, drone duties have increased significantly.

1.2 Group Flight History (UAV Swarms)

Swarming is a technique that has always been used by wild animals to achieve their objectives more efficiently. Being in a group, they can better defend themselves from predators, or, on the other hand, it is easier to trap prey by encircling it, for instance. Being in a swarm can also make it possible for individuals to rely on each other by specializing in different jobs.

As flying machines were invented by looking at and imitating nature, also this idea came from such observations. However, one of the biggest examples of animal swarming is human society, meaning that most of the theoretical knowledge is already available.

The history of UAVs swarming is relatively short, as they require more advanced communication technology.

The first drone groups made their appearance in military research at the beginning of the new millennium. At first, they were used for large-area observation, and in 2016 the US Department of Defence was able to launch a swarm of 103 drones capable of decision-making

and self-healing[18]. Now the research is working on using UAS (Unnamed Aerial System) and sUAS (small UAS) for commercial applications.

1.3 Group Flight Motivations and Advantages

Drone swarms can be homogeneous or heterogeneous, meaning that sometimes it is possible to have different drones in the same system for better workload distribution. Being in a swarm allows every drone to have a simpler task. As a consequence of that, a lot of advantages can be exploited:

- **Hardware Simplification**: the complexity reduction of the tasks for every unit will reduce the required hardware, making drones lighter, more reliable, and cheaper.
- **Mass production**: since more drones are needed to be produced, and because of their simpler hardware, it becomes possible to increase the production line efficiency. However, in heterogeneous swarms, mass production may not be as optimized as in the homogeneous case, although other advantages can be exploited.
- **Task specialization**: in a heterogeneous swarm, it is possible to build drones able to execute a specific operation of the job, making its hardware function-specific, thus even more simplified.

  Despite the optimal hardware, this solution may complicate the swarm's algorithm, as it will be needed to summon the suitable drones for what is required at that moment.
- **Fault tolerance**: A failure in a single UAV system can cause the complete system to stop working. In many cases, the whole vehicle can be destroyed or not retrievable.

A similar fate can be reserved for some elements of a UAS, but in this case, the system will just need some rearranging of elements (self-healing) to be able to cover the tasks previously assigned to lost vehicles.

Vehicles in a UAS can be less reliable than a single UAV system, as an eventual failure will not heavily impact the whole system by shutting it down, but will have only a low replacement cost.

1.4 UAS Applications

Most UAS applications are the same as UAVs but with better optimizations. They are listed below [19].

Figure 1.3: Drones flying over a field [3]

- **Agricultural monitoring**: being able to map large fields efficiently can be exploited for precise water distribution and plant disease containment, thus reducing the use of

water and reducing food losses. The map will also be handy in the advent of autonomous agricultural vehicles. As with every monitoring task, using a drone swarm rather than a single UAV will result in a quicker and more efficient mapping of large areas.

- **Construction**: research is developing a new construction system where humans cooperate with a swarm of drones, making the whole process more efficient.

  This is a clear example of a heterogeneous swarm, as drones are expected to have different roles, such as material delivery, inspection, and assembly of elements.

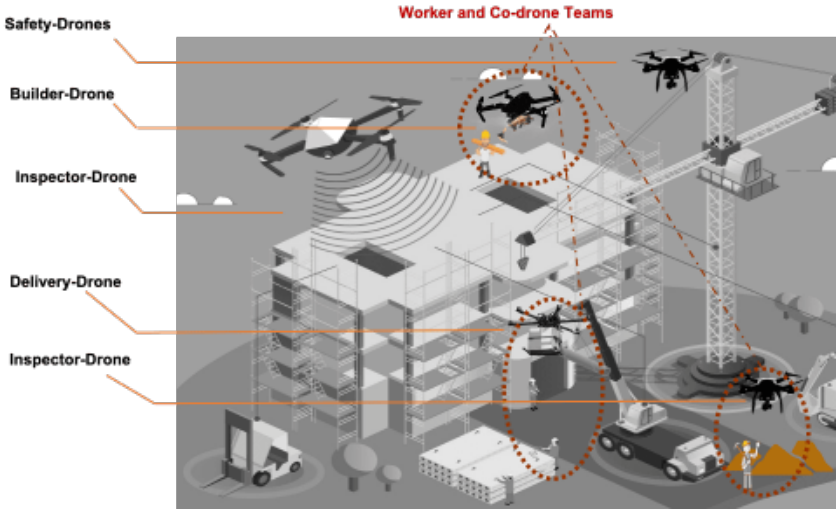

Figure 1.4: Concept of a heterogeneous swarm in a construction site [4]

- **Survivors research and rescue**: after a natural disaster, being able with a drone swarm to quickly map the area can significantly increase the chances of finding missing people still alive.

Figure 1.5: Representation of a drone swarm taking off for serving an emergency response [5]

- **Light show**: having multiple sUAVs makes it possible to use them as movable pixels for creating an interesting light show.
- **Packages Delivery**: Goods transportation logistics can be optimized and automatized using a drone swarm. Being a swarm will enable the system to complete multiple deliveries at the same time and manage the needs of every specific task independently. This last task implies that drones are lifting weights, meaning that here is where this thesis can be exploited.

Figure 1.6: Light show at Singapore marina for the Chinese New Year [6]

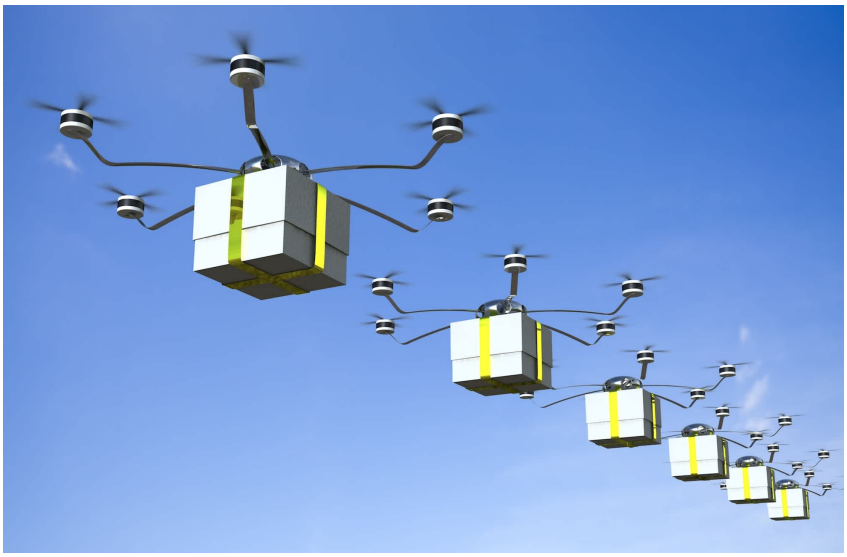

Figure 1.7: Concept of a drone swarm for package delivery [7]

CHAPTER 2

SYSTEM DESIGN PROCEDURES AND MODELS

To achieve the thesis objective, it was necessary to follow 2 different methods as the simulation model didn't always exist or could not be accurate. The model-based design procedure was used to develop all the algorithm sections that were uncorrelated with the drone interactions (line tracking and altitude control), while the Model-free design procedure was used for the drone interactions (that didn't have an existing simulation model) and for managing all effects not considered in the simulation models (battery discharge for instance). For the hardware structures, especially in the alternative altitude estimation, the design takes inspiration from the classic data path plus the control unit structure of the digital circuits.

2.1 Model Based Design

Many projects can be completely decomposed into known problems. This means that it is possible to create a good system model to speed up the design process through simulation.

Most of the time, when a controller design is needed, the model is already well known [20]. For example, a single autonomous vehicle is usually a well-known system that can be converted into a model. In reality, most of the time, the model is simplified and optimized to correctly simulate only the characteristics managed by the desired controller.

The model-based design approach is defined in such a way that the whole process is subdivided into simpler steps, with the hardware test only at the last step and concentrating most of

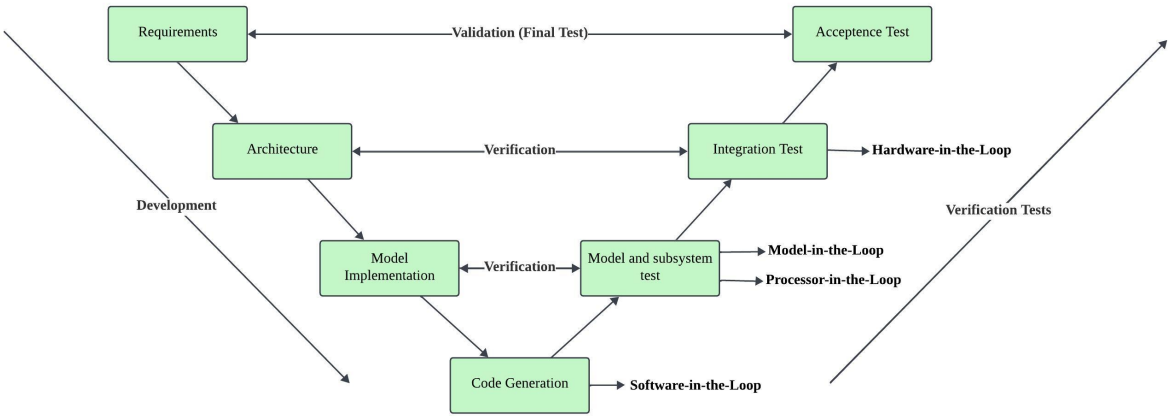

Figure 2.1: Model Based Design V Shape

the development actions in a software environment. As a consequence, it is possible to see that in the V model of this procedure (Figure 2.1), all the design takes place in the software section, while with the hardware, only validations and integration tests are done. This is because testing on the hardware is much more time-consuming than in a simulator.

2.1.1 Model-in-the-loop-test

Here, the main idea is developed. Starting from the specifications and using the model of the system as a simulation base, the main part of the project algorithm is developed. All actions of this step are done via software and in a simulation environment.

The objective of this step is to design all the software functions that will be used, including the controller.

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

Once the program is correctly set up, the next step is to apply it to the system model for simulation. If the model is correctly representing the real system, it will be possible to debug all the structures composing the project and validate them.

The objective of this step is thus to obtain a virtually working version of the project and will finish with code generation (C code).

At this stage, system optimization is also performed as it is sufficient to validate it through simulation.

2.1.3 Processor-in-the-loop test

Once the code is done, the program will run on the target hardware, but all data will be read by the simulator. The purpose of this stage is to validate the program within the target hardware.

At the end of this stage, everything will be ready for a first working test.

In this specific project, this step wouldn't be performed as the manufacturer already did a general certification, such that the hardware could be kept to an unopen-source version. In later tests, a vulnerability that wasn't fixed by the manufacturer in this process was found (see chapter 9.2.3).

2.1.4 Hardware-in-the-loop test

Now, all the actuators and peripherals are correctly connected, and the real system is tested completely. This stage has the objective of indicating and correcting all actuator-related

problems. Usually, problems found at this step are caused by the excessive simplification of the simulating model.

The objective of this test is to see if all unpredictable effects of the real-world application are contained and that everything works correctly.

2.2 Model free System design: a traditional approach

When the simulation model doesn't exist or it is very computationally expensive to simulate, a more traditional approach can be applied.

As said in chapter 6.1.2.11, this project's model, in fact, is simplified to correctly simulate only the path planning logic's results. As a consequence, drone interactions, battery discharge behavior, and drone imperfections are not simulated.

A model for drone interactions could have been obtained by representing the exact vehicles and applying the fluid dynamics theory. However, this approach would have been too time-consuming, making it steer to this more traditional procedure.

The other cases are much more unpredictable and weren't simulated as well.

2.2.1 Decompose the problem

The first thing to do is to understand the problem as a whole and try to decompose it into smaller functions. For each function, a measure procedure and expected results are defined. This will help set up "ad-hoc" experiments for each model section.

For every experiment, it is necessary to execute code generation, software debug, processor in the loop test, and finally, hardware in the loop tests, meaning that, on a smaller scale, every

section of the Model-Based Design (2.1) where model simulation is not performed, must be completed for ensuring a correct test.

The objective of this step is to theoretically analyze the model and plan a research process for finding all needed characteristics.

2.2.2 Execute Measures and Data Analysis

Once defined the tests to be performed, execute them to find multiple measures of every observed value. Data analysis must be performed to better understand their behavior.

Every observation in this step might have 3 main possible outcomes:

1. From the measured signals, it is possible to observe the searched characteristics, and some further analysis could be performed to obtain a result. This is the best-case scenario, as it is associated with the success of this step.
2. The analyzed quantity is not observable in the measured signals, meaning that no further analysis can be done. When having this result a change of observed signal or used approach is needed.
3. From the measured signals, it is possible to observe the searched signal, but the result is completely different than what is expected, or some consistent but not predicted behaviors are observed. This case is a bit more difficult to analyze, but it is usually correlated with an unexpected discovery or an error in the previous step.

   Based on the result, the next action of this step is often to come back to the beginning and define some new system analysis procedures. That might be caused by insufficient information or failure to analyze a signal.

2.2.3 Measures Validation

For better project reliability, after having gathered enough knowledge about every needed phenomenon, a set of new tests is defined. This step becomes extremely crucial when values are observed after the test (like in this project).

Usually, the validation tests are very similar to the measuring procedures, but the difference is that the system must automatically retrieve the desired information from the previously analyzed signals to demonstrate its sufficiently high SNR.

If this stage is successful, it is now possible to use the newly defined procedure in the actual algorithm.

2.2.4 Actual System Design

Now, the design procedure becomes similar to the model-based design again, even if only software debugging is not performed on the system.

2.2.4.1 Software-Debug

Here, the whole algorithm must be checked for bugs and mistakes. If one of these is not fixed and comes out later, it will be difficult to understand its origin.

Using a test bench for each section can be useful at this stage.

2.2.4.2 Processor-In-The-Loop-Test

This step is exactly the same as in the Model-Based-Design 2.1.3.

2.2.4.3 Hardware-In-The-Loop-Test

This is where most of the design time is spent. Not being able to perform simulations, most of the tests arrive at this much slower stage and need to be correctly analyzed. Preliminary debugging and a safe testing environment are crucial in this case, as an error that might damage the real system is much more likely.

In the end, if there is a solution, a working project is obtained.

2.3 Data path and Control Unit Hardware Structure

This is the classic way of structuring digital circuits. They are divided into 2 sections, which split the job of managing all the calculations and managing the process flow. In the alternative altitude estimation, the system structure was heavily inspired by this design model.

2.3.1 Data Path

This is the main body of the circuit where all the functional units are located. Here, the circuit is responsible for actually executing operations, but it needs the Control Unit's directions to work correctly.

2.3.2 Control Unit

This element is the "brain" of the system; it is not connected to any output and is in charge of enabling all operations based on the system state. A finite-state machine is usually the core of this section.

2.4 Project goals

The research performed in this thesis project aims to design a model-free algorithm through the available Matlab and Simulink platforms that can program a Parrot Mambo mini-drone.

Using **Paolo Ceppi's thesis work** [20] as a base point, drones were kept limited to a straight line because it is necessary to guarantee a precise interaction for a reliable measure. Measures must be performed by exploiting some prototyping algorithms developed only for the specific purpose.

Once initial explorations are performed, data must be analyzed and validated to define a system model to be used in the actual algorithm.

With that done, the tracking procedure must be defined to complete the whole objective.

This project doesn't have any particular specifications other than finding a working solution, as this was an unknown research topic.

As the drones used weren't designed for group flights, the sensor disturbance problem must be solved. In parallel to the previous objectives, an alternative altitude control algorithm needs to be designed as the ultrasonic sensor used for that role is not working anymore when 2 drones are interacting.

Such a design can be initially crafted following the model-based steps. However, it's important to note that the simulation model used in this process was not entirely accurate. This lack of precision is justified by the fact that the primary objective of this simulation model was not to develop altitude estimators. Therefore, the final stages of the design development need the implementation of tests in real-world conditions.

This section also doesn't have any particular specification other than the final result must work.

2.4.1 Additive Project Goal

Once obtained a working solution of the drone following algorithm it was noted that speed patterns of the leader can be read by the follower. This is nothing different from an implicit communication channel. As a result of this idea, an additive objective is to demonstrate that it is possible to send a message through this newly invented (completely by me, Leonardo Cerruti) communication channel.

CHAPTER 3

MODELING AND CONTROL OF SINGULAR AND QUADCOPTERS SWARMS

In this chapter, all the fundamental aspects of modeling a quadcopter are briefly explored with an eye on the drone swarm structure.

A drone has a number of main components:

- Frame: this is the main physical structure that keeps the vehicle parts together. This definition can be used as a synonym of the drone's local system of reference, as the latter is defined on this element.
- Rotors: their number depends on the model, but in the case of a quadcopter, they are 4 and are coupled in the 2 opposite rotating directions for having 0 total resulting angular moment and better controlling the yaw (see section 6.1.2.6).
- Battery: This component must be as light as possible as it is one of the heaviest overall. Usually, that requirement means having a very high ripple on the voltage output (see section 5.2) and very low energy, resulting in a short flight time.
- Electronic Board: here, all control and movement algorithms are uploaded and executed using the rotors as final system actuators (for Parrot Mambo, see section 5.4.1). This electrical part contains all the system sensors used for position and movement estimation, such as the accelerometer, gyroscope, camera, and sonar.

3.1 Brief Drone Flight Control Description

Depending on the model, these specific controls' architectures may change. A drone has 6 control algorithms working in parallel, one for each degree of freedom. In this specific case, the couples pitch-Y and roll-X have the same controller because, in the way a quadcopter is done, the position is controlled with those angles' movements, and such angles must be almost always at around zero to ensure system stability.

The remaining 4 controllers are able to work in parallel and their outputs are summed together at the rotors' command signals.

- **Altitude Control (Z)**: this controller has the same output to all 4 rotors as it is controlling the total thrust.
- **Pitch Control**: Depending on the direction, the 2 front or back engines are increased in power, but to not influence other aspects, the other two engines' power is decreased. This produces a torque around that angle, and such inclination will also cause an X movement.
- **Roll Control**: This is very similar to the pitch angle control, but it manages left and right engines.
- **Yaw Control**: To have ideally 0 total angular moment, in a quadcopter, the rotors are spinning 2 in one direction and 2 in the opposite. Blades spinning in the same direction share the same diagonal as this is the most stable configuration. To control the yaw, the angular moment is purposely put out of balance: the 2 engines spinning in the direction of desired rotation are increased in power, while the others are decreased such that there

is no effect on the total thrust and on the other angles, but there is on the total angular moment. Now, the yaw will change.

3.2 Space Representation

For an easier calculation of all aspects of singular and multiple drones, a standard was defined that uses the following systems of reference:

- World: this is the absolute system of reference. It is based on the earth (or the room), and it is considered a fixed orthogonal triplet. Axis are oriented such that the main forward direction of the path is x_w, the direction pointing left (while looking forward) is y_w, and z_w is pointing up.
- Drones: every drone will have its own system of reference. For better coherence with the world coordinates, this system is defined with x_i representing the forward direction of the vehicle "i", y_i the left direction, and z_i its altitude. The origin of this system (O_i) will correspond to the center of the drone. In the Parrot System, the z and y axes are inverted: for environmental calculation, it is preferred to use the system defined now and convert it back to the drone program at the end.

Coordinate frames	
O_i	Origin of the reference frame "i" axes
L	Local system of references
W	World system of references
Position and Orientation Vectors	
$\underline{\mathbf{p}_i} = (x_i, y_i, z_i)$	Reference axes of system i
i_a, j_a, k_a	Unitary reference axes vectors of system "a"
ρ_i, ϕ_i, θ_i	Orientation of the system i (roll, pitch, yaw in order)
$\underline{\mathbf{p}_j^i} = (x_j^i, y_j^i, z_j^i)$	Position vector of the i origin in the j system
$\underline{\alpha_j^i} = (\rho_j^i, \phi_j^i, \theta_j^i)$	Orientation of the i frame in the j system
$\underline{T_j^i} = (\underline{\mathbf{p}_j^i}, \underline{\alpha_j^i})$	6-DOF system "i" pose vector in the system j
Transformation Vectors	
$\underline{\underline{\mathbf{R}_j^i}}$	Rotation matrix of the i frame in the j system
$\mathbf{r}_{ab}^{ij}$	Projection of the i vector of the "a" frame to the j vector of the "b" frame
$\underline{\mathbf{q}_j^i}$	Quaternion representing the rotation from system i to system j
Movement vectors	
$\underline{\mathbf{v}_j^i}$	Velocity vector of the i origin in the j system
$\underline{\mathbf{a}_j^i}$	Acceleration vector of the i origin in the j system
Control variables	
$\underline{\mathrm{K}}$	PID Control Gains: K_P Proportional, K_I Integral, K_D Derivative
Misc. variables	
g	Scalar value of gravitational acceleration

TABLE I: Table of variables used for mathematical representations

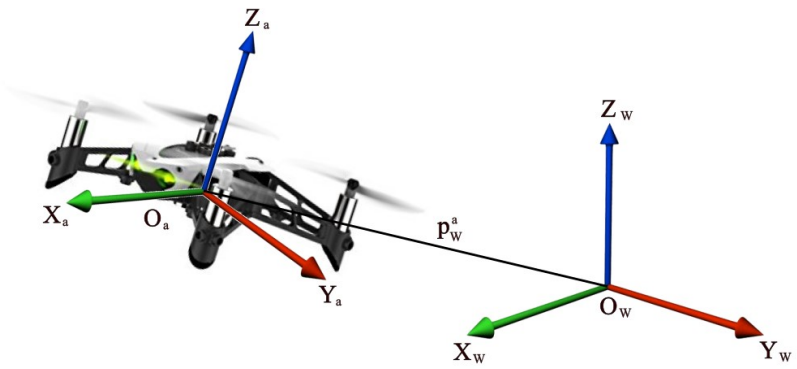

Figure 3.1: Example of local and world system of references

3.2.1 Space Transformations

Using a multiple-systems-of-reference environment will require defining some frame transformations [21]. In the World reference system, it is possible to represent the position of every drone (origin of its local coordinates system) with the positional vector $\mathbf{p}_w^i$ (Where "i" indicates the drone's local system). Similarly, it is possible to find the relative position between elements using a drone as the main system (3.3.1). Since changing the reference system of multiple points doesn't effectively change their relative positions, it could be useful to execute all relative calculations regarding them in the same coordinate system (usually, everything is done in the world system), as the results are differential and wouldn't change. All these concepts apply to the translation movement as well because the only thing to change would be the translation vector.

Now, the relative orientation must be found. Most of the time, all systems will have different orientations in continuous change: rotational transforms will need to be applied.

For calculus simplicity and because drone rotation always happens around their center of mass, all transformations and future movement rotations will be considered to happen around a system's origin. Furthermore, the later explained quaternion theory requires the rotational axis to contain the origin.

The first thing to do with the rotational transform is to assume that the origin of the initial and target systems are at the same point. An easy solution is to use the initial system origin position vector in the target coordinates, translate the target system's origin, apply the rotational transform using the newfound translated system as a target, and translate the result back to the actual target system.

Now that the 2 origins O_a and O_b are superposed, it is possible to define the axes' (i_a, j_a, k_a to i_b, j_b, k_b) transformations:

$$i_b = i_a * r_{ab}^{ii} + j_a * r_{ab}^{ji} + k_a * r_{ab}^{ki} \tag{3.1a}$$

$$j_b = i_a * r_{ab}^{ij} + j_a * r_{ab}^{jj} + k_a * r_{ab}^{kj} \tag{3.1b}$$

$$k_b = i_a * r_{ab}^{ik} + j_a * r_{ab}^{jk} + k_a * r_{ab}^{kk} \tag{3.1c}$$

At this point, it is quite clear that the **rotational matrix** $\underline{\underline{R_{ab}}}$ has the form:

$$\underline{\underline{R_{ab}}} = \begin{bmatrix} r_{ab}^{ii} & r_{ab}^{ji} & r_{ab}^{ki} \\ r_{ab}^{ij} & r_{ab}^{jj} & r_{ab}^{kj} \\ r_{ab}^{ik} & r_{ab}^{jk} & r_{ab}^{kk} \end{bmatrix} \tag{3.2}$$

This definition has 9 degrees of freedom, which is too many for the used application; in fact, this transformation works on the 3 axes independently, while it is well known that those will always be an orthogonal triplet.

Following this reason, the problem can be decomposed into 3 rotations around the 3 axes x, y, and z (respectively roll pitch and yaw), and 3 simplified matrixes are obtained:

$$\underline{\underline{R_x}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\rho) & sin(\rho) \\ 0 & sin(\rho) & cos(\rho) \end{bmatrix} \quad \underline{\underline{R_y}} = \begin{bmatrix} cos(\phi) & 0 & sin(\phi) \\ 0 & 1 & 0 \\ sin(\phi) & 0 & cos(\phi) \end{bmatrix} \quad \underline{\underline{R_z}} = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

Applying these rotations consecutively makes it possible to obtain a composed matrix for the total rotation. Notice that the final result depends on the matrix's order in the multiplication.

$$\underline{\underline{R_x}} \cdot \underline{\underline{R_y}} \cdot \underline{\underline{R_z}} = \begin{bmatrix} cos\phi \cdot cos\theta & cos\phi \cdot sin\theta & sin\phi \\ cos\theta \cdot sin\rho \cdot sin\phi + sin\theta \cdot cos\rho & sin\theta \cdot sin\rho \cdot sin\phi + cos\rho \cdot cos\theta & sin\rho \cdot cos\phi \\ cos\theta \cdot cos\rho \cdot sin\phi + sin\rho \cdot sin\theta & sin\theta \cdot cos\rho \cdot sin\phi + cos\theta \cdot sin\rho & cos\rho \cdot cos\phi \end{bmatrix} \tag{3.4a}$$

$$\underline{\underline{R_z}} \cdot \underline{\underline{R_y}} \cdot \underline{\underline{R_x}} = \begin{bmatrix} cos\phi \cdot cos\theta & cos\theta \cdot sin\rho \cdot sin\phi + sin\theta \cdot cos\rho & cos\theta \cdot cos\rho \cdot sin\phi + sin\rho \cdot sin\theta \\ cos\phi \cdot sin\theta & sin\theta \cdot sin\rho \cdot sin\phi + cos\rho \cdot cos\theta & sin\theta \cdot cos\rho \cdot sin\phi + cos\theta \cdot sin\rho \\ sin\phi & sin\rho \cdot cos\phi & cos\rho \cdot cos\phi \end{bmatrix} \tag{3.4b}$$

A different and more optimized way to define a rotation is by using the **quaternions**. These are the evolution of the bi-dimensional complex numbers and follow the definition:

$$\underline{\mathbf{q}} = a + b \cdot i + c \cdot j + d \cdot k \tag{3.5}$$

where [22]

$$i^2 = j^2 = k^2 = -1; \quad i \cdot j = -j \cdot i = k; \quad j \cdot k = -k \cdot j = i; k \cdot i = -i \cdot k = j \tag{3.6}$$

Relationships between the imaginary parts of the quaternion are the reason why the product is non-commutative.

The Euler's formula (Equation 3.7a) can be developed to work in this domain (Equation 3.7c) as well:

$$\underline{\mathbf{c}} = a + i \cdot b = h \cdot \left[cos\left(\frac{\theta}{2}\right) + i \cdot sin\left(\frac{\theta}{2}\right) \right] = h \cdot e^{i \cdot \frac{\theta}{2}} \tag{3.7a}$$

$$\underline{\mathbf{q}} = a + b \cdot i + c \cdot j + d \cdot k = h \cdot cos\left(\frac{\theta}{2}\right) + h \cdot (u_x \cdot i + u_y \cdot j + u_z \cdot k) \cdot sin\left(\frac{\theta}{2}\right) \tag{3.7b}$$

$$\underline{\mathbf{q}} = h \cdot e^{(u_x \cdot i + u_y \cdot j + u_z \cdot k) \cdot \frac{\theta}{2}} \tag{3.7c}$$

Notice that since these transforms are only about rotation angles (direction vectors rotation), it means that all values are normalized and $h = 1$. Furthermore, in the quaternion, it is possible to distinguish between the rotation axis (that passes through the origin):

$$\left(\frac{b}{sin\left(arcos(a)\right)}, \frac{c}{sin\left(arcos(a)\right)}, \frac{d}{sin\left(arcos(a)\right)} \right) \tag{3.8}$$

and the rotation angle $\gamma = arcos(a)$.

All quaternions can be translated into a rotation matrix [23]:

$$\begin{bmatrix} 2a^2-1+2b^2 & 2bc-2ad & 2bd+2ac \\ 2bc+2ad & 2a^2-1+2c^2 & 2cd-2ab \\ 2bd-2ac & 2cd+2ab & 2a^2-1+2d^2 \end{bmatrix} \tag{3.9}$$

As for the rotation matrixes, the quaternion product (or quaternion composition) depends on the order of multiplication, and it is calculated in a scalar way and collapsed following the relationships at Equation 3.6:

$$\underline{\mathbf{q_1}} \cdot \underline{\mathbf{q_2}} = (a_1 + ib_1 + jc_1 + kd_1) \cdot (a_2 + ib_2 + jc_2 + kd_2) \tag{3.10a}$$

$$\begin{aligned} \underline{\mathbf{q_{res}}} = a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 + (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)\mathbf{i} \\ +(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)\mathbf{j} + (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)\mathbf{k} \end{aligned} \tag{3.10b}$$

3.2.2 Dynamics Transformations

In this project, for better interaction reliability, all of the drone movements were reduced to quasi-translational, meaning that the speed and acceleration for an element "i" can be represented in different systems as a derivation of the position vector. Rotational movements are kept to a minimum.

$$\underline{\mathbf{v}^{\mathbf{i}}_{\mathbf{w}}}(t) \approx \frac{\partial \underline{\mathbf{p}^{\mathbf{i}}_{\mathbf{w}}}(t)}{\partial t} \tag{3.11a}$$

$$\underline{\mathbf{a}^{\mathbf{i}}_{\mathbf{w}}}(t) \approx \frac{\partial^2 \underline{\mathbf{p}^{\mathbf{i}}_{\mathbf{w}}}(t)}{\partial t^2} \tag{3.11b}$$

3.3 Quadcopters Swarms

As this project aims to use multiple drones in the system, a few more concepts must be introduced to correctly represent a multiple structure.

3.3.1 Relative Bearing

From a robot "i" reference system, the relative bearing is the T_b^a 6-DOF Pose of another robot $(\mathbf{p}_b^a, \alpha_b^a)$ in such a system (Example in Figure 3.2). This concept also includes the rotational characteristics of the target [24].

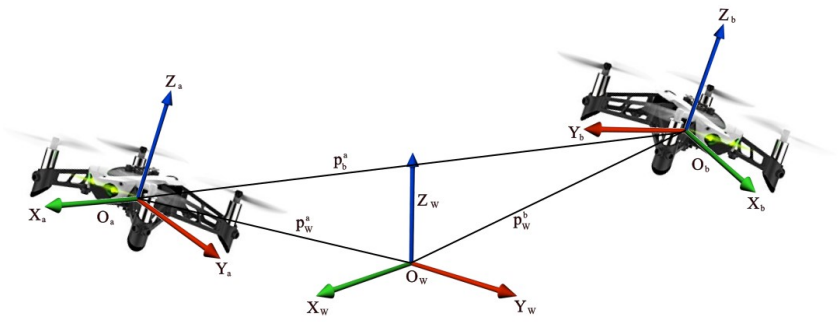

Figure 3.2: Example of local and world system of references

In the used project, this definition will have the 3 degrees of freedom about the orientation at ≈ 0. For this reason, all the relationships with speed and acceleration will be similar to Equation 3.11.

3.3.2 Swarm system

In case drones need to fly in formation, it is possible to define the coordinate system of the swarm. This is useful for managing only the relative positions.

In this project, the swarm reference corresponds to the leading drone coordinates system.

CHAPTER 4

PREVIOUS WORK

This whole project research is not starting from zero, but it is using as a starting point Mr. Paolo Ceppi's Thesis project, "Model-based Design of a Line-tracking Algorithm for a Low-cost Mini Drone through Vision-based Control" [20], where he successfully completed the "MathWorks Minidrone Competition" by working on the given template [25].

There are 2 main reasons why this previous project was used:

- Having a ready-to-fly platform to start working on: this means that all the control tuning steps were already done.
- Ensuring a correct position of the drones: By just setting "$y = 0$" in the commands, the built-in sensors and algorithms are not good enough to keep the drone in place, and the whole system will derivate. Using the line tracking function previously implemented, it is possible to have a more reliable positioning. (This tracking algorithm was then replaced later for software optimization needs)

This section is meant to indicate how and where this introductive work was used and developed.

4.1 Controller Tuning

To control all aspects of the quadcopters' 6 DOF, Mathworks set up a combination of controllers that were correctly tuned and set up by Paolo. Such control combination is reported in Figure 4.1, and the only modifications applied in this project are the addition of the output

highlighted in green and an introduction of calibration weights in the block "thrustsToMotor-Commands" (see calibration section in chapter 9). The former is needed for cleaner drone detection (7.1), and the latter is used for ensuring a take-off on the spot.

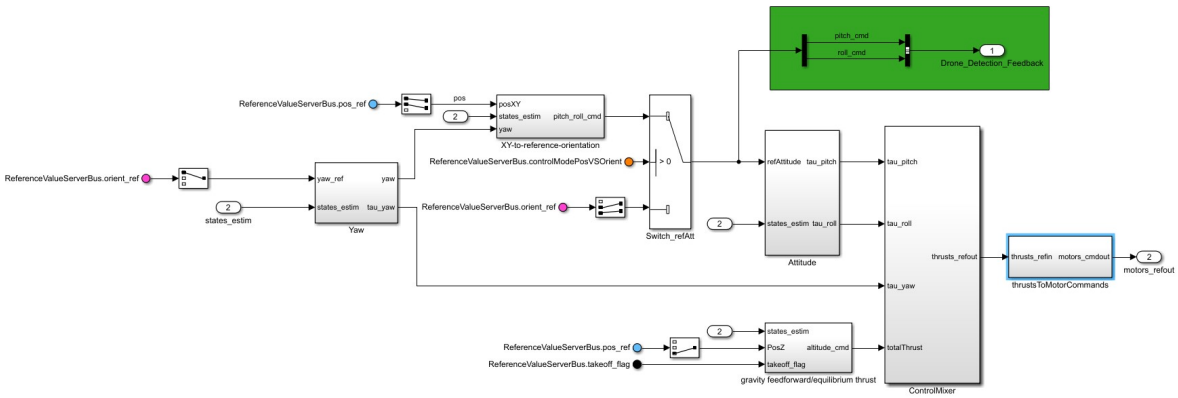

Figure 4.1: Full control of the 6-DOF.

4.2 Path Planning Logic

Here, the original logic was heavily modified, but there are still some sections coming from previous work.

- kept and reused elements: the whole section about tracking y of the drone was kept intact. Furthermore, the Matlab function "Tracking", used for setting gains on the Y position, was copied and reused for yaw control. Also, the final section, where the signals are added to the current ones and collapsed into "pos_ref," was kept intact.

  Reused parts are underlined in Figure 4.2.

- Landing logic: this whole section was initially removed as detecting a location to land was not a requirement of this project. A simplified version was implemented in the Z

execution unit (Figure 8.8) to test the ”land” command’s correct detection (See chapter 8.1.4.2).

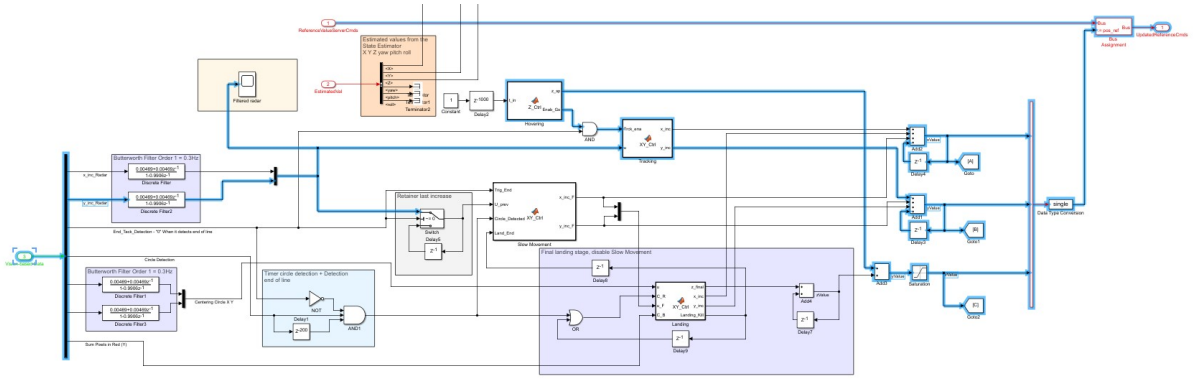

Figure 4.2: Reused sections of the previous path planning block.

4.3 Image Processing System

Because this section needs to be optimized, the original work was almost completely replaced. The section about detecting the end of the track was completely removed, as in the path planning block. The tracking part was utilized only in the initial test and was heavily modified by the optimization problem.

CHAPTER 5

HARDWARE DESCRIPTION AND FIRMWARE SETUP

For the fulfillment of all project goals, 2 different drone hardware platforms were used. This chapter will start by describing the main electrical components of a quadcopter in general and will conclude by describing more specifically the 2 used platforms.

5.1 Sensors Description

Depending on the model's budget, a drone might have a different set of sensors. The main ones' working characteristics used by the 2 platforms are briefly described here.

5.1.1 Ultrasonic Sensor

An ultrasonic sensor (commonly known as Sonar) is mounted on the Parrot Mambo platform for better altitude estimation. However, this sensor causes a lot of problems (see sections 5.1.1.2 and 6.1).

5.1.1.1 Working Principle

Every model of this sensor is composed of a transmitter and a receiver, which could be 2 separate elements or a single one called a transceiver [26]. The working principle is very simple and can be divided into 3 phases:

1. **Emission**: after a trigger signal happens, a short sonic burst at a frequency between 23 kHz and 40 kHz is transmitted. The frequency must be carefully chosen at design time

so as not to interfere with any disturbance effect, such as air movement caused by rotors, and be recognizable.

2. **Reflection**: the short sonic burst bounces on the target object (ground in this application) and starts coming back to the receiver.

3. **Reception**: when the reflected burst arrives at the receiver, it is identified thanks to its frequency. A signal called echo is activated, and it is received by the microcontroller as an interrupt.

Once the echo signal is received by the micro-controller, it is possible to measure the time difference and calculate the overall space distance (h) following the formula in Equation 5.1, where V_s is the sound speed in the air (344 m/s), t_{echo} is the time stamp in seconds when the reception happens and t_{trig} is the time stamp when Emission was performed. (The whole result is obviously divided by 2 as the space covered by the sound is twice the measure)

$$h = \frac{V_s \cdot (t_{echo} - t_{trig})}{2} \tag{5.1}$$

5.1.1.2 Sonar Beam Angle and Consequences

Sound signals are a composition of pressure waves that usually propagate following a cone shape. The beam angle β represents the slopes of the cone with respect to the central direction of the beam (see Figure 5.1). This angle is the reason why a drone that uses an ultrasonic sensor for altitude estimation can't fly correctly when too close to walls or other drones with sonar (6.1). Calling "d" the distance with the closest wall, "h" the flying altitude, and β the ultrasonic

beam angle, the maximum possible flying altitude can be found following Equation 5.2, and the concept is well visible in Figure 5.1. There, it is possible to see from left to right:

1. A drone flying at an altitude low enough to avoid wall reflections
2. A drone flying at the maximum possible altitude without wall interferences. In this case, the sound reflected by the wall and the one reflected by the floor takes the same time.
3. A drone experiencing wall reflection interference. In this case, the signal reflected by the wall arrives before the one reflected by the ground, meaning that it is considered valid. As a result, the drone will climb up indefinitely as it requests an altitude higher than the sonar's measurement (that will never increase as long as the wall is that close).

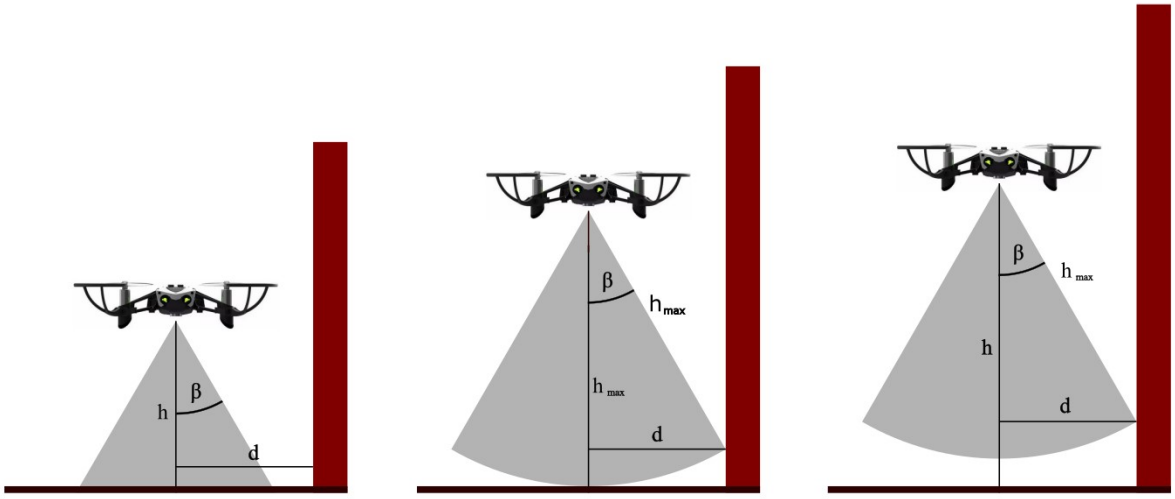

Figure 5.1: Visual explanation of the relationship between beam angle, wall distance, and maximum altitude

$$h_{max} = d \cdot arcsin(\beta) \tag{5.2}$$

Real walls are not perfectly flat, meaning that the sound is also reflected in directions different than "$180^o - incident\_angle$" with respect to "Z_W".

5.1.1.3 Reducing the Beam Angle

As only a small room was available, and a flying altitude higher than h_{max} was needed for keeping a safe vertical distance between the drones, the beam angle had to be reduced. The solution was to mount a sponge around the sonar transceiver that would filter out the wider section of the ultrasonic beam (see Figure 5.2). This solution was able to increase the maximum altitude from around 1 meter to 1.5 meters.

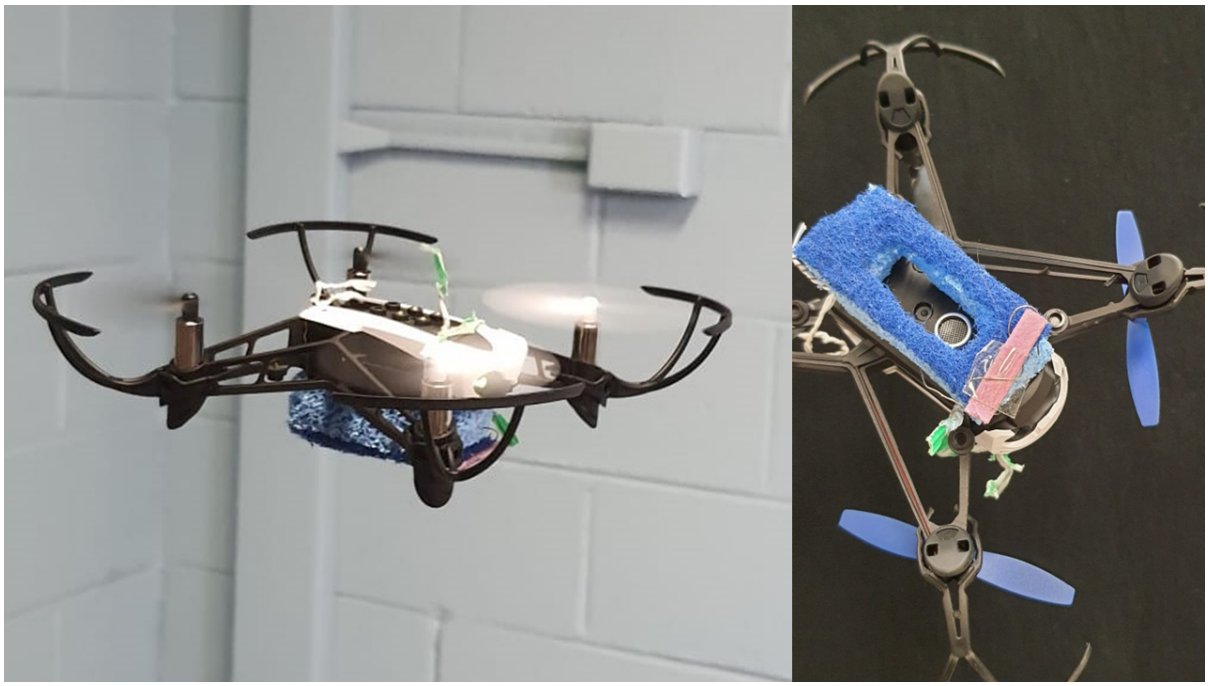

Figure 5.2: Modified drone for a reduced beam angle (on the sides)

5.1.2 Gyroscope and Accelerometer

These two sensors exploit a similar working principle as they both measure acceleration but in an angular or linear way. Multiple hardware versions of these sensors exist, but the most common one exploits the differential capacitors [27], [8].

5.1.2.1 Differential Capacitors

As shown in Figure 5.3, these devices are a double capacitor with a moving central element. The central element movement happens because it is not completely secured to the object thus, it is sensitive to inertial forces.

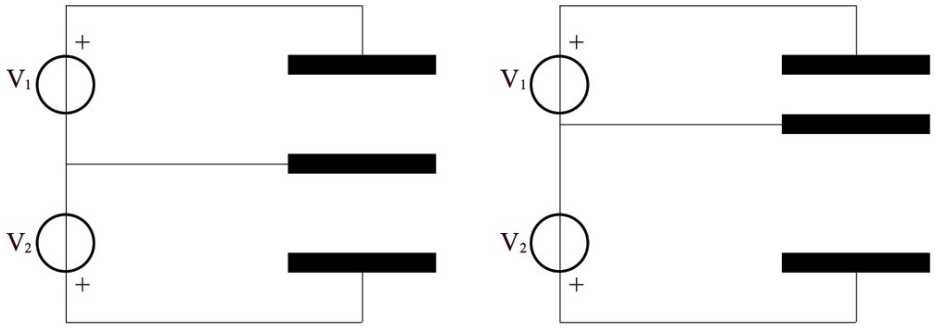

Figure 5.3: Differential capacitor representation with moving central plate

When the central plate is moving, the 2 coupled capacitors will change capacitance accordingly as the distance between their plates changes. Forcing to always have the same voltage will thus cause a current whenever the section is moving.

$$C(t) = \frac{A}{d(t)} \cdot \epsilon \tag{5.3a}$$

$$Q(t) = C(t) * V \quad \Longrightarrow \quad Q_1(t) = \frac{k}{d_1(t)}; \quad Q_2(t) = \frac{k}{d_{tot} - d_1(t)} \tag{5.3b}$$

$$I_1(t) = \frac{\partial Q_1(t)}{\partial t} = \frac{-k}{[d_1(t)]^2} \cdot \frac{\partial d_1(t)}{\partial t} \tag{5.3c}$$

$$I_2(t) = -\frac{\partial Q_1(t)}{\partial t} = \frac{-k}{[d_{tot} - d_1(t)]^2} \cdot \frac{\partial d_1(t)}{\partial t} \tag{5.3d}$$

As well visible in Equation 5.3, the current in each generator is correlated to the speed of the central plate, and as that object movement is correlated to the inertial force, that is correlated with the external acceleration, it is possible to obtain information about the system movements.

As the final measurement is the acceleration, to obtain absolute measures, these sensors are not very reliable and would deviate at a certain speed.

5.1.2.2 Accelerometer

This element [27] consists of a linear structured differential capacitor and works following the principle explained in chapter 5.1.2.1. In Figure 5.4, it is possible to see the moving section of the device in orange and the fixed electrodes in light blue. For measuring acceleration in all directions, it is possible to create a system of 3 accelerometers, one for every spatial axis, and use vectorial composition to estimate all possible directions.

5.1.2.3 Gyroscope

Instead of the linear acceleration, this sensor measures the centripetal acceleration, so the structure is slightly different [8]. The architecture reported here belongs to the Vibration Gyro Sensors described by EPSON [8], where the **Coriolis force** principle is exploited. Considering

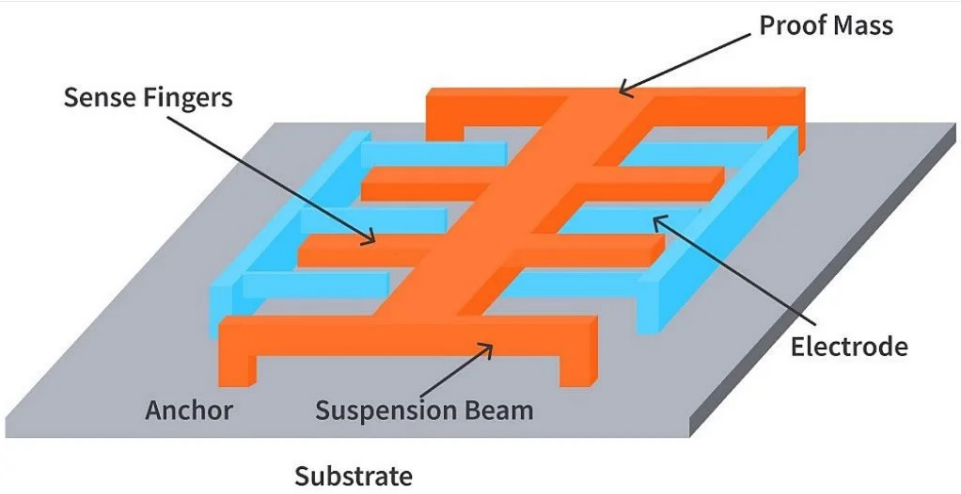

Figure 5.4: Internal structure of an accelerometer

a system in rotational movement, every object not bundled to it and trying to move linearly will be observed to have a lateral acceleration called Coriolis force.

$$a_{cor} = 2 \cdot \omega \cdot v_{\perp} \tag{5.4}$$

As well visible in Equation 5.4, this acceleration is directly related to the rotational speed ω. The perpendicular speed to the rotation axis $v_{\perp}$ will need to be induced to cause the Coriolis force, and it is known intrinsically.

The vibrating Gyro sensor structure is visible in Figure 5.5, together with the measuring phases.

1. At first, the sensor is not rotating: the lateral "T" structures are vibrating symmetrically such that the 2 coupled capacitances are varying symmetrically, keeping similar values.
2. When a rotation starts to occur, the Coriolis force will induce a new vibration perpendicular to the initial one on the "T" structures.

3. As all elements are physically connected, the vertical vibration will cause another vibration on the central rod. In the end, the 2 coupled capacitances of the differential capacitors will not be equal anymore, inducing an electrical signal following the same previously explained principle.

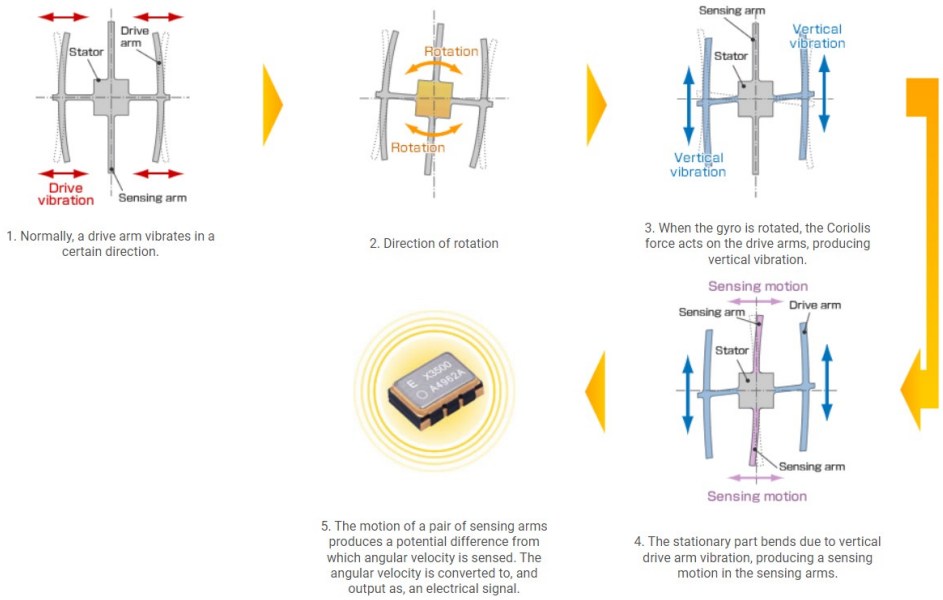

Figure 5.5: Vibrating gyroscope double T structure [8]

5.2 Lithium Battery

For a flight unbounded from the power network, a lithium battery is mounted in the drones. As the weight of the vehicle affects the flight time, the batteries are made as light as possible, meaning that their output will have a large ripple. All lithium batteries (while discharging) follow a similar output voltage shape depending on their discharge point. The curve is affected

by a lot of random effects, making it change every time, but it also follows a general shape similar to the "sigmoid function".

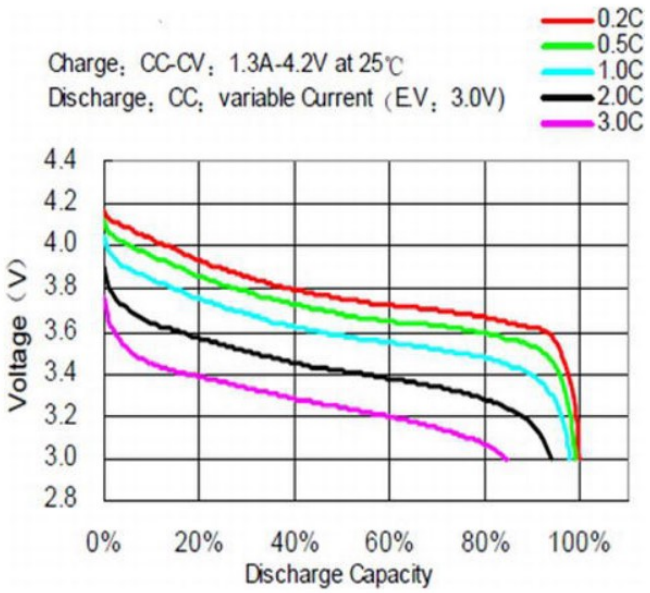

Figure 5.6: Examples of battery discharge voltage curves given the output current [9]

As the voltage is correlated with the output power of the engines in a quadratic way, to keep the same altitude, all engines will have to increase the requested power to keep the same effective output (see 6.1.2.11).

5.3 Engines

Designers used DC motors as output actuators for spinning the blades. Such elements are also called brushed motors as they are able to invert the internal coil current using a commutator that contains brushes and changes coil points of contact thanks to the rotation [10].

The working principle of these motors is to exploit the current flow running in a copper coil to generate a magnetic field (Biot-Savard law) that, coupled with the permanent magnet in the engine body, produces a spinning moment. If the current direction didn't change, the situation would be similar to having 2 permanent magnets and the result would be that these 2 magnets will just align. Current inversion will invert the magnetic field, and if that is done at the proper moment (the commutator is aligned correctly), the spinning movement will continue.

There are two main models of brushed motors, and they are briefly described now.

5.3.1 Typical Motor

This is the older design shown in Figure 5.7: its copper coil is bounded around a metal body that works as a structural support. This kind of engines can have multiple coils for better efficiency. However, more coils mean more brush contacts, and in small designs, this might not be possible.

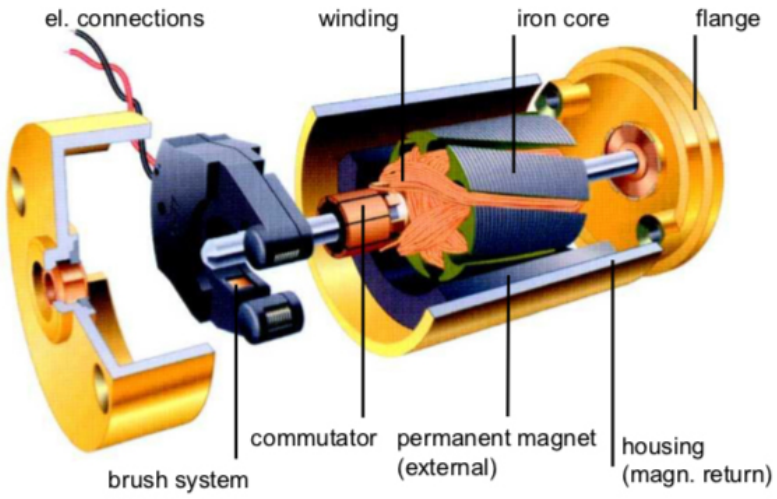

Figure 5.7: Representation of a generic Brushed motor [10]

5.3.2 Coreless Motor

A newer design wants to use the coil as a structural element of itself (see Figure 5.9), removing the iron core. This is possible by using epoxy to harden the structure. The permanent magnet is moved to the center of the engine (occupying the space of the old iron core), and this whole design has some advantages overall:

1. **Lower inertia**: due to the reduced weight of the spinning section of the motor, this solution will have less intrinsical inertia, meaning that acceleration and braking can be more efficient.

2. **Lower magnetic losses**: not having the iron to disturb the magnetic field, the efficiency is increased. When the iron is located in a periodically inverting magnetic field, it is possible to observe its hysteresis cycle. When a magnetic field is applied, the metal will consume part of that energy to magnetize itself. This attitude depends on the previous magnetization status and the instant magnetic field.

   In some materials when changing magnetization, their structure will also change size (It gets longer in the direction of the field depending on its strength). This could cause additional vibrations in the engine.

   The full cycle happens when the magnetic field is periodic (Figure 5.8), and its area represents the work consumed for magnetization changes [11].

   This factor can be synthetically indicated as "iron magnetic losses".

3. **More compact design**: having removed a substantial part of the motor weight, this will only improve the efficiency of an eventual vehicle using it.

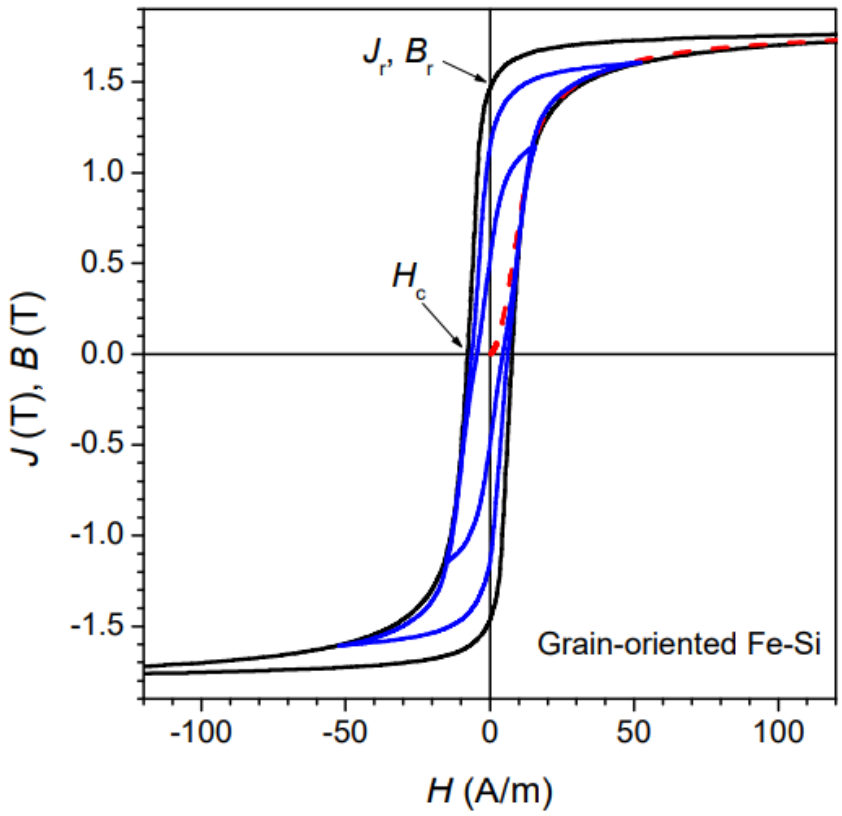

Figure 5.8: Example of a hysteresis loop from a grain-oriented material (one of the 3 main types that are used in transformers and engines). H is the coercive field (correlated with the magnetization), and B is the applied magnetic field. Blue cycles indicate execution with lower peak-to-peak magnetic field applications. [11]

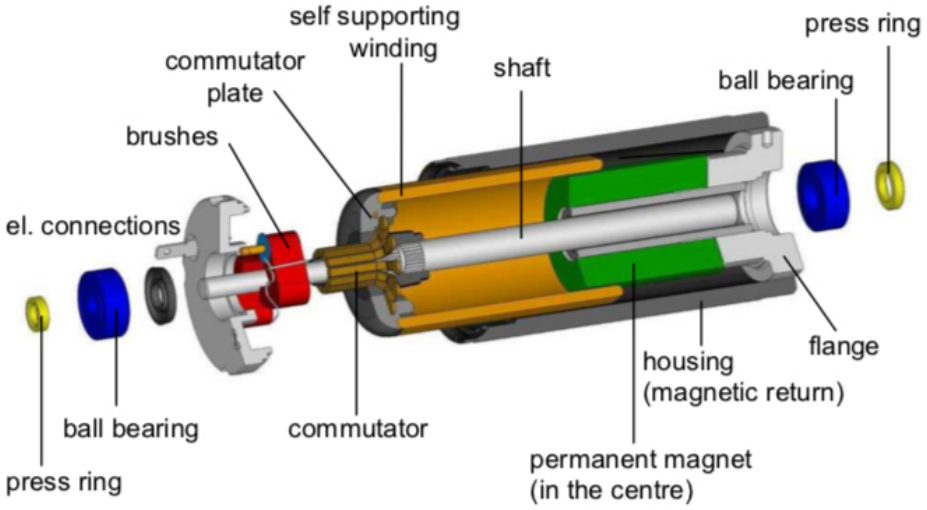

Figure 5.9: Representation of a generic Brushed motor [10]

5.4 Parrot Mambo

The main used element in this project is the Parrot Mambo (Figure 5.10), a cheap mini drone platform that exists with its own Simulink support for setting up autonomous flights.

Figure 5.10: Parrot Mambo Minidrone [12]

5.4.1 Technical Details

- **Size**: 18x18 cm

- **Weight**: 63 g
- **Battery**: 660 mAh LiPo
- **IMU (Inertial Measurement Unit)**: 3 axis Gyroscope and Accelerometer are included.
- **Pressure Sensor**
- **Ultrasonic Sensor**
- **Camera**: 120x160 pixels
- **Operating System**: Linux

The motherboard is shown in Figure 5.11 and is equipped with an 800 MHz ARM A9.

Figure 5.11: Moderboard of the Parrot Mambo

Two models of this drone were used for this project before adding the next element to the fleet.

5.5 Hasakee Q9s

This drone (Figure 5.13) was born with the purpose of being a toy and was used as a manually driven leading drone to follow.

Figure 5.12: Hasakee Q9s drone and controller [13]

5.5.1 Technical Details

As this is a toy drone, most details are private.

- **Size**: 17.5W x 17.5L x 5H cm
- **Battery**: 500 mAh LiPo
- **Transmitter Frequency**: 2.4 GHz.

5.6 Miscellaneous Instrumentation

Some other elements were required to complete the project safely.

- **Black Base Cloth**: This element was needed to reliably create a black background of the track, as the camera wasn't able to distinguish colors (Since black paper is a bit lucid, it will reflect light and would be detected as white) (see 6.1.2.3).
- **Paper Sheets and Tape**: These were used to create the track needed to keep the drones in position (see 6.1.2.3).
- **Red Strip**: To help the optical flow, 2 red strips were placed parallelly to the track as outside boundaries (see 6.1.2.3).

- **Safety Glasses**: Needed to be protected from uncontrolled flights.
- **USB Bluetooth Dongle**: at first 2, Mambo mini-drones were used, and one PC couldn't be connected to both, and the second PC didn't have a Bluetooth element, so an external USB dongle needed to be added.

Figure 5.13: Safety glasses [14] and Bluetooth USB dongle [15]

5.7 Firmware Description

All needed tools to successfully complete and run this project are based on Matlab and are all listed below with their version.

- **Matlab** (Version R2019b): this is the tool where everything runs.
- **Simulink** (Version 10.0): this tool allows the user to build system schematics and run them using the Matlab software. It is a very powerful tool, especially in model-based designs for the simulation steps.
- **Navigation Toolbox** (Version 1.0)
- **Computer Vision Toolbox** (Version 9.1)
- **DSP System Toolbox** (Version 9.9)

- **Image Processing Toolbox** (Version 11.0)
- **Simulink Control Design** (Version 5.4)
- **Control System Toolbox** (Version 10.7)
- **Embedded Coder** (Version 7.3)
- **Simulink Coder** (Version 9.2)
- **MATLAB Coder** (Version 4.3)
- **Simulink 3D Animation** (Version 8.3)
- **Signal Processing Toolbox** (Version 8.3)
- **Optimization Toolbox** (Version 8.4)
- **Aerospace Toolbox** (Version 3.2)
- **Aerospace Blockset** (Version 4.2)
- **Simulink Support Package for Parrot Minidrones** (Version 19.2.5): This is the most important package for this project as it is managing the communication with the drone.

CHAPTER 6

SENSOR INTERFERENCES CORRECTION ALGORITHMS

The Parrot Mambo mini-drones were designed to be flown singularly as they use a sonar for altitude estimation.

6.1 Sonar Interference

Using a group of drones in close formation and at different altitudes will create an uncorrelated sonar array, making it impossible for every drone to distinguish the reflection signal of its sonar from the units close by (representation in Figure 6.1).

It is discouraged to mount a different sensor on the drone (a more directional sonar or a laser pointer, for instance), as its basic hardware and software are private, and the specific model is also obsolete, making it even more difficult to modify.

6.1.1 Sonar Synchronization method

Being able to synchronize the sonar emission signal, and since all drones are flying at different altitudes, it is possible to distinguish between the same sonar reflection and the other unwanted reflections. The algorithm would define a valid time window where the reflection is expected to be sensed. This interval would be centered on the last measured altitude reflection time and have a margin defined as the maximum altitude variation per sampling period with uncertainties.

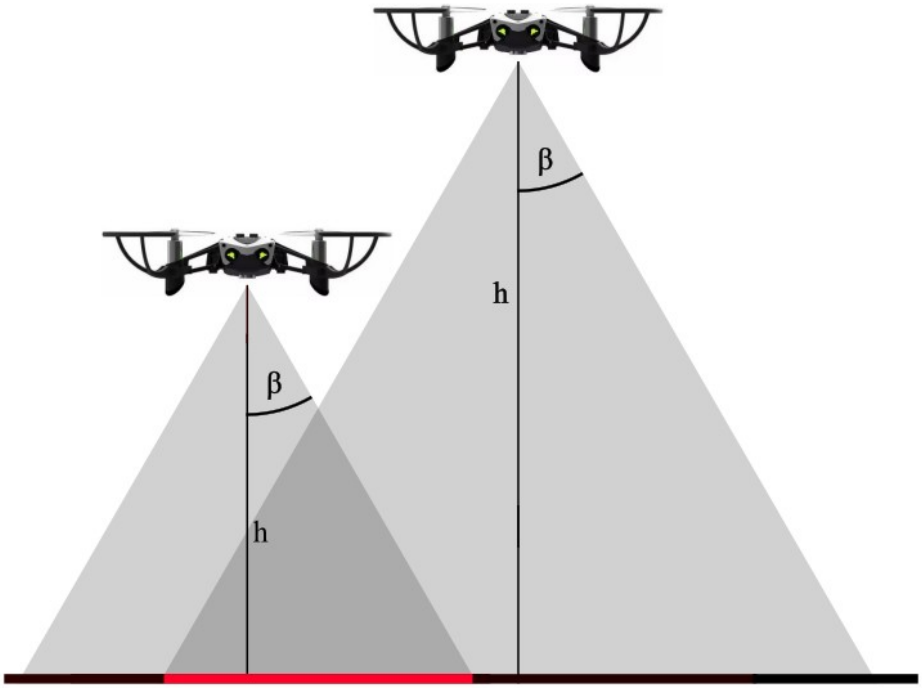

Figure 6.1: Representation of sonar interference. The red section of the ground is where all interferences are created

Every reflection signal falling out of the valid time window will be discarded as it must be caused by other vehicles around.

Although sonar synchronization is the easiest solution to the problem, it will need some interaction between drones, and the goal of this research is not to use direct communication. It might be possible to start all drones and sonars at the same time with the same sampling period and avoid derivation via implicit communication, but however, this method wasn't further explored as it might be affected by too high uncertainties. Furthermore, if drones were flying at similar altitudes (even if this is not the specific case), there could be more signals falling in the same time window.

6.1.2 Sonar Replacement method

Using the available sensors and signals in the drone, it was found to be possible to design a new altitude estimator. The instant sum of rotor commands' power, the estimated pitch and roll, and the down-facing camera are enough to design an alternative altitude estimator that can take over on the sonar when needed. A more elaborate solution completely replaces the sonar signal with a designed alternative estimator when any interference is detected. The sonar output will be replaced when the other estimator is reading an altitude higher than the sonar plus a threshold as visible in the "Estimation_selector" block in Figure 6.2.

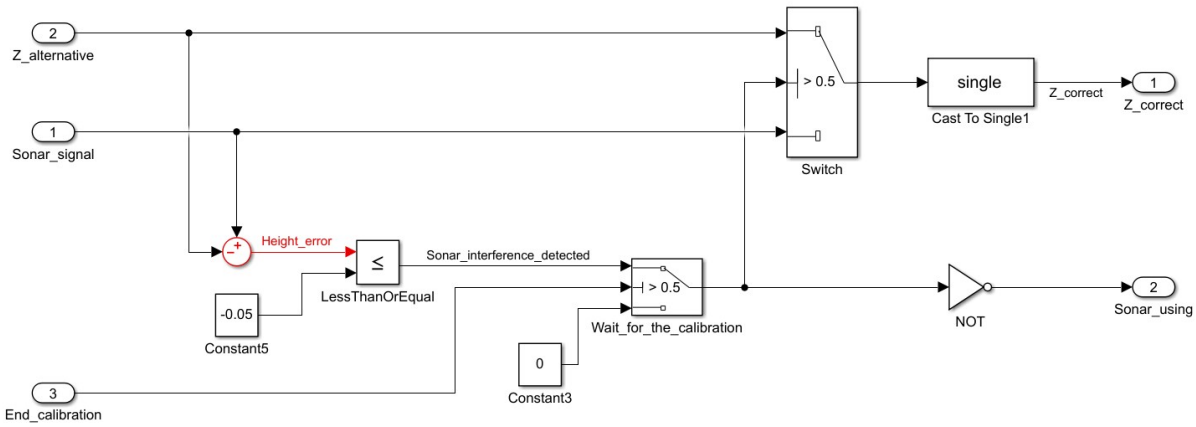

Figure 6.2: Estimation selector block to choose correct altitude.

As it is not suggested to modify the hardware, the new estimation algorithm was based on combining the already existing camera and the control output power commands to the rotors.

6.1.2.1 Down-Facing Camera data Correlation with Altitude

It is well known that any object seen by a camera is detected as smaller as its distance from the objective increases.

Measuring ground objects' size in camera pixels thus results in a good starting point for estimating the altitude.

In this chapter, all lines parallel to the drone's x-axis will be referred to as vertical, and all lines parallel to the y-axis will be referred to as horizontal since this is the way they appear on the camera image.

Calling "p(t)" the obtained result from a ground object size in pixels at the sample "t" the relationship with the altitude at that time "Z(t)" is:

$$Z[t] = \frac{1}{p(t)} \cdot \frac{Z_{ref}}{\frac{1}{p_{ref}}} \tag{6.1}$$

Where the 2 constants p_{ref} and Z_{ref} are respectively the dimension in pixels seen at the reference altitude and the reference altitude, both defined at calibration time (6.1.2.2).

Assuming the camera is always parallel to the ground while changing altitude, sizes in all directions will change following the same ratio, meaning that it is sufficient to measure only horizontal (with respect to the drone) distances between shapes' borders.

This approximation can be done as the drone always flies at low yaw and pitch angles. Cases with more significant inclinations happen in a short enough time period that doesn't significantly affect the measure.

In this project, the measured size comes from an average measured in each image line between the vertical margins of every object (see 6.1.2.3).

In synthesis, this method is able to give a good absolute measure. However, the main drawback stands in the low data rate of the image processing system, which can be, at best, 20 times slower than the control system's sampling frequency (fixed at 200 Hz) if all data is valid. In the real world, the data rate will be affected by the computational cost of each frame's image processing, while in simulation, that frequency is fixed at 5 Hz. Furthermore, camera noise and yaw oscillation will invalidate a considerable amount of samples, increasing the process complexity.

If the vertical speed is low or there are no visible shapes on the ground, the corresponding samples will be canceled, reducing the already low data rate. Using only this altitude estimation will produce unwanted oscillations as the drone's position will derivate between 2 samples.

6.1.2.2 Camera Altitude Calibration

For a more reliable "ad-hoc" measure of $\frac{Z_{ref}}{\frac{1}{p_{ref}}}$, a dedicated calibration can be set up: the drone is kept at a fixed altitude over the path, and the program is run with power output 0%.

Once obtained and loaded into the Matlab workspace, the average image distance from the ".mat" file, run the script visible in A.3.1.

In this specific case, two tables were used to keep the drone at a fixed altitude, and such altitude was measured manually and used in calculations (see Figure 6.3).

Figure 6.3: How the camera calibration measure was set up

6.1.2.3 Ground Path Description

Another major flaw of the image altitude estimation algorithm stands in the need to have all vertical lines at the same horizontal distance. A path following that requirement will be formed by lines all parallel to the X-axis. Position estimation is based on ground objects. Thus, a ground composed of only parallel lines will result in the impossibility of estimating the X position. At the same time avoiding this condition will generate unwanted effects on the altitude estimation, depending on the drone's movement, but the position estimation is more important:

In case a new object with a different horizontal width is encountered while moving forward, there will be a ramp effect on the signal "P(t)" that indicates the average horizontal ground shapes' size. In fact, these different horizontal samples will gradually occupy more image lines in every time frame, making the average slowly change while staying at the same altitude.

Moving sideways, if a new object at a different horizontal distance from the main track is encountered, a sudden step will happen on the signal "P(t)". A number of samples equal to the vertical width in pixel of the new object will be suddenly added to the average calculation, making it quickly change.

The same effects would happen if the drone was measuring vertical distances but with the movements inverted.

Using both vertical and horizontal estimations would let the drone able to detect new vertical or horizontal objects by looking respectively at steps on the vertical or horizontal measures and consequently invalidate the ramps happening on the other estimator. However, this method happens to not work on diagonal movements as both measures will experience undetected ramps and invalidate the whole estimator.

Since mapping the flight area is time and memory-consuming and all flights took place indoors in a controlled area, it was possible to design a ground track with some "ad hoc" features. For long-range outdoor drones, the solution might be different as the pressure measure is available together with other sensors. In the end, the final solution was to make ground objects all with the same horizontal width and at the same horizontal distances, using the shapes described in Figure 6.4 This solved the horizontal movement step problem.

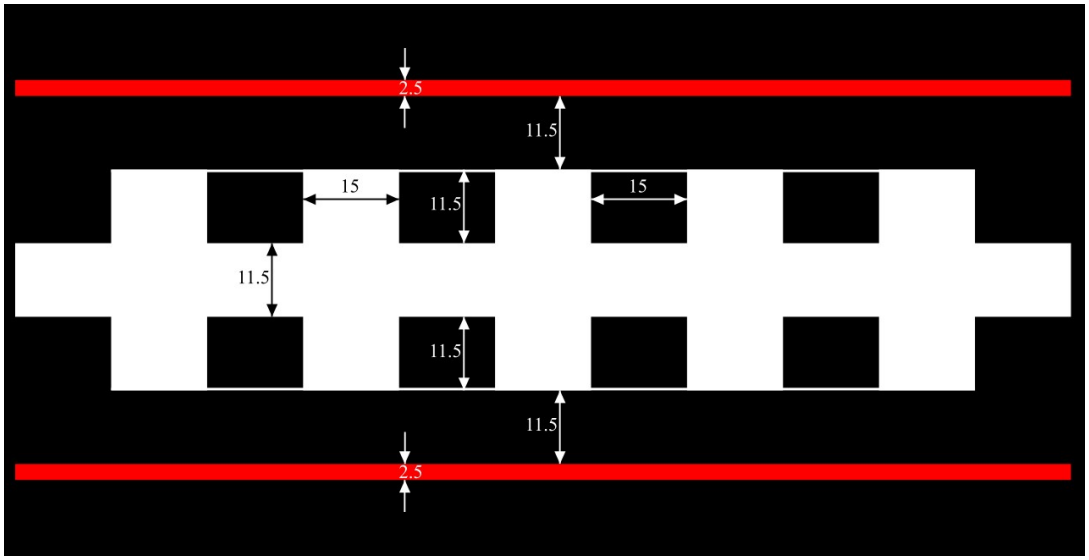

Figure 6.4: Used Track with quotes in centimeters (colors are the same visible in reality)

For the vertical movement ramp problem, all visible vertical lines are enhanced from the portion occupied to the whole image matrix (see 6.1.2.4), and since all objects are aligned thanks to the previous condition, new encounters will only be useful to keep the line from disappearing. This idea was implemented based on the specific case, but it could also be used in other cases by exploiting some environmental points of reference.

In Figure 6.5, it is possible to see the real world and the virtual version of the track. Notice that in the virtual world I tried to use colors more similar to what the drone sees.

6.1.2.4 Image Processing System for altitude estimation (First solution)

Everything described in the previous chapter is computed in the image-processing system block Figure 6.6. First, camera data is converted into the RGB format using the "PARROT Image Conversion" block, then using a color filter (Black Binarization in Figure 6.6), the image is converted into a binary matrix having 1 where there is something that is not black cloth. In this case, there are 3 filters: the first is filtering white-only shapes and it is used to center the

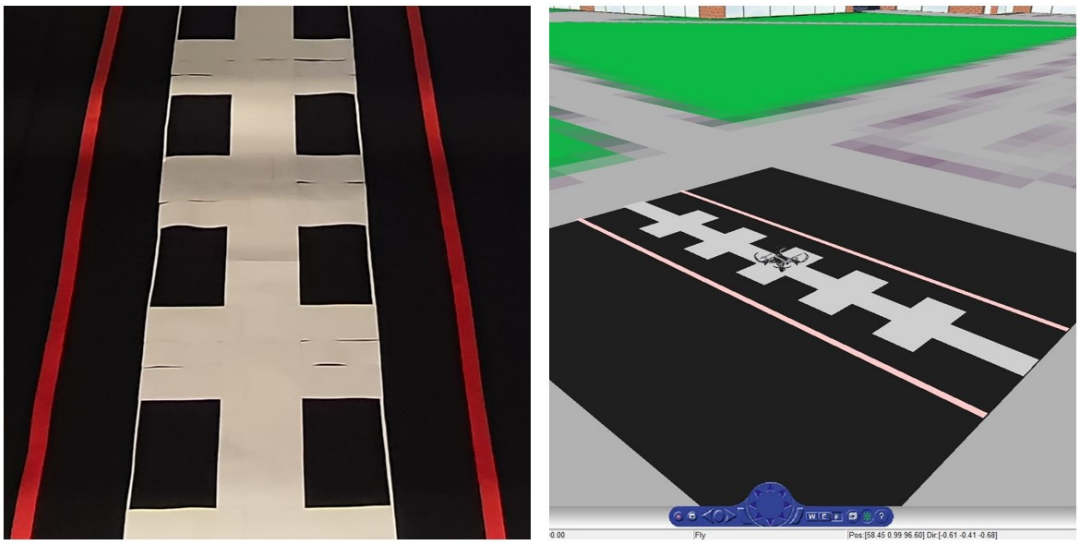

Figure 6.5: Real track on the left and virtual track on the right (with colors more similar to what the drone sees)

track, while the second and third are filtering out all that is not black. Between the real world and simulation, there are differences in the black cloth detected color.

Following the black binarization, another filter is applied to reduce noise (same for tracking and altitude measuring), (see "Image_Filter" and "Noise_Filter" in Figure 6.6). Until this point, the process is similar to the line tracking algorithm used to control the Y coordinate. The tracking section is an optimization of previous work [20] and will be described in chapter 6.1.2.5 and 6.1.2.6.

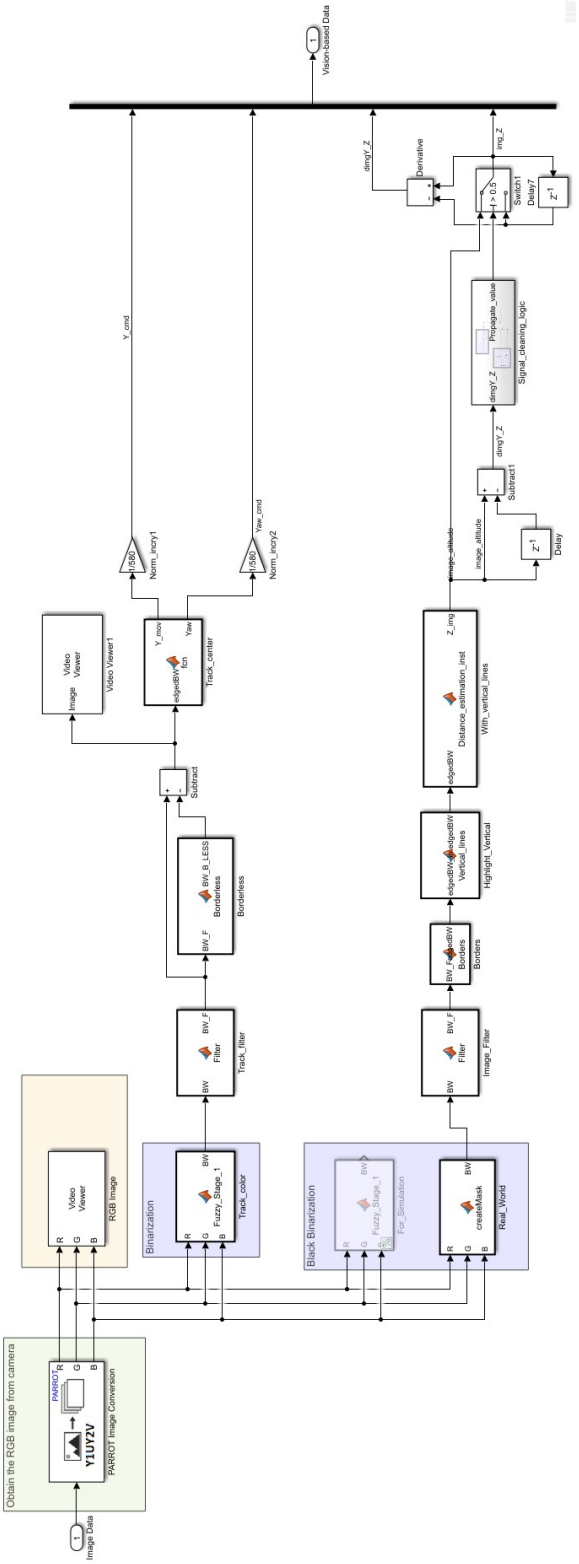

Figure 6.6: Full Optical Flow structure for Real World application

Next, using the **prewitt** method (in the "Borders" function), ground shape borders are found, and since the points of interest are only vertical lines, and the track is not made only by such lines (see 6.1.2.3), every matrix's column is completely set to 1 (white) or 0 (black) based on the number of pixels set to 1 contained ("Vertical_Lines" function). A higher number of ones indicates a higher chance of having a vertical line there. A problem with this function stands in the yaw of the drone, which might be different from zero (see 6.1.2.6). Once the vertical lines-only image is obtained, the "Distance_estimation_inst" function calculates the average horizontal distance between white pixels for each line and converts it to a not normalized altitude estimation using the Equation 6.1 without Z_{ref} and p_{ref} (they are multiplied later), and sets to 0 all those cases with insufficient or no data (It filters out also too small distances (strip width) and too wide distances).

Now, the output signal still has some noise, also caused by the no data periods that set everything at zero, which will cause too big steps when the drone is climbing or small unwanted steps when hovering.

Canceling those steps can be done by working on the signal derivative (calculated by subtracting from every sample its previous) by combining 2 conditions (Figure 6.8):

1. Clear the derivative if it is smaller than the low noise threshold: this lower limit was verified experimentally via data analysis and a certification circuit.

2. Clear the derivative if greater than the higher noise threshold: together with all jumps to and from zero, there are also some more glitches that can cause random peaks. All

these invalid data are treated such that in the derivative, they are cleared, while in the absolute measure, they are overwritten with the previous value.

As a result of this operation, the derivative with cleared data won't correspond exactly to the derivative of the final signal but will contain some permanent biases (It is thus calculated again from the final signal). Imagining a peak signal, it is very unlikely that values around it will be exactly the same. Such a difference would be lost when clearing the peaks.

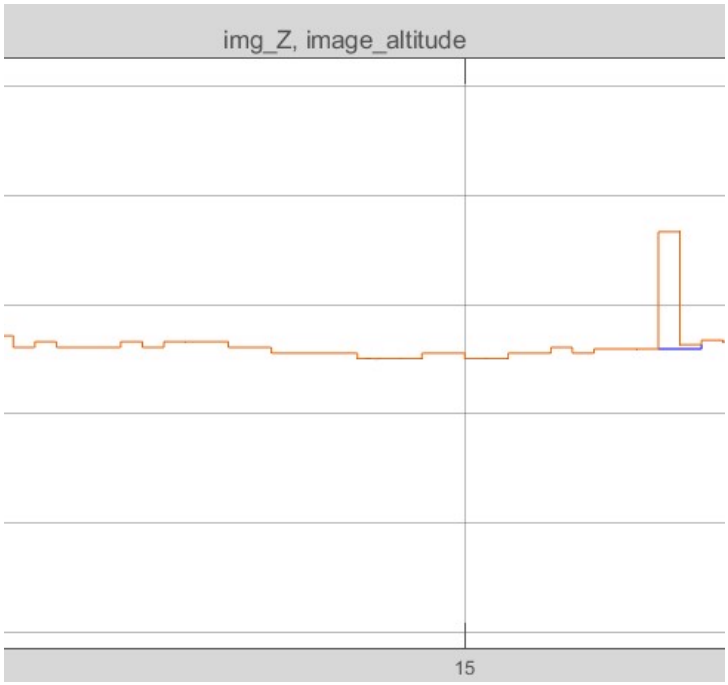

Figure 6.7: Example of a peak cleared by the peak clearance function

Finally, the improved versions of the signal and its derivative are sent to the "Alternative Altitude Estimator" to validate the results.

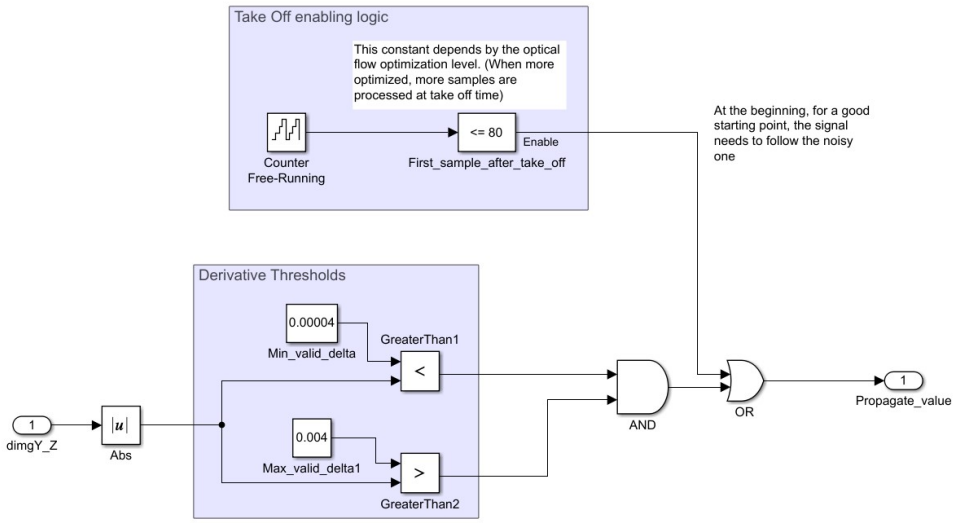

Figure 6.8: The absolute value of the derivative is valid when between 2 thresholds. While at take off, the propagation is forced to avoid clearance of the first step

Before, a different algorithm was used, and its idea was the following: at every cleared step, save it and sum it to all the future ones. Their sum would be needed because a peak is not symmetric, and if both sides were just cleared, cumulative biases would be introduced to the signal. However, this idea has a major flaw: if, in a peak, one side has a step higher than the threshold and the other lower, only one would be cleared and summed to other peaks. As a result, such a peak would result in a step of the final signal. This previous algorithm worked very well only when noise steps are all well detectable. The newer solution is worse but it was ideated such that no biases are introduced in all cases.

6.1.2.5 Image Processing System for altitude estimation (Optimized)

When testing in the real world, some problems were noticed. At first, the red trip appeared as a group of white dots, and it was also noticed that the sample rate was too low, and optimization was needed.

To solve this first problem, the "Borderless" function that was used to detect the landing circle in previous work[20], was used to find all those dots and subtract them from the image. This function is giving as output an image of only shapes that don't touch the image's border. By making a matrix subtraction to the input image, only the track (that is always touching the image border) is kept.

Secondly, it was noticed that despite the simulation always having a rate of 5 Hz, in the real world, this version had a data rate of around 0.2/0.3 Hz, meaning a sample every 3-5 seconds. Adding the not valid samples in the mix and the effective data rate would have been, on average, a sample every 5 seconds: too low for what was needed.

After that finding, 2 main optimization steps were applied.

1. **Matrixes become lines**: as this section is looking only at the Y direction and at the Yaw, it is possible to reduce matrixes to horizontal vectors.

   This means that the whole original track detection section [20] had to be replaced by a simpler version (see 6.1.2.6 and Figure 6.6).

   About the altitude measuring, it was possible to combine the effects of "Borders" and "Highlight" blocks (Figure 6.6) into the "Vertical_Positions" block (Figure 6.9) that would collapse the image into a horizontal vector by summing all column elements, thus counting white pixels in every column. As the track is made (see 6.1.2.3), the resulting vector will have some clearly distinguished regions based on the white pixels count. Defining the correct number of white pixels difference between columns, it will then be possible to point out the borders of these regions where vertical lines are.

Because of these actions, also the "With_Vertical_Lines" function (Figure 6.6) had to be slightly modified and optimized to become "Without_Vertical_Lines" (Figure 6.9). The main difference between these two is that the former was looping on all 120 lines of the image, while the latter is now performing a single iteration as there is only one line, and calculations are sensibly reduced.

2. **Remove filters**: the previous step arrived at a point where results were around the same as before but with a higher data rate (around 3-4 Hz). Now, since the data rate is still not good enough, image filters were removed. Those blocks were performing a matrix convolution with a 3x3 box to filter out noise: that operation is very expensive, and the drone board is not powerful enough.

   Furthermore, as the black binarization step is expensive, it was decided to connect the path planning section to the same block as the rest. This meant that red strips were completely detected, and the "Borderless" function wasn't useful anymore. In the end, data were observed to be similar to previous results. The only different thing was the data rate, which now arrived at a good frequency (around 7 Hz).

Data rate in the real world doesn't have a constant frequency, but it will continuously change a little: a possible explanation stands in the fact that the optical flow hardware is executing in parallel both position estimation and "Image Processing System" logic.

The final optimized Image Processing System is visible in Figure 6.9 See all Matlab codes used in the blocks in A.1.

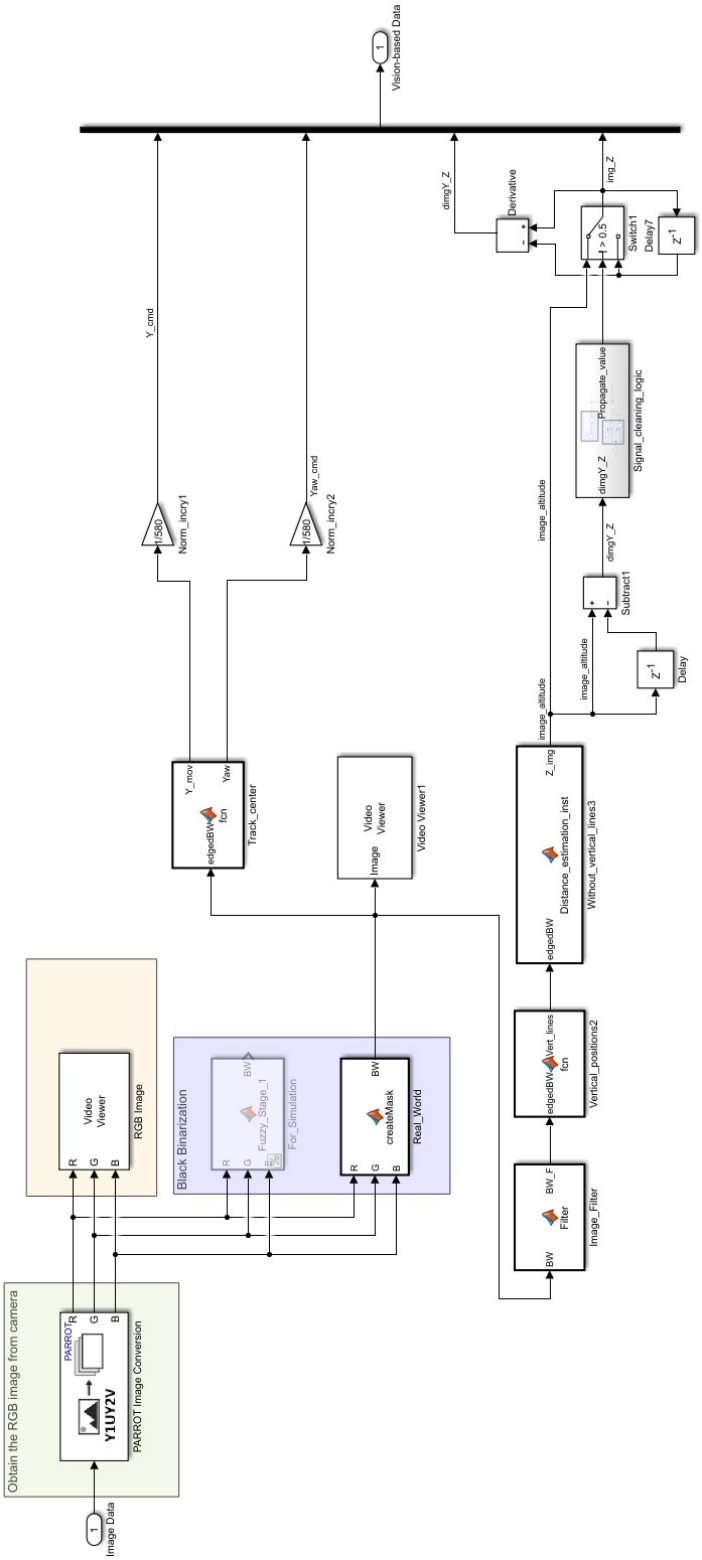

Figure 6.9: Full Optimized Optical Flow structure for Real World application

6.1.2.6 Yaw Control Algorithm and New Track Detection

While optimizing the track detection section, it was also decided to insert also a yaw detection. The reason behind this is that when a rotation happens, the vertical lines detected won't be parallel anymore to the matrix's columns. At first, they are detected in a wider number of columns, but then (due to the now too-high differential threshold) they are just canceled.

This effect will reduce the measured ground distances while maintaining the same altitude (resulting in a ramp effect in Z estimation), and when the yaw is too high, all estimation becomes random.

Controlling the Yaw angle will reduce this problem.

The idea applied in the "Track_Center" block (code in A.2.1) is to divide the image into an upper half (first 60 lines) and a lower part (remaining 60 lines). Both sections, for optimization purposes, are immediately collapsed into line vectors.

Later, both vectors are divided in half, and through a threshold, the program counts how many elements represent a sufficient number of white pixels in their left and right sections.

Once obtained these counts it is possible to measure the distance with the center of the track and the yaw using Equation 6.2. At first, the difference of white points between left and right for both vectors are found (Equation 6.2a and Equation 6.2b), then Y and Yaw are found as their average and their difference (Equation 6.2c and Equation 6.2d).

$$dY_{top} = WhiteCount_{top}^{right} - WhiteCount_{top}^{left} \tag{6.2a}$$

$$dY_{bot} = WhiteCount_{bot}^{right} - WhiteCount_{bot}^{left} \tag{6.2b}$$

$$Y_{dist} = \frac{dY_{top} + dY_{bot}}{2} \tag{6.2c}$$

$$Yaw = dY_{top} - dY_{bot} \tag{6.2d}$$

6.1.2.7 Rotors' Power Command Output data Correlation with Altitude

In the original Parrot Mini-drone Competition, all the commands are translated into power gains for the four rotors. These gains are calculated by the controller block and correspond to the system output called "motorCmds", a vector of four elements.

As said by J. Gordon Leishman [28], the ideal relationship between the vertical thrust (T) of a blade and its power consumption (P) is in Equation 6.3, where v_i indicates the induced velocity in the rotor plane.

$$T = P \cdot v_i \tag{6.3}$$

This equation is ideal (it does not consider other factors, such as pressure changes, for instance), but it is a representation good enough for the used flying environment (a small closed room).

Since the power gain (P_G) of a propeller is directly proportional to the vertical force, vertical acceleration (a) can be easily obtained. The induced rotor speed is considered to be constant for this application.

$$F(t) = P_G(t) \cdot v_i(t) \qquad v_i(t) \approx v_i \tag{6.4a}$$

$$m \cdot a(t) \approx P_G(t) \cdot v_i \tag{6.4b}$$

$$a(t) \approx P_G(t) \cdot \frac{v_i}{m} = P_G(t) \cdot \beta \tag{6.4c}$$

As visible in the calculation, all the physical characteristics not of particular interest for this application were collapsed into the β factor and will be calculated at the calibration time together with all other constants.

During a regular flight, there will be some periods when the drone is flying with a small angle, meaning that for a higher precision on the vertical force, estimated pitch and roll need to be used. To find the purely vertical force, it is necessary to use the trigonometry theory for defining a rotation matrix 3.2.1.

Using the rotation definitions in Equation 3.3, I defined the rotation on the pitch (Y-axis) and the one on the roll (X-axis), transforming the vertical vector with respect to the drone system to the vertical vector with respect to the world system. The obtained result from these calculations is:

$$F(t) = (P_{G1}(t) + P_{G2}(t) + P_{G3}(t) + P_{G4}(t)) \cdot \beta \tag{6.5a}$$

$$F_z(t) = cos(\phi(t)) \cdot cos(\rho(t)) \cdot F(t) \tag{6.5b}$$

Once this formula is applied to the signal (In the system it was done in the "Pure_Vertical_Thrust" block see Figure 6.10), the vertical position can be calculated using the double integral definition:

$$Z(t) = \int_0^t \left(\int_0^t (a_Z(t) - bias_1)dt - bias_2 + C_1 \right) dt + C_2 \tag{6.6}$$

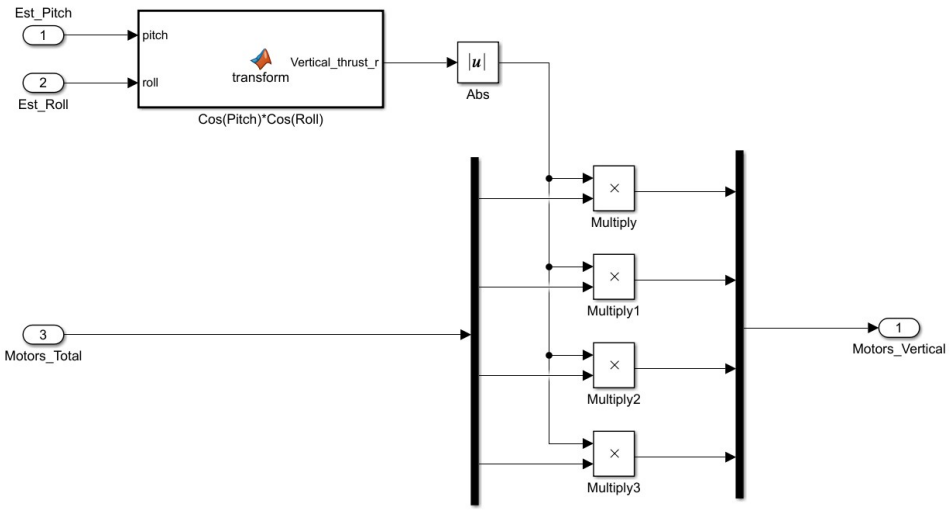

Figure 6.10: Here, the rotational transformation depending on pitch and roll is applied to each rotor signal (the Matlab function block computes the cosines' product (Equation 6.5b))

But as the system time is discrete:

$$Z(t) = \sum_{0}^{t} \left(\sum_{0}^{t} (a_Z(t) - bias_1) - bias_2 + C_1 \right) + C_2 \tag{6.7}$$

Constants "C_1" and "C_2" will be arbitrarily defined with values respectively 0 and "Z_{ref}", while the 2 biases were introduced to avoid divergence. These signals would probably not have a 0 average because they are discrete, and this aspect will cause integrative divergence.

All these transformations are valid if starting from zero, but since in the following algorithm, all calculations use the flying altitude as a reference, there will be the need to subtract the reference power gain from the final obtained vertical force. Thus: $P_G(t) = P_{G\_tot}(t) - P_{G\_ref}$

On the contrary of the image altitude estimation, this method works at the same data rate of the controller (200 Hz), meaning that it is able to reduce the blind derivation between two samples, but it presents two big drawbacks:

1. **Derivation**: even if the two integrative constants were precisely estimated at calibration time, there would still be a parabolic derivation in the position caused by measurement errors and system characteristics derivation. A periodic correction of these constants' estimation becomes necessary for long flights.

2. **Jerk**: this cinematic aspect is defined as the acceleration derivative (or position triple derivative) in kinematics and has a spiking behavior every time the movement shape changes, as the acceleration is usually shaped as steps in the flight commands. A simple example would be when the drone is accelerated for moving upward: acceleration command instantly changes from "$g = 9.81m/s$ to "$a \neq g$", meaning that rotors will experience a spiking power request, making the estimated altitude a lot higher than reality. Jerk spikes won't happen only if the position changes as a perfect hyperbolic cosine function since its triple derivative in 0 is actually 0. A discrete-time system makes verifying this condition impossible, leaving the only option of muting the thrust altitude estimation for a short amount of time when a high jerk is requested.

   For this application, only vertical jerk effects are muted (Figure 6.11) as the horizontal is automatically cleared when summing the rotors (Figure 6.12): when accelerating horizontally the 2 rotors pointing in the direction of travel will have a decreasing peak while the opposite 2 will experience an opposite behavior such that the sum won't present that problem

By looking at what is discussed above, it becomes clear that by combining the image and thrust altitude estimators, it is possible to cancel most of their drawbacks. The image will be

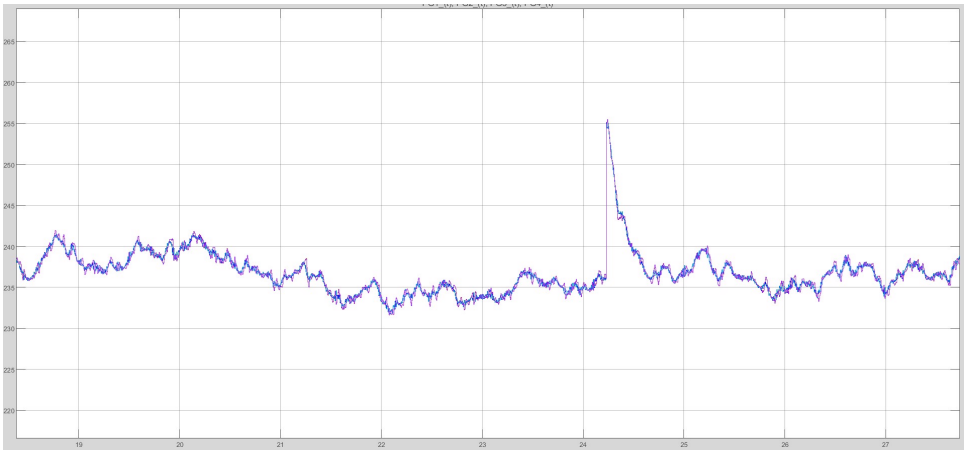

Figure 6.11: Example of vertical jerk effect on the 4 rotors after the sin calibration (after second 24)

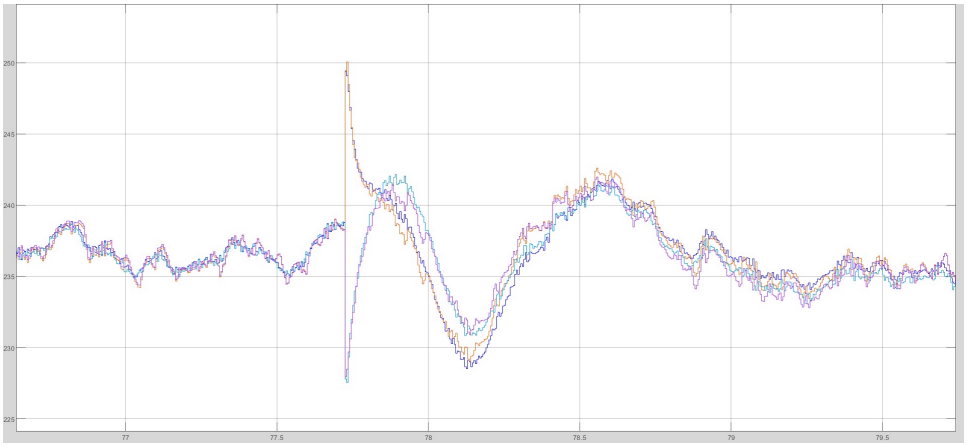

Figure 6.12: Example of horizontal jerk effects on the 4 rotors when a movement is started (after 77.5 seconds)

used for the running correction of integrating constants and as a substitution in case of a high jerk happening, while thrust will be used for a continuous estimation (6.1.2.8).

6.1.2.8 Final Altitude Estimation System

Now, knowing all the mathematical relationships, the algorithm can first be divided into two main sections: calibration and running corrections. It is assumed that the sonar is correctly working during the first main section.

Performing a separate calibration and recording the results can't be done as initial engine constants are correlated with the battery status (tests were performed at different battery levels and with multiple batteries). The whole flow is managed by a Finite State Machine using one-hot encoding as output (Figure 6.13). The main body can be found in the "Alternative_Altitude_estimator" block (Figure 6.14), located in the "State Estimator" (Figure 8.14), where the sonar signal is taken as input.

Every step of the Finite State Machine is managed by one of the visible subsystem blocks: blocks in magenta represent steps executed one time, thus belonging to the calibration section, while dark green subsystems are executed periodically for running corrections.

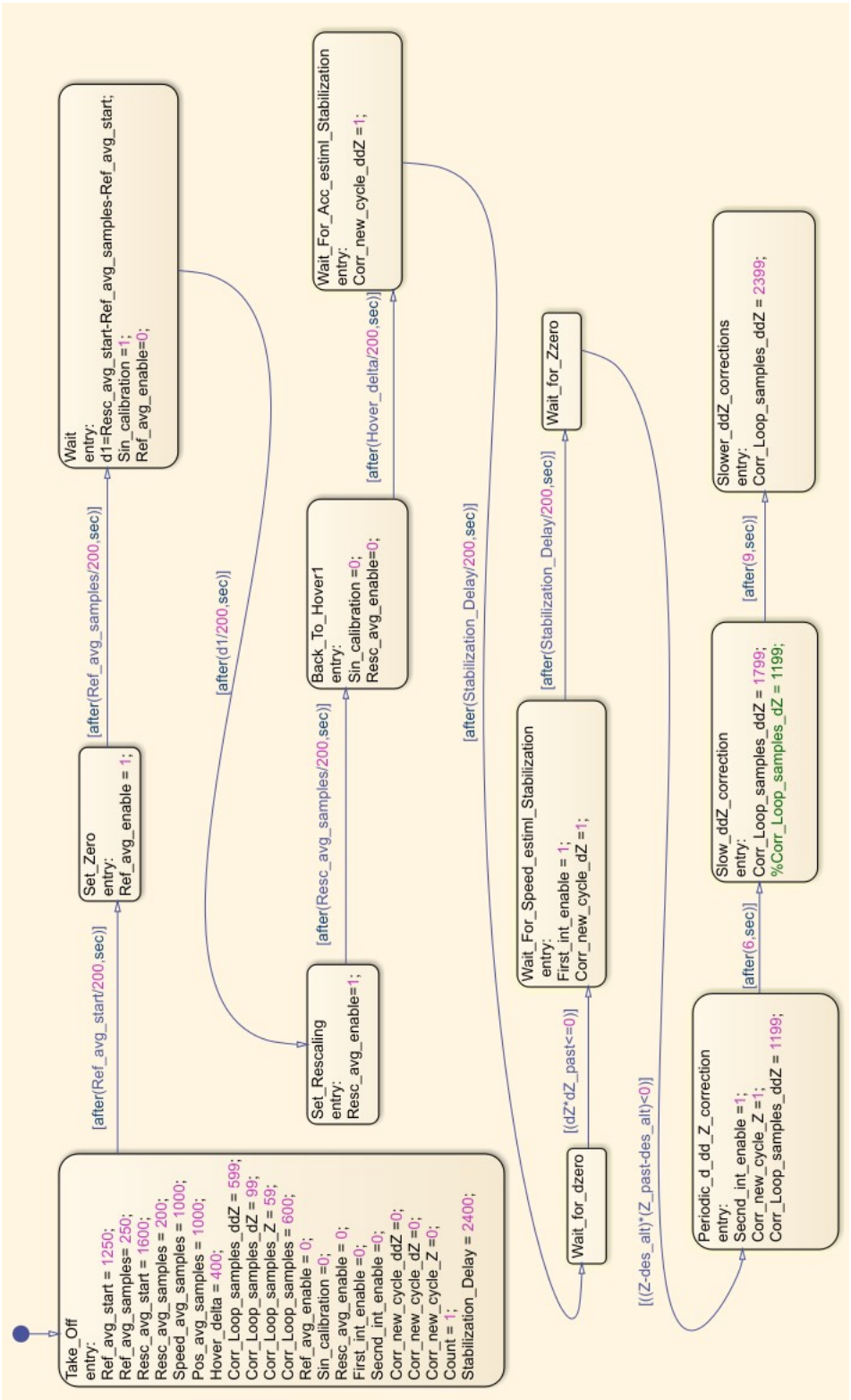

Figure 6.13: Full used FSM in the Z altitude alternative estimation

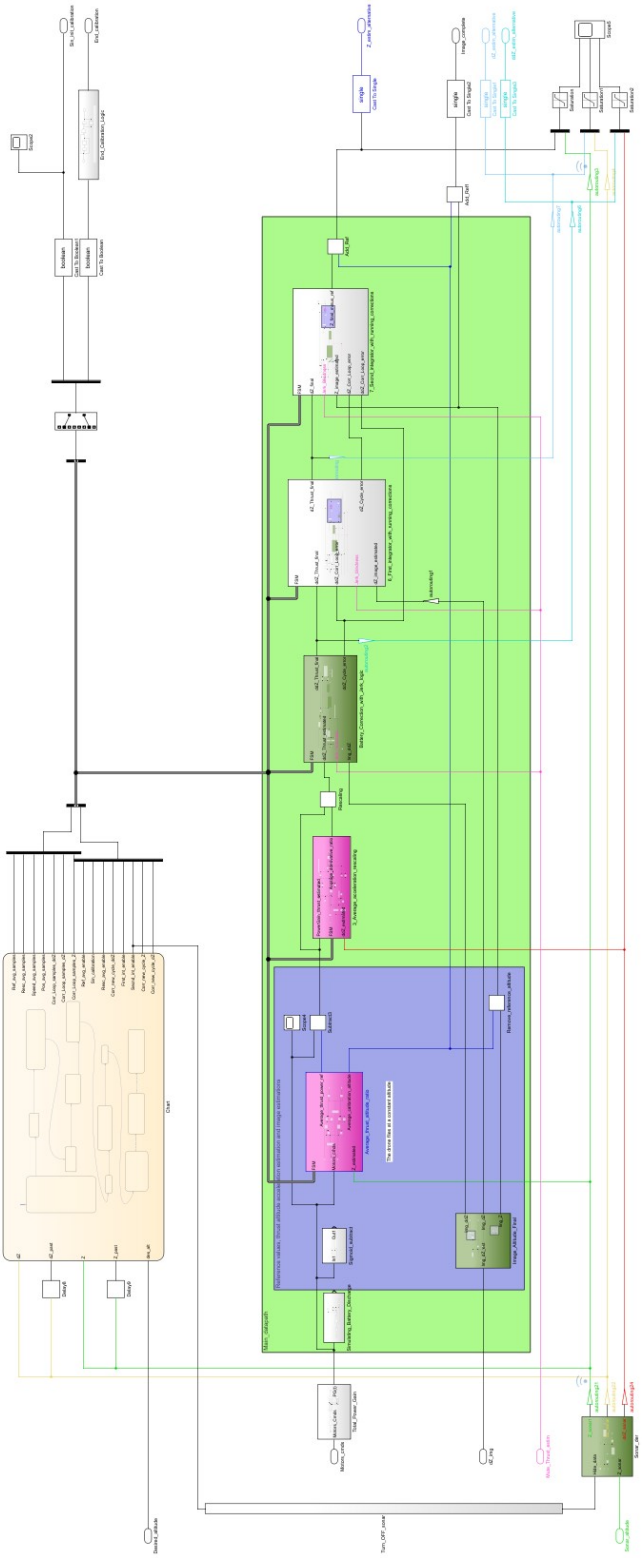

Figure 6.14: Full Alternative Altitude Estimator Logic

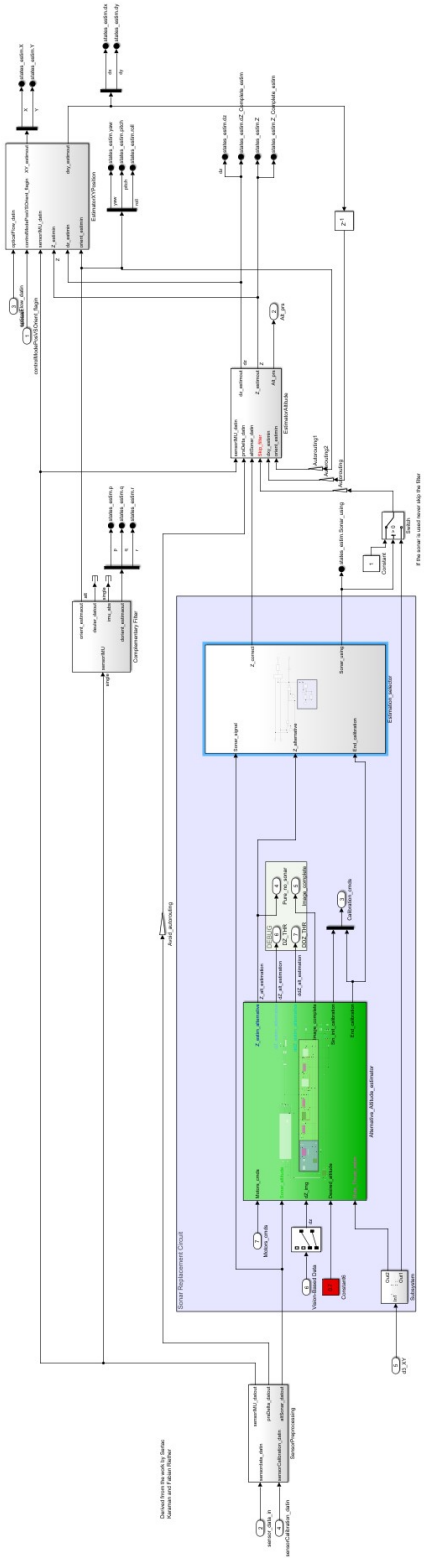

Figure 6.15: Full modified states estimator

Finally the algorithm flow steps can be listed as follows.

1. **Take Off**: as a very first step, the drone needs some time to stabilize correctly at the desired altitude. Here, the Finite state machine takes care of initializing all its one hot encoding outputs to 0 and all the calibration interval lengths to their predefined values. (See "Take_Off" state in Figure 6.13).

2. **Set Zero**: since all calculations will be done starting with the drone already at the reference altitude, the first thing to do is to estimate with an averaging circuit (Average_thrust_altitude_ratio block in Figure 6.17) the reference total power gain (Figure 6.16) of the rotors "P_{G_ref}" and the average measured altitude.

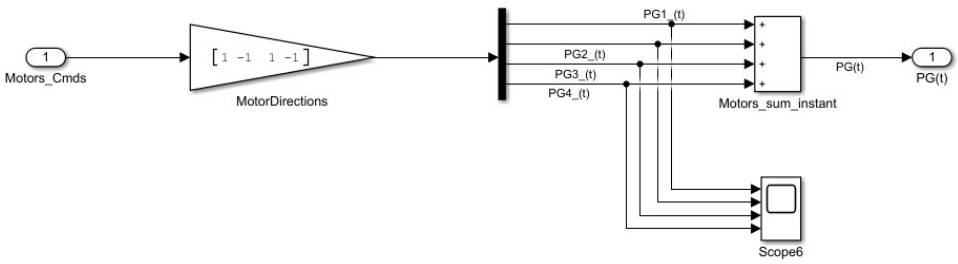

Figure 6.16: Before the averaging block, the total Power sum must be computed.

In parallel to the averaging block, the image altitude formula is completed in the "Image_Altitude_Final" block (Figure 6.18) by multiplying the previously measured Z_{ref} and p_{ref}, both for the derivative and the absolute image altitude estimations (also image altitude acceleration is calculated at this point). Derivatives are calculated without dividing by the sampling time because everything is rescaled in the vertical acceleration, and all

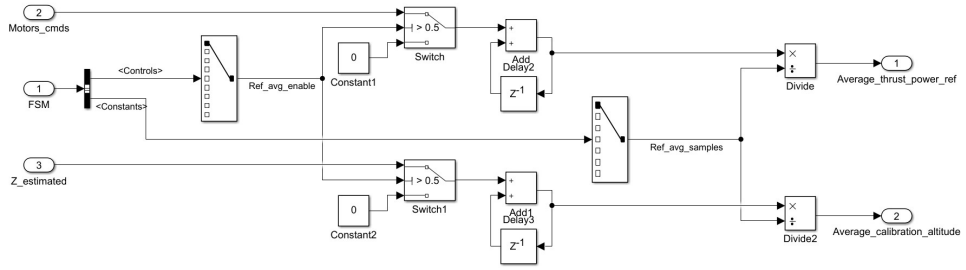

Figure 6.17: Averaging circuits that are activated and receive the number of samples by and from the FSM (Figure 6.13)

integrals are computed by not multiplying for the sampling time (thus, it is a mathematical simplification). Furthermore, since all next calculations will be executed using the flying altitude as a reference, the absolute Z image estimation needs to be subtracted from the reference altitude (Figure 6.19). For the rotors' power gain, the found average value is needed to keep the drone at the reference altitude, meaning that for looking at acceleration changes, it needs to be subtracted.

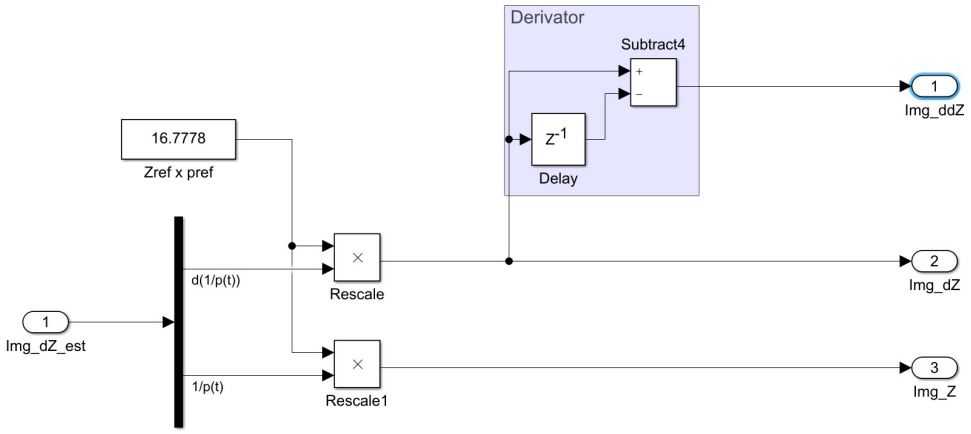

Figure 6.18: In this block, the Z, Z', and Z" values are obtained

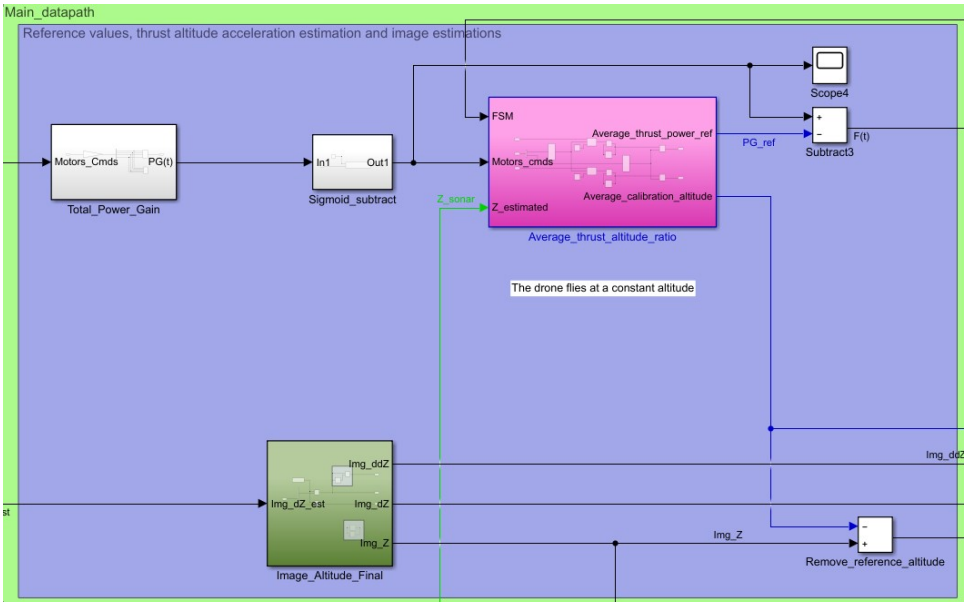

Figure 6.19: All blocks activated in the "Set Zero" state

At the end of this step, all image signals become available, together with all the reference altitude constants.

3. **Start Sinusoidal Calibration**: after taking the reference measures, a vertical sinusoidal movement is started to enhance the vertical thrust acceleration signal, such that the signal-to-noise ratio is increased. This change of movement will cause a Jerk peak. For this reason, a waiting state is set, with a calculated period of "(start time of the next state) - (start time of the previous state + period of the previous state) (see Figure 6.13).

   In this step some commands are sent to the path planning logic for correctly executing the vertical movement.

   At the end of this step, no new final signals are found.

4. **Thrust Signal Rescaling**: at this point the "Average_speed_derivative_rescaling" block (see Figure 6.20) is activated in the same way as the previous averaging block. It calculates an average of the thrust signal and an average of the sonar double derivative signal (Continuously calculated using 2 consecutive derivation blocks in "Sonar_Der" logic (Figure 6.21)) and the final ratio between them. Referring to the Equation 6.4c, the result obtained here is beta, and it is multiplied by the rotors thrust externally (see Figure 6.14).

   For a more consistent output, a final switch is activated at average completion by detecting the falling edge of the activation signal. The "OR + delay" blocks work as an anti-fuse and will keep this block's output active until program completion.

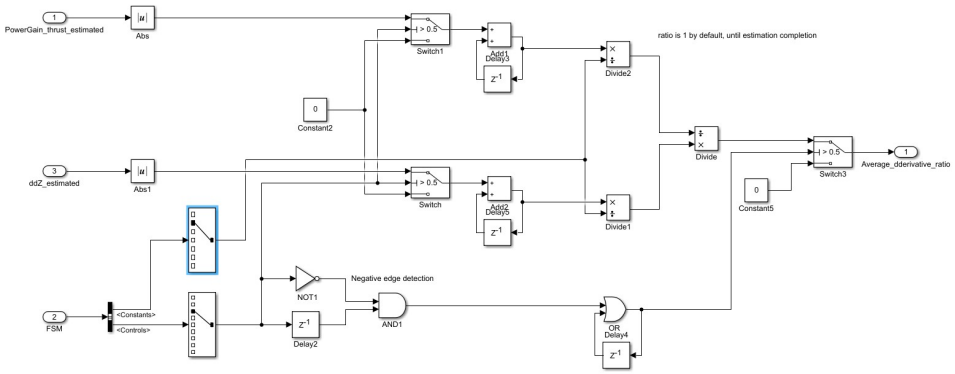

Figure 6.20: Acceleration rescaling block

At the end of this step, the vertical acceleration signal is found (It will contain a variable bias).

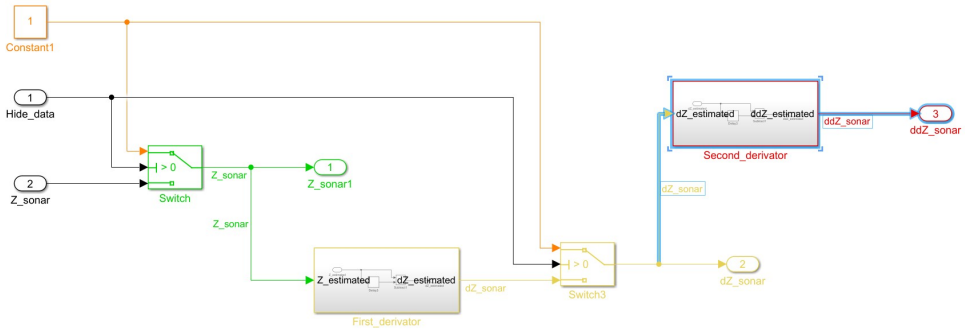

Figure 6.21: Sonar signal derivations. This block receives a signal that invalidates all sonar data after some time to demonstrate that everything is working.

5. **Back to Hover**: Now that the sinusoidal calibration, as it is not needed anymore, is stopped, a new jerk peak needs to be skipped. This step is again waiting for drone stabilization (see Figure 6.13).

   At the end of this step, no new informations are obtained.

6. **Wait for zero speed**: now the system is ready to start the first integration, needed to find the vertical speed. This operation can be done only when the measured sonar speed is at zero, as it is the same initialized value in the integrator. Avoiding to respect this condition will result in a biased speed estimation, as the zero would be set in a random position. Since all signals are discrete, the only way to detect a zero crossing is to verify that 2 consecutive samples have different signs ($dZ(t) \cdot dZ(t-1) \leq 0$). For a better result, it was decided that some additional time must be waited to ensure the acceleration correct stabilization; in fact, in this step, the running corrections of the acceleration are already activated (see 6.1.2.9).

At the end of this step, nothing happened, but at the beginning of the next step, the first integration is active.

7. **Wait for zero position**: now the first integration is active, and for the same reason as before, it is needed to wait some time for the signal to stabilize and a zero value to avoid biases.

   At the end of this step, the vertical speed signal is correctly stabilized, and at the beginning of the next one, the position integral is enabled.

8. **End Calibration**: When all signals are considered to be stable, the "end calibration" flag is set, indicating to the path planning logic that all movements are now enabled.

6.1.2.9 Running Corrections

As the Z values present variable random biases that might cause integration divergences, some continuous error estimations and corrections must be periodically run. There is one for each signal:

1. **Acceleration**: in the "Battery_Correction_with_Jerk_logic" block (Figure 6.22), all running corrections of the acceleration take place.

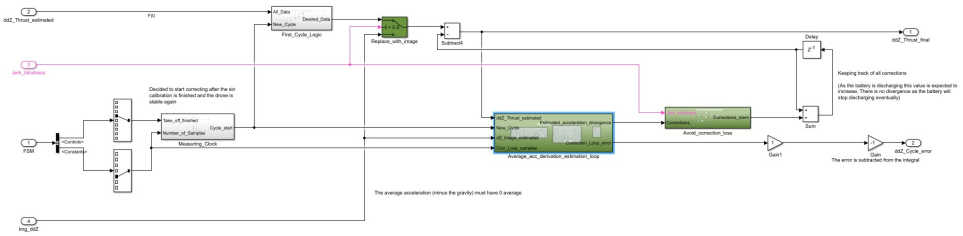

Figure 6.22: This is the running corrections block (it is similar for all 3 signals)

This block receives the initial enable and the number of samples for each correction cycle from the FSM.

The acceleration signal enters the block and is put to zero by registering the first sample in "First_Cycle_Logic". This is similar to the "wait for zero" logic that the other 2 signals have.

A periodic Clock is enabled and will generate a pulse to start a new cycle every "number of samples" value read from the FSM (see Figure 6.23). This block is used in all 3 applications. Every new cycle, the value of the free-running counter is registered and subtracted from the counter itself. When the time difference between the beginning and the present is found to be equal to the number of samples, a reset is self-performed, and a new pulse is generated.

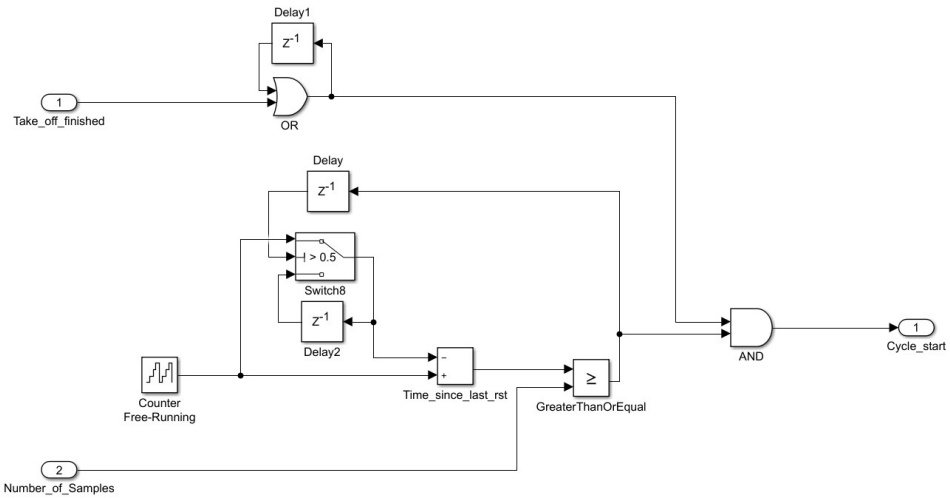

Figure 6.23: This is the running corrections clock (it is similar for all 3 signals)

In every cycle, errors are calculated through the resettable averaging circuit "Average_acc_derivation_estimation_loop" (see Figure 6.24), where the average difference between the image and the thrust is computed. The average image is considered to be correct, as said in chapter 6.1.2.7.

Notice that this block has a series of "first use only" logic: this is needed to ensure that the initial average starts from 0 and that the first calculated error is corrected with a double gain. The latter decision was taken because at the first time, the divergence will likely look like a triangle, while in later iterations, it should have a similar shape to the correct signal. Furthermore, it is possible to see that the measured error is immediately subtracted from the signal, and a secondary correction "per sample" value is adjusted by keeping track of previous data. The reason why previous errors are needed can be explained with a simple example: In a loop where the input already has a correction per sample, the error estimation will result as "error - correction," meaning that setting up a correction with such value won't be a complete error elimination by definition.

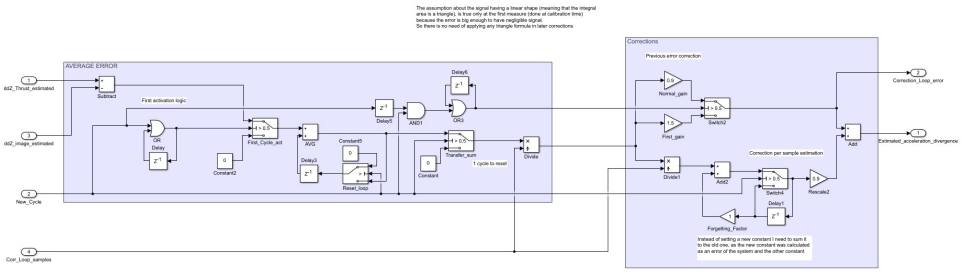

Figure 6.24: Here it is where all acceleration error is calculated. The forgetting factor was inspired by neural networks' training

After having all the corrections, the Jerk blindness logic is encountered (Figure 6.25). This block in normal conditions is transparent (signal passes through untouched), but when a jerk interval is detected, this block starts summing all corrections samples together while setting to zero its output. In the first sample, where no Jerk is detected anymore, all summed corrections are added to the instant correction of that instant. This block, thus saves all corrections for later because, during the jerk interval, the thrust signal is replaced with the pure image signal (see Figure 6.22). The "Memory2" block is in charge of retrieving the saved data and adding it to the first transparent sample.

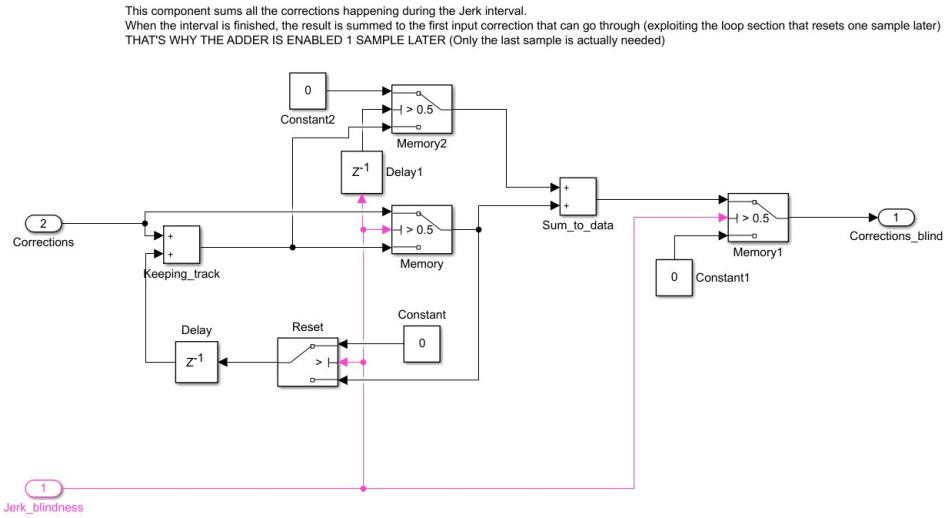

Figure 6.25: Jerk blindness error recovery logic

Finally all corrections are summed with the past ones. This happens only in the acceleration logic, as the next signals will have an integration already doing it.

2. **Speed**: Most of the blocks are similar to the acceleration corrections (see Figure 6.26). The main differences are in the absence of the "set zero" block, as this operation is already performed by the FSM, and in the presence of a signal integration.

   Notice also that corrections are starting when enabled AND when the integral is enabled (for a better conceptual solidity).

   Furthermore, here it is well visible the purpose of the jerk logic: in a Jerk interval all the integral value is replaced with the image, resulting in an open ring structure and a consequent flush. The Jerk logic will recover all corrections.

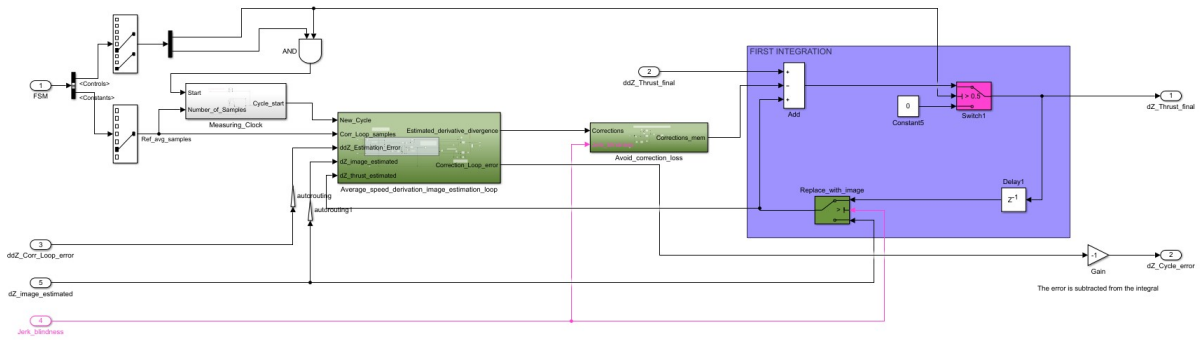

Figure 6.26: Here it is where all speed error is calculated. The forgetting factor was inspired by neural networks' training

3. **Position**: Here, the algorithm is almost identical to the speed version (see Figure 6.27). The main difference stands in the fact that the integral is not opened for being replaced with the image signal during jerk intervals. This is because the same operation was already performed in the previous step, and no disturbances would arrive at this integral.

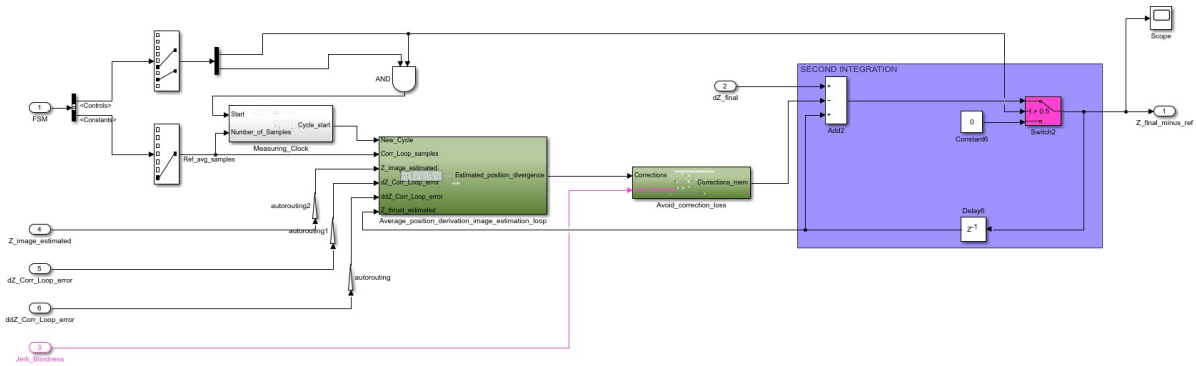

Figure 6.27: Here it is where all position error is calculated. The forgetting factor was inspired by neural networks' training

6.1.2.10 Simulation Results

In simulation, it was possible to observe this algorithm correctly working. It was possible to see that signals would stabilize and were able to control, almost as perfectly as, in the sonar case, the altitude. Figure 6.28 represents a successful altitude control in simulation: in this case, the acceleration correction is not being performed, and the old image correction algorithm (end of 6.1.2.4) is used. This is because the described altitude estimator was already considering some real-world characteristics not visible in the simulation.

6.1.2.11 Real World Application Results

By transferring everything to the real world, it was found out that the simulation model was simplified and didn't really match reality. This model was meant to be used to design tracking algorithms and not altitude controls, so for software optimization, some parts were simplified.

1. **Battery**: as visible in chapter 5.2, the voltage output is decreasing following a random sigmoid curve. This results in an increase in the engines' power outputs that is not possible to be correctly estimated and corrected (in Figure 6.29 it is possible to see engines power

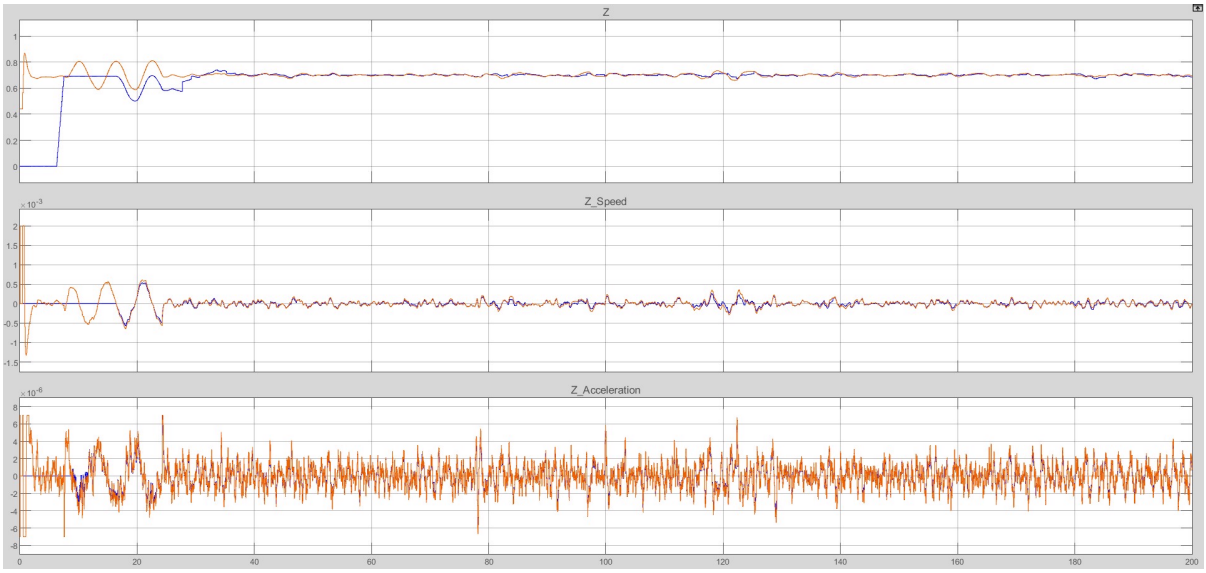

Figure 6.28: In this experiment, the drone is not using the sonar between second 40 and 175

increasing in a sigmoid-like way). Furthermore, the controller will go slightly out of tune as part of its driving capability is counteracting the battery discharge.

2. **Yaw**: It was observed that it is not possible to keep an acceptable yaw all the time. The reason behind this is that in a real drone, when all rotors have the same thrust force, probably the total angular moment is not zero because of minor imperfections. As a consequence, in a real drone, it is not possible to precisely control both Yaw and position. This explains the existence of drones with more than 4 rotors, as having more blades will, on average, better counteract their imperfections.

3. **Camera Noise**: In the real world also the camera presents noise. In Figure 6.30, it is possible to observe some ramps that may be caused by the yaw and some peaks caused by camera noise.

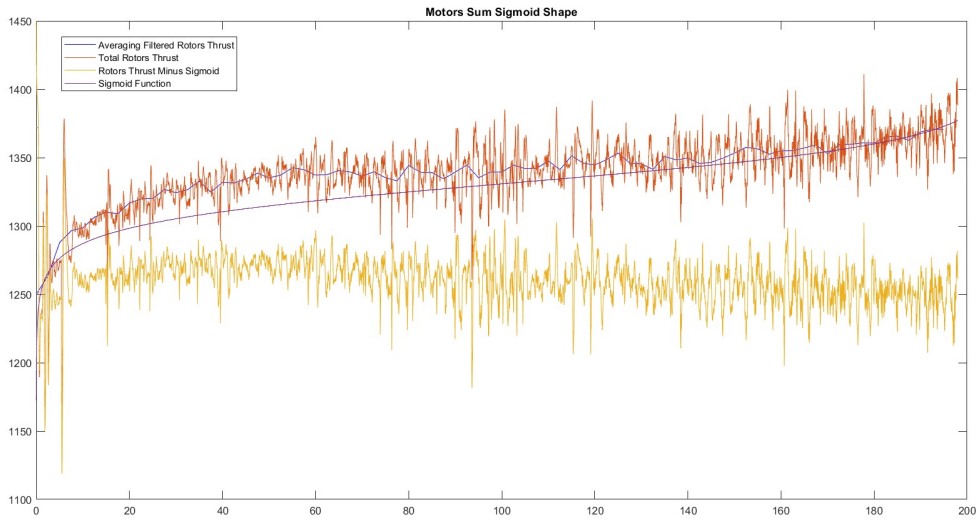

Figure 6.29: This time an attempt to regularize the output was made, but the shape is always different for every experiment

Finally, it is well visible that even with all correction algorithms, there are still some too-large random aspects in the 2 used signals. Increasing the estimation algorithm complexity might find a solution, but in that case, the drone's computational power is not high enough.

In conclusion the best solution is mounting optical sensors.

6.2 Optical Interference

All drones have a downfacing camera, meaning that when a detection happens, the upper vehicle will see the other drone, and that might create some positional (XY) estimation interferences.

Such position estimation problems might be used to detect the lower drone from the top one, so some tests were run.

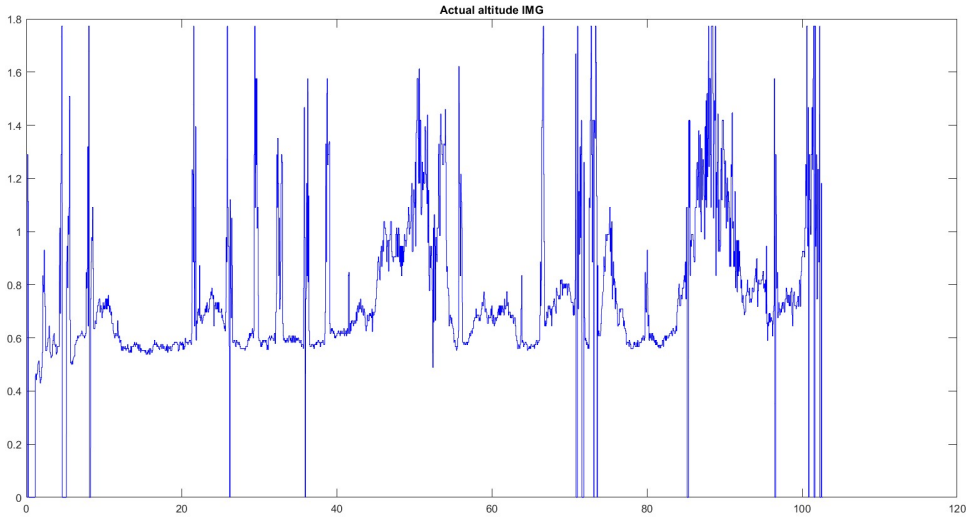

Figure 6.30: Here, the image altitude estimation signal measured in the real world is visible. It is very noisy and it can't be used to reliably validate the also very noisy thrust signal.

6.2.1 Run Tests and Results

The test was simple: Keep one drone hovering, try to pass under it with another one, and observe the results.

It was observed that no positional disturbances happened except for the sonar signal, which bounced on the other drone, resulting in an altitude estimation problem.

In conclusion, this interference is not effective in the system's behavior.

Other solutions might be used to detect the lower drone, but they haven't been explored further.

CHAPTER 7

INTERACTION MEASURES AND MODEL

To develop a drone-following algorithm using only blades' wind detection, the first thing to do is to actually measure and understand how the drones behave when encountering such airflows.

7.1 Initial Measuring Procedure for First Detection

Once had 2 calibrated drones, the lower one was programmed such that it was moving forward, and the higher one was only spinning its rotors while being kept in place by hand. By making the 2 drones interact, it was found out about the sonar interference problem (6.1), and for completing the measures, the higher drone's sensor was covered to avoid issues.

In the main result of this experiment, it was observed a sudden tilt forward on the lower drone, and as expected, in the measured data, the estimated pitch recorded a spike. Repeating the measure multiple times, it was found that the pitch spike had a value of -0.06 most of the time during detection.

Further observations showed that the pitch detection peak value is correlated with the approaching speed. In fact, in Figure 7.1 and Figure 7.2, 2 different speed cases are observed, and in the slower one, a detection is missed.

Looking at the control system for the pitch in Figure 7.3, let's call the "Control"'s block delay τ.

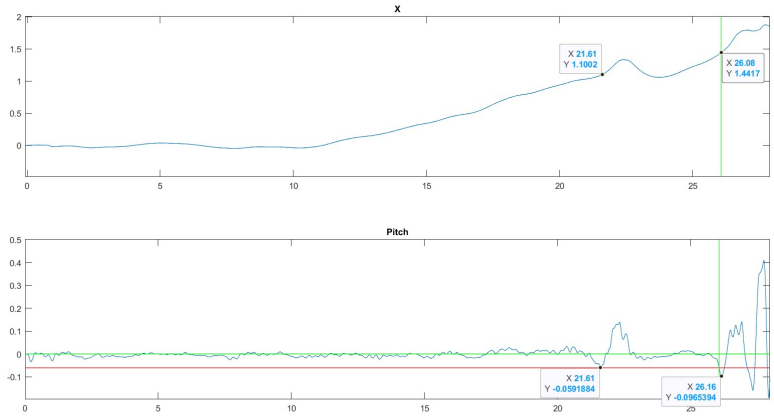

Figure 7.1: Drone detection at slow speed (0.0003m/sample = 0.06 m/s). A detection is even missed

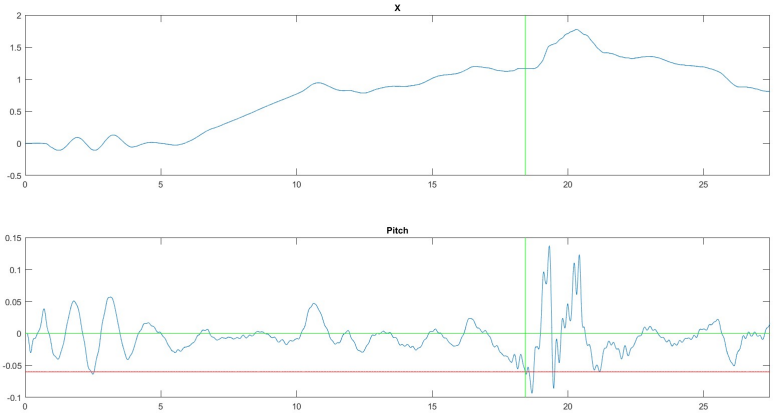

Figure 7.2: Drone detection with higher approaching speed (0.0006m/sample = 0.12 m/s)

As the pitch is related, through the function "R", to the wind force "F" and since the latter changes with the position x, it is possible to write:

$$Pitch(t) = R\left(F\left(x(t)\right), Pitch(t-\tau)\right) \tag{7.1}$$

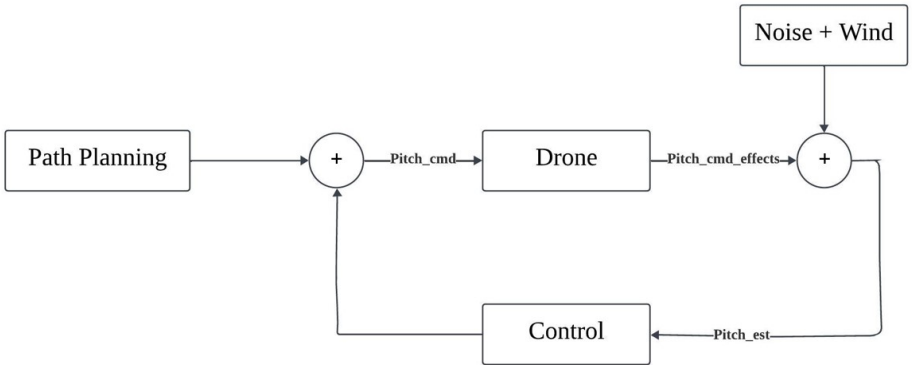

Figure 7.3: General Structure of the Pitch Controller

The controller is a PID type, and the feedback loop is included in the function "R". Thus, that function depends also on a linear combination between the Pitch, its derivative, and its integral signals of "τ" time ago whereas τ is the feedback circuit delay. Considering that in a drone detection procedure, the Pitch signal will be approximately close to zero until the peak happens, it is possible to say that in the time interval between when the **peak begins** t_b and the instant $t_b + \tau$, the function R(t) doesn't depend on previous pitch values as they are all ≈ 0.

$$R(F(t), Pitch(t-\tau)) \approx R(F(t), 0) = R(F(t)) \tag{7.2a}$$

$$\frac{\partial Pitch(t)}{\partial t} = \frac{\partial R\left(F\left(x(t)\right)\right)}{\partial t} = \frac{\partial R\left(F\left(x(t)\right)\right)}{\partial F\left(x(t)\right)} \cdot \frac{\partial F\left(x(t)\right)}{\partial x(t)} \cdot \frac{\partial x(t)}{\partial t} \tag{7.2b}$$

This demonstrates that the pitch angle derivative, in the initial time interval, is directly related to the speed.

With that being considered, calling the speed S(t), calculating back the Pitch signal:

$$Pitch(t_b + \tau) = \int_{-\infty}^{t_b+\tau} \frac{\partial Pitch(t)}{\partial t} dt \tag{7.3a}$$

$$Pitch(t_b + \tau) = \int_{t_b}^{t_b+\tau} \frac{\partial R\left(F\left(x(t)\right)\right)}{\partial F\left(x(t)\right)} \cdot \frac{\partial F\left(x(t)\right)}{\partial x(t)} \cdot S(t) dt \tag{7.3b}$$

In the observed case, the speed was constant, meaning that it is possible to see that the value $Pitch(t_b+\tau)$ is directly proportional to the speed. This demonstrates the observed phenomenon of having a larger pitch magnitude with a higher speed.

Finally, these measures pointed out that using the "estimated pitch" signal, false detections and missed detections might happen (see Figure 7.2 and Figure 7.1 where if there isn't a false or missed detection the signal is still very noisy). At low battery, the probability of this phenomenon is usually increased; in fact, the problem is related to the controller gains. The controller was tuned in charged battery conditions, meaning that when at a lower percentage, gains will be too low for correct behavior and, together with rare occurrences, might cause some temporary oscillations in the feedback loop (see chapters 5.2 and 6.1.2.11). These oscillations will then be summed to the detection signal, causing additive or destructive interference. The former stands behind the false detections, while the latter is correlated with the missed detections.

Because of the feedback delay "τ", external airflow has no instant correlation with the pitch controller output signal, meaning that by subtracting it to the estimated one, air effects can be isolated. The signal to be subtracted is "pitch_cmd" (visible in Figure 4.1), and with this operation, it was observed indeed a great reduction of false and missed detections (see

Figure 7.4). Rare external occurrences might still cause wrong detections, but for now, it was assumed to have calm surrounding air. This method is based on an approximation: previous calculations considered as control output the signal "pitch_cmd_effect" given as input to the drone while only the signal "pitch_cmd" is available (Figure 7.3). Since it is not possible to predict the control output precisely and delaying the estimation signal would result in a cleaner but late detection, this approximation was obliged. Such delay was observed to be small enough not to have considerable effects on the measure. Except in the late detection peak (detection already happened), the control signal derivative is contained, so the delay effect is negligible.

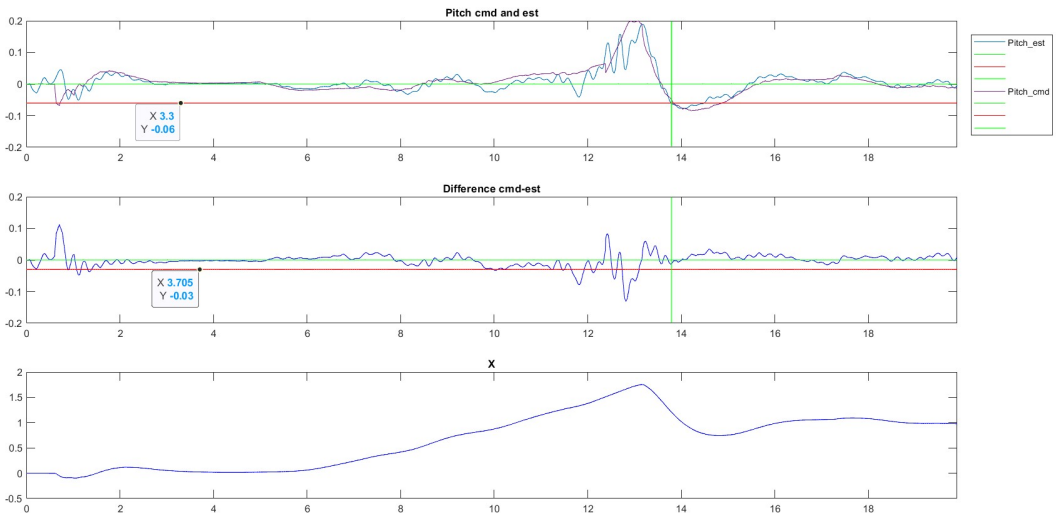

Figure 7.4: Successful Recovery of a missed detection

In this way, the concept of **detection peak** was defined as the pitch peak happening at the **detection point**, which marks the effective border of the downdraft region. "Effective" because there isn't a proper border but rather an increase in the wind force.

7.1.1 Altitude Difference Analysis

Another aspect that influences the detection peak's intensity is the altitude difference between drones. Using 2 different models of drone leaders, it was noticed immediately that even if they had similar size and weight, the Hasakee Q9s (Figure 5.13) produced a much more powerful airflow, meaning that they had to stay at different altitudes for optimal detections. This effect might be caused by the different propeller shapes as in the Parrot Mambo they are double, while in the other they are triple. The result might be that if we used the same definition of the ultrasonic sensors (5.1.1.2), the Parrot Mambo has a wider beam, resulting in a lower applied air pressure to the follower drone.

As a consequence, this simple experiment demonstrates that the altitude difference is strongly correlated to the specific drone model, and altitude tests must be performed on every new quadcopter model.

Tests performed have shown to have 3 possible outcomes depending on the altitude difference:

- **Low altitude difference** (< 50 cm for Parrot Mambo and < 70 cm for Hasakee Q9s): the pressure force applied to the secondary drone is too high. As a result, a dangerously steep pitch inclination is induced, having a high probability of causing a crash. This altitude difference must be avoided as it causes instability.

- **Optimal altitude difference** ((60 ± 10) cm for Parrot Mambo and (80 ± 10) cm for Hasakee Q9s): Here, the crash probability is well contained and has similar values to the "detection miss" probability. This is the optimal case because as the probability of a correct detection is at its maximum.
- **High altitude difference** (> 70 cm for Parrot Mambi and > 90 cm for Hasakee Q9s): at this point, detections will start to be missed or delayed (sometimes a missed detection might start some oscillations wide enough to cause a detection later (see example in Figure 7.4 for the case without correction)). This case is not as dangerous as the low altitude difference, but it still has two drawbacks as consequences: first, leader overtakes might happen; second, if we wanted to use more drones, they would occupy a huge vertical space.

In Figure 7.5, there is a visual representation of the 3 previously described probabilities.

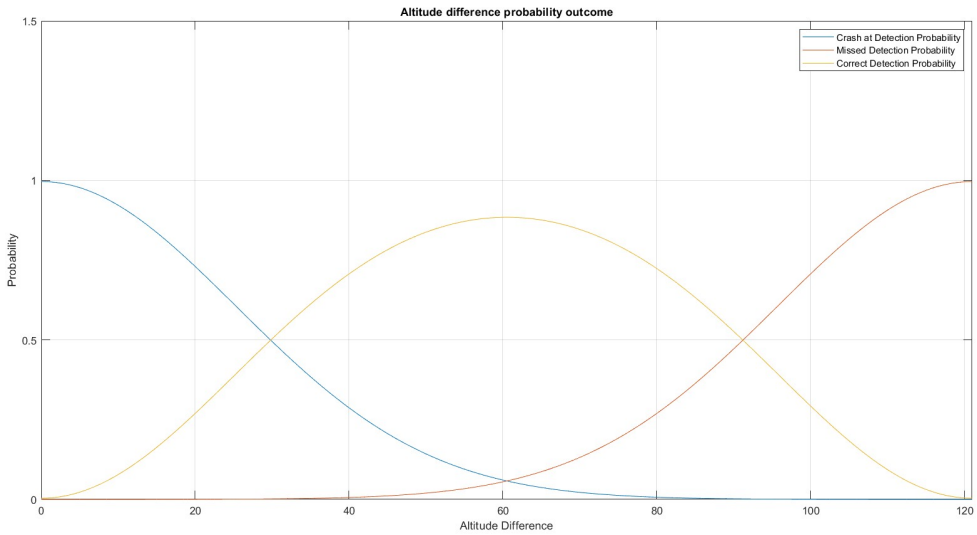

Figure 7.5: Visual representation of behavioral probabilities depending on the altitude difference

In conclusion, the altitude difference must stay in the optimal range to ensure the best working behavior. However, this aspect is strongly dependent on the drone's structure, meaning that for any new application, some initial measures following the used procedure must be executed.

7.1.2 Detection Model Validation Experiment

To validate the previous result and to better understand where drone detection happens, a software kill switch was connected to the detection signal. This simple experiment helped to better understand where the detection point is located with respect to the upper drone position. In the previous case, the drone was set just to move forward, making it difficult to understand the precise detection position. As soon as a peak was observed, the drone would shut down and land on the spot, marking the detection point. Repeating this test, it was found that the detection can happen from directly under the higher drone to 60 cm before. As seen before, it was observed that the detection point's relative position with respect to the other drone depends on the speed as well. To demonstrate this behavior, recall the previous demonstration around (Equation 7.3b), and since the initial speed is constant (Because of the pitch change, at detection, a slight acceleration happens), it is easy to understand that the space covered between "t_b"and"$t_b + \tau$" is greater at greater speed.

7.1.3 Detection Stabilization Experiment

Similarly to the previous test, this time, the drone was set to stop moving forward when a detection happens. Because of the huge amount of random variables (including the random

position of the detection point), the drone was stabilizing between 20 and 80 centimeters behind the leader, and its flight could be subdivided into 4 phases, visible in Figure 7.6.

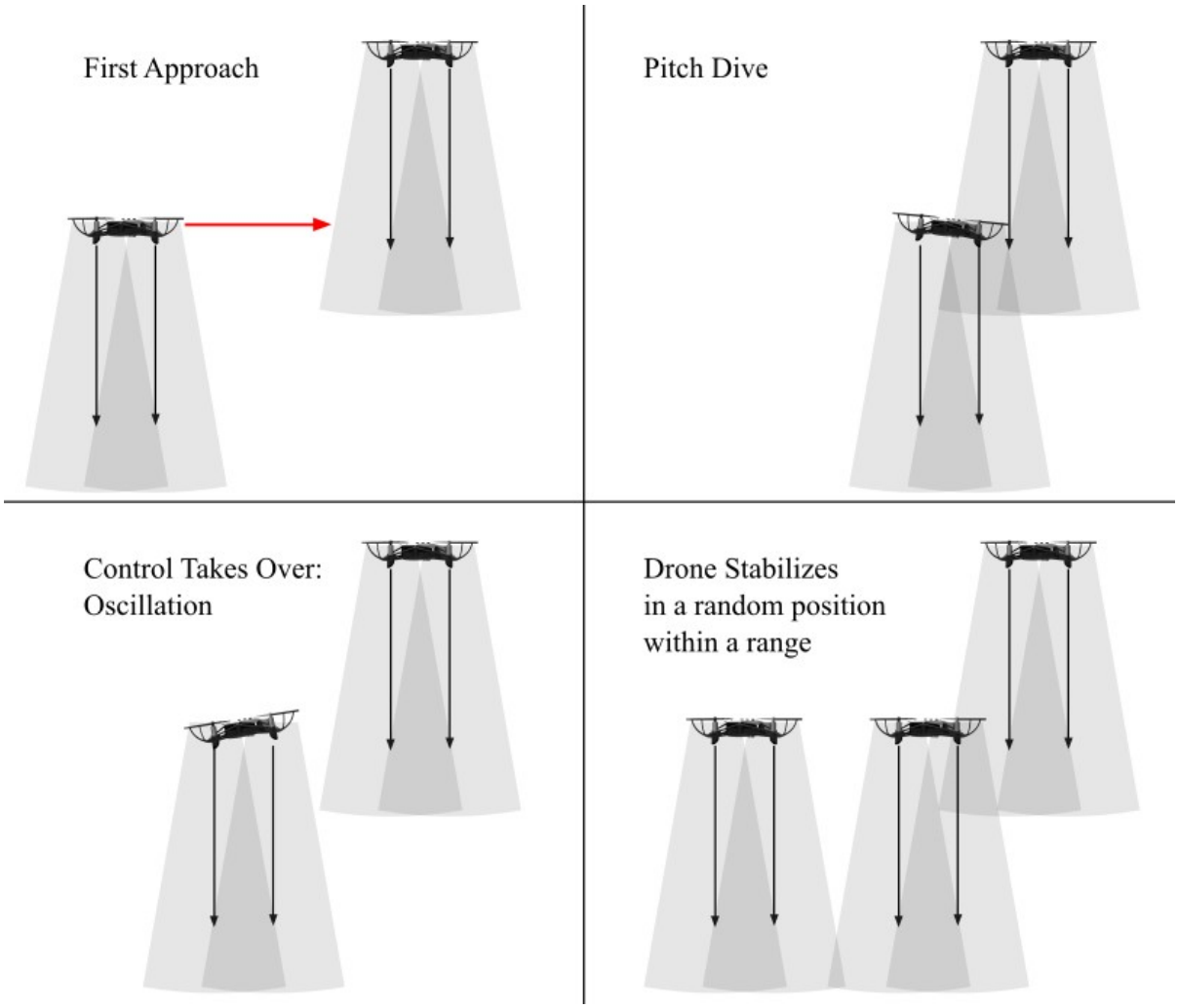

Figure 7.6: Summary of what could be observed

7.2 Continuous Detection Attempt for Position Control

Now that the drone is able to detect the first encounter, the ability to follow it must be implemented. Since the downward airflow produces a constant force on the front side of the lower drone, it means that the pitch difference between the measured signal and the control output must be a bias $b \neq 0$. The initial idea was to set up a PD controller that, reading this signal, would keep the drone close to the leading vehicle.

7.2.1 Controller Tuning via Try and Error Method Attempt

Following this idea, a PD controller (Figure 7.7) was designed, and weights were tried to be set. This controller would have been activated after the first detection (and after take-off stabilization, obviously) because its objective was to "stay in position" and not to find the leader.

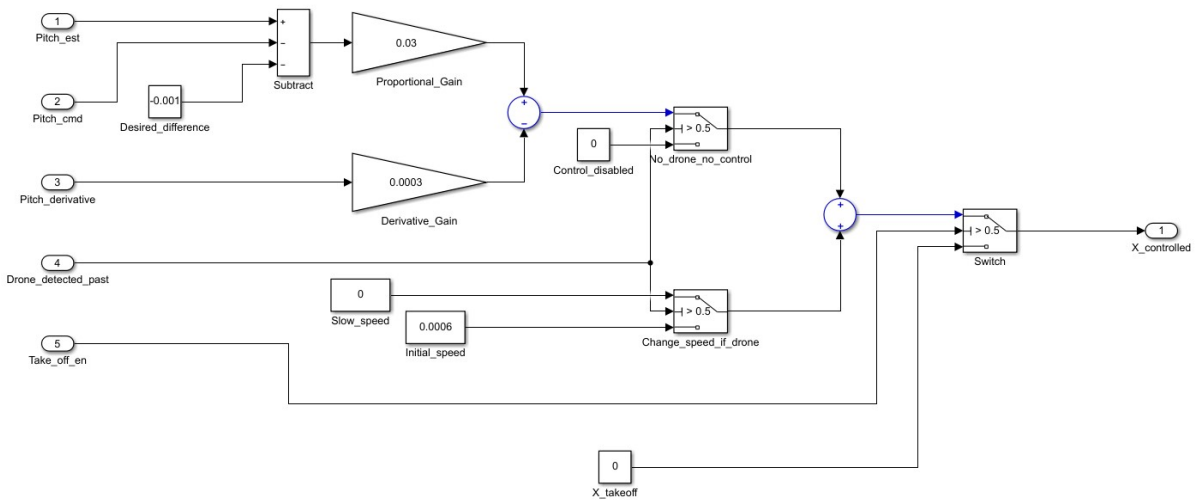

Figure 7.7: This PD controller on the top will be enabled from the first detection onward.

The objective was to try and stabilize the controller to an arbitrary pitch difference value ("Desired_difference" in Figure 7.7). After many attempts, it was possible to tune the controller so that it could keep the drone in place only if the leading drone was not moving. This is because, to avoid diverging oscillations, all gains were reduced until the drone was extremely slow to move. Obviously, this is a degenerate case, and it was not possible to find the desired solution as diverging oscillations were happening in all other cases. This solution worked only if the leader moved slowly backward (as it would encounter the other and "push it back"). As a final attempt, it was tried to define a "slow speed" that would force the drone to always move forward (and force new encounters), hoping that when in position, the controller would cancel it, but then leader overtakes started to occur, and this solution was discarded.

Multiple reasons behind this problem are the following:

- As found before, the detecting pitch signal has a spiking behavior depending on the approaching speed. Since the controller will try to keep the system in place by approaching the desired spot at a random speed, the input signal will be heavily affected. The controller was working only when the leader wasn't moving or retracting because the relative speed was keeping the same value, 0, or because when the leader retracts, a new pitch peak happens, and the normal flight controller would move the drone away to stabilize it again.
- Since now the relative speed of the 2 drones is much lower than at the detection time, all effects will be reduced down to the noise order of magnitude.

- These experiments had two drones flying closely, and here it is where the sonar issue became not tolerable anymore (see 6.1).

7.2.2 Conclusion

In the end, this method was abandoned as it did not have a satisfying solution. However, the attempt to add a fixed speed gave the initial idea for the next solution.

7.3 Finite State Machine Following Algorithm for Speed Control

Since no other signal was found to be clearer than the first detection signal, this method's idea was to repeat the same approaching procedure. In the previous attempt, the controller wasn't able to control the position with respect to the leader independently from the slow speed value, so the idea that came out was to force it to change the speed via software and perform these multiple approaches. After detecting the leader, the drone will immediately back up and start a new approaching cycle. This solution can have multiple developments.

7.3.1 Pure Finite State Machine

A first test of this method was implemented by modifying the program where the drone was landing at leader detection. Now, an FSM controls the speed so that at every detection, the drone reverses for an arbitrary amount of time before restarting the forward movement. This idea would avoid the overtakes that were happening previously. If the leader starts to move, the approaching procedure would indirectly follow it.

The results of this experiment showed that if the backward movement wasn't lasting long enough, there was a high chance of having the next detection point closer to the leader drone, independently from the approaching speed. This effect happens because the drone needs to

be moving forward to actually sense the downdraft region border. If there wasn't a minimum amount of distance, the actual detection point would be overtaken. Furthermore, if the drone restarted moving forward while still in the downdraft region, there would be a very high chance of it overtaking the leader without sensing it. This effect was particularly bad when the leader was going backward. An easy solution was to set a minimum distance from the last detection point so that the drone would keep the reverse speed enabled if it couldn't get far enough during the time given.

From this experiment, the first working prototype of the drone-following algorithm was obtained, with some margin of improvement. However, in some cases (backward leader, for instance), the speed difference might be too big, making the sensing difficult.

7.3.2 Hybrid Control and Finite State Machine with Speed estimation

Once the last experiment was completed, it was found that recording the X coordinates and the timestamps of the detection points made it possible to estimate the leader's speed. The estimation was observed to be already good enough by using only the current and the last detection point.

Calling t_{past} the previous detection timestamp, X_{past} the previous detection point, $t_{present}$ the timestamp of the current detection, and $X_{present}$ the current detection point, leader's speed (S) can be estimated in the following way:

$$S = \frac{\Delta X}{\Delta t} \quad \Longrightarrow \quad S = \frac{X_{present} - X_{past}}{t_{present} - t_{past}} \tag{7.4}$$

Setting the speed to the estimated value (S) will result in 3 possible scenarios:

1. S is greater than the actual: soon, a new encounter will occur and cause a new detection that refines the measure.
2. S is greater than the actual, but the leader suddenly accelerates: the following drone won't be able to keep up with the leader as it doesn't know about the acceleration. No interactions will happen again, and references might be lost unless the leader decelerates.
3. S is close or lower than the actual speed: the 2 drones might keep the same distance forever or even get far apart. As in the previous case, no interactions will happen again, and references might be lost unless the leader comes back.

To avoid losing contact with the leader, it was defined that after an arbitrary amount of time, a forward acceleration must be forced. This solution will increase the frequency of leader detections and will contain the distance amount between the 2 vehicles.

For quicker backward movement and stabilization, a PD controller was tuned with the "try and error" method and is activated only for a short time after detection occurs. This controller is very similar to the position control experiment; the only differences are that it has an enable signal, and its gains are much higher (see Figure 8.4).

Since the detection position is saved, after disabling the controller, the FSM will manage the remaining backward movement and all other phases.

CHAPTER 8

DRONE FOLLOWING ALGORITHM WITH IMPLICIT COMMANDS DETECTION

After having completed all measures and tests the final drone-following algorithm was completed.

8.1 X Contol Logic

The algorithm was developed around controlling X, but it was predisposed to other solutions as well. The complete section on X is visible in the path planning block and in Figure 8.1.

This section also contains the speed commands decoder, which is a block able to receive commands by observing only the behavior of the following algorithm.

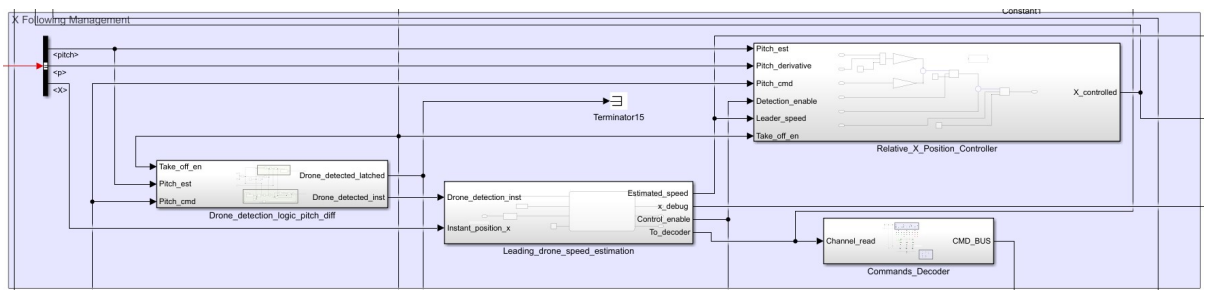

Figure 8.1: Complete X Following Logic

8.1.1 Finite State Machine

The management of all steps comes from the FSM in Figure 8.2 and are described below: All movement speeds in this section will be expressed in meters/sampling_time, meaning that they need to be multiplied by 200 to obtain m/s and by 720 for Km/h.

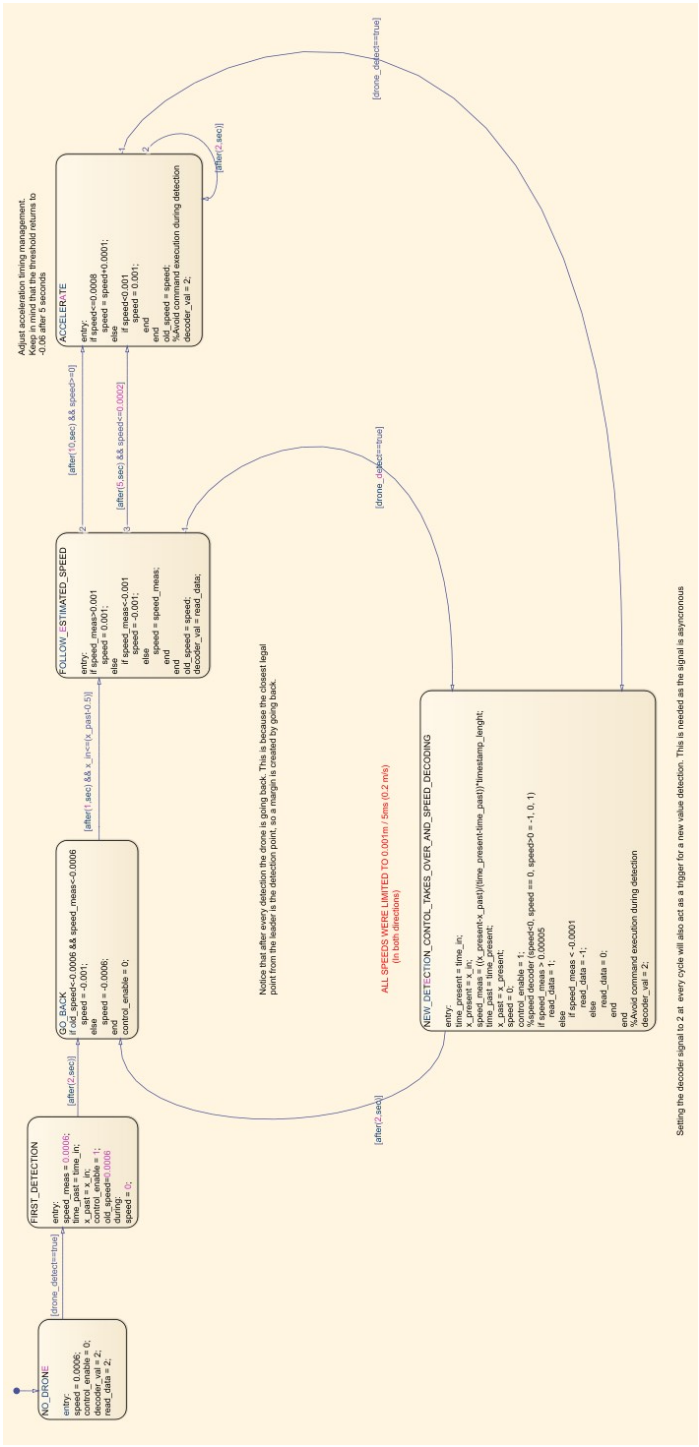

Figure 8.2: Finite State Machine in charge of managing the drone-following procedure

To see the following states in action observe Figure 8.10. I will put time intervals to better indicate.

1. **No Drone**: after take-off, no detection happened yet. The drone speed is set to an arbitrary value of 0.0006 m/sample (0.12 m/s) in the forward direction.

   This state will end with drone detection (interval in Figure 8.10 from 0 to 13).

2. **First Detection**: as no past data exists yet, it is not possible to estimate the leader's speed. This step, thus, will only register the detection position and timestamp and will set the "measured speed" to the same arbitrary value of the beginning state. As explained in chapter 7.3.2, after a detection, the PD controller must be enabled for a short time. This state is performing that operation as well.

   This state will end after the arbitrary time dedicated to the controller (interval in Figure 8.10 from 13 to 15 and 20 to 22 ...).

3. **Go Back**: for reliable detections, the drone needs to get far enough from the leader. This state will ensure that the next detection will start the next approaching cycle from an arbitrary distance. If the overall system speed is quick backward (faster than -0.0003m/sample = 0.06 m/s), the "go back" state will set a quicker backward speed at (-0.001m/sample while by default it is -0.0006).

   This state is the first encountered that will be repeated multiple times.

   After this step, the drone will be ready to keep the measured speed until the next detection or time-out (interval in Figure 8.10 from 15 to 19 and 22 to 26 ...).

4. **Follow Estimated Speed**: All speeds were limited to a maximum absolute value of 0.001 m/sample (0.2 m/s) to avoid wrong high measured values and for better safety.

   This step can end with a drone detection or a time-out (interval in Figure 8.10 from 19 to 20 and 26 to 31 ...).

5. **Accelerate**: in case of a time-out, the drone might be in one of those situations where the leader is never met again (see 7.3.2), and to avoid that, its speed is increased.

   Based on the speed's previous value, time-out value changes (if speed lower than 0.0002 m/sample the time is halved from 10 sec to 5), and in case of no detection even after a first acceleration doesn't happen, a second can be performed. Also, the speed is limited to 0.001 m/sample.

   This stage ends with a detection (interval in Figure 8.10 from 31 to 53).

6. **New Detection**: here it is where the leader speed is estimated following Equation 7.4 and the controller is enabled again.

   This stage ends after an arbitrary time and it is followed by another "go back" step.

8.1.2 Drone Detection Logic

Following the information found in chapter 7, this logic subtracts the pitch_cmd from the estimated pitch and searches for the detection peaks (Figure 8.3).

An isolated detection happens when the signal goes out of the interval (-0.06, 0.1) triggering the detection signal given to the FSM. From data analysis, it was observed that after a detection,

a small oscillation was initiated with peaks that might be lower than -0.06. As a consequence, the lower drone detection threshold is moved to -0.08 for 5 seconds to avoid false detections.

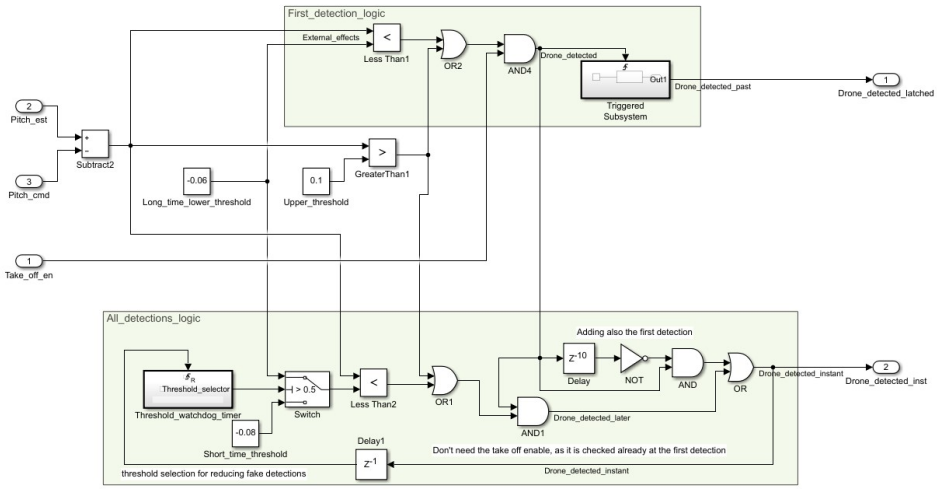

Figure 8.3: Pitch Detection Logic

8.1.3 PD Backward Movement Controller

For a quicker backward movement this controller is enabled for a short amount of time. The structure is the classic PD, and 2 switches are added to mute it. Tuning procedure had to happen through real-world tests, following the "Model-Free Design Procedure" (2.2) as a model precise enough for that action doesn't exist.

8.1.4 Implicit Communication Decoder

After having completed the previously described algorithm, it was noticed that it was frequently reading a signal value decided by an external drone. The idea is simple: by discretizing

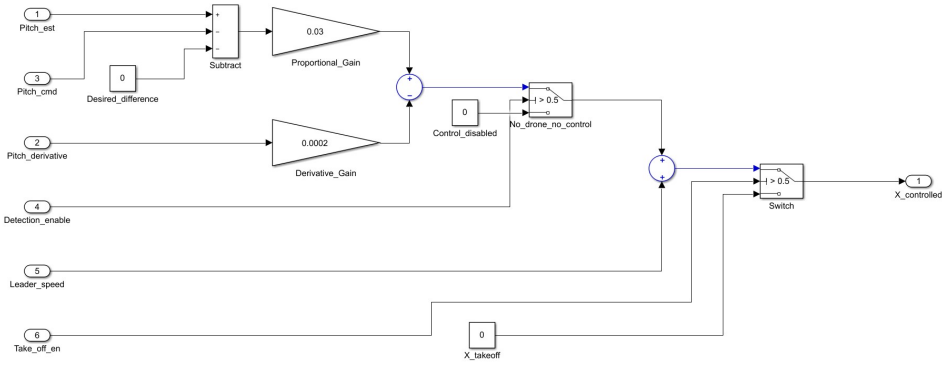

Figure 8.4: Controller used for quick backward movement with enabling switches

such signal and setting some arbitrary patterns, it is possible to define a series communication protocol that transmits data from the leader to the following drone.

8.1.4.1 Communication Protocol

A first test was set up. The serial communication protocol is defined such that at every new detection (in the "new detection" state), a new value is received and translated to one of the possible symbols. During the "Follow Estimated Speed" state, the symbol is fed to the decoder (see Figure 8.2). Outside of the "Follow Estimated Speed" state, the data is set as not valid (this choice was made to limit commands' execution to start in that state, far away from the leader).

The decoder block (visible in Figure 8.5) is composed of a shift register that saves the last "n" values and, by the actual decoder, is organized as an AND gate for each command.

A logic signal is set to 1 every time the input is not valid (not valid symbol = 2 in this example), and the shift register is designed to shift by one at each falling edge of this logic signal, resulting in reading the first sample of the valid data.

Once saved in the shift register, command execution becomes independent from the FSM, but it must be noticed that some orders might need to be stopped after some time to avoid losing the leader.

When a command is matched, its corresponding AND gate will have output at 1. All outputs are sent to a BUS signal that connects them to the management unit of their commands (see chapter 8.1.4.2).

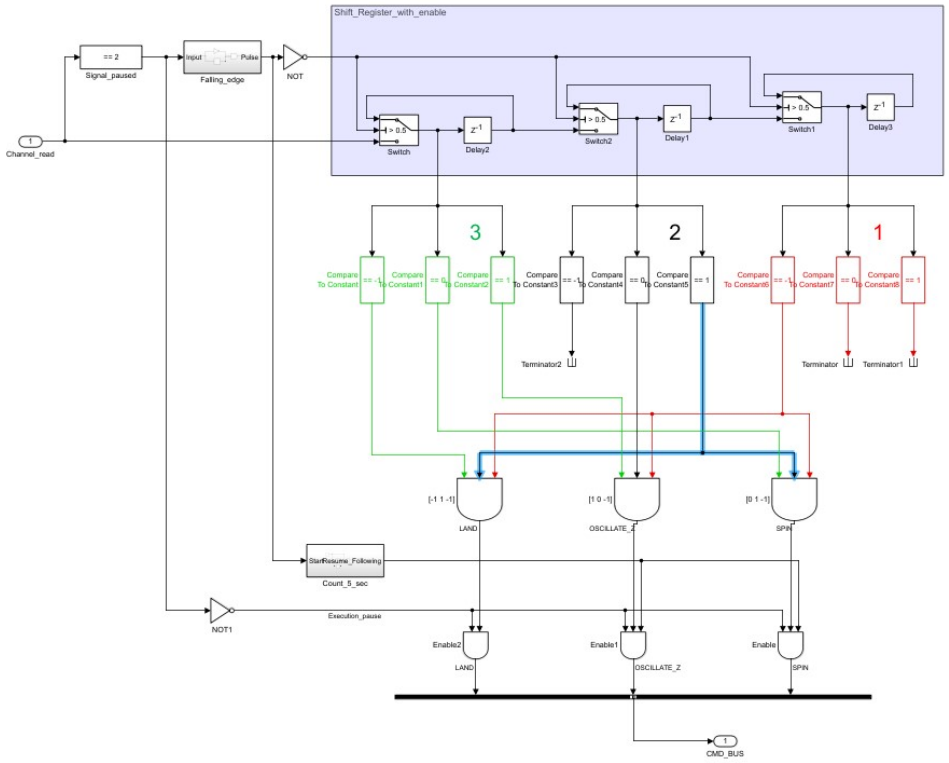

Figure 8.5: Commands decoder block with 3 commands examples

8.1.4.2 Command Execution Units

For the defined commands (Vertical Oscillation, Spin, and Land) all execution units are found in the central planning unit (Figure 8.11) and are visible in Figure 8.6, Figure 8.7, Figure 8.8 and Figure 8.9. It must be noted that some of these units are not directly executing the commands but are just muting some other signals to ensure the command's success (X and Y, for example).

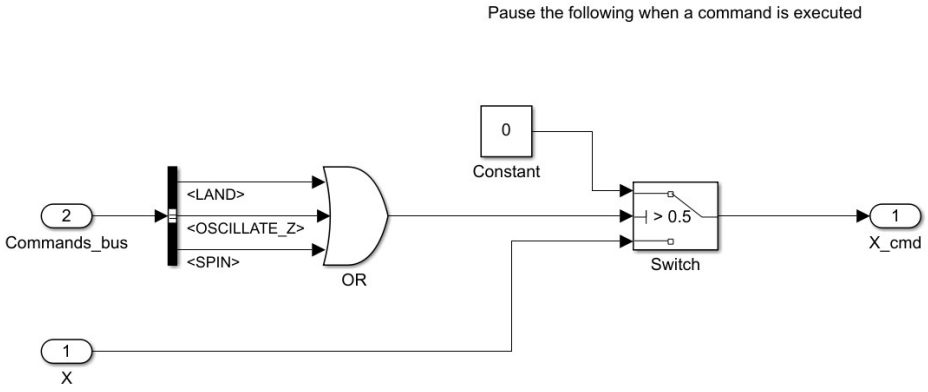

Figure 8.6: X execution unit

This decoder was tested with the landing command and the obtained results are visible in the graphs of Figure 8.10. It is well visible that the test was a success. The top graph shows that the drone landed, the second graph is the detection signal, the third graph reports the command speeds, and the last graph shows the read data. The codification of this command is [-1, 1, -1] and it is well visible in the graph.

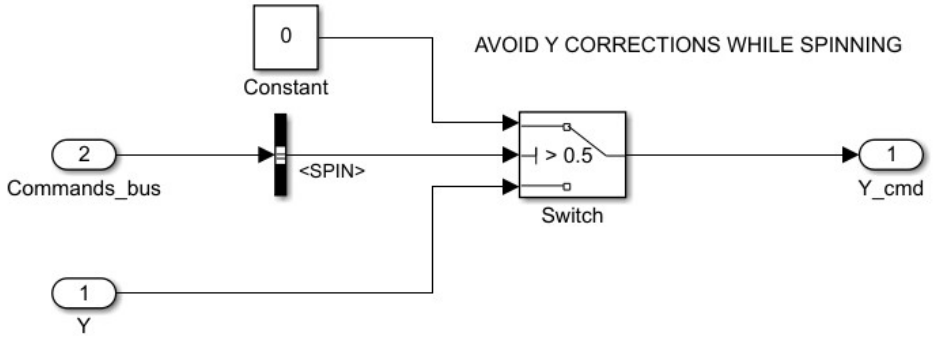

Figure 8.7: Y execution unit

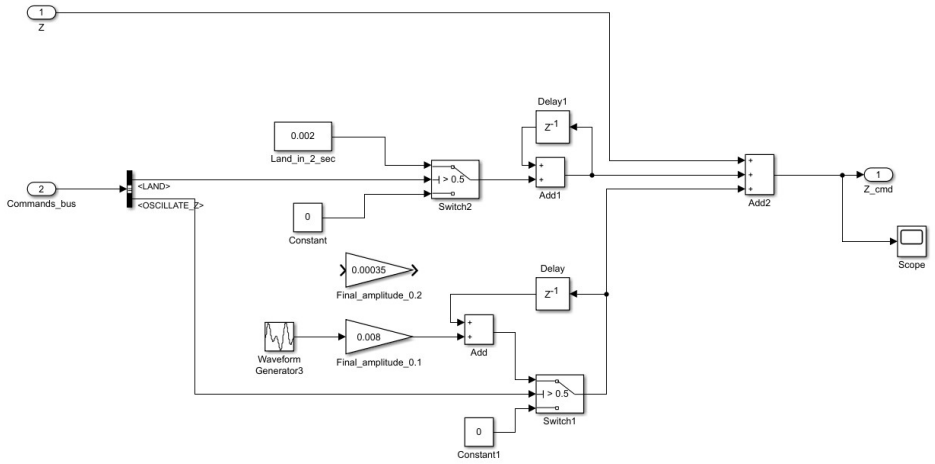

Figure 8.8: Z execution unit (Land, Vertical oscillation)

8.2 Central Planning Unit

This section is where everything is put together to create the final commands (Figure 8.11). This structure resembles Paolo's path-planning logic [20] (Figure 4.2) as it is a heavily modified evolution.

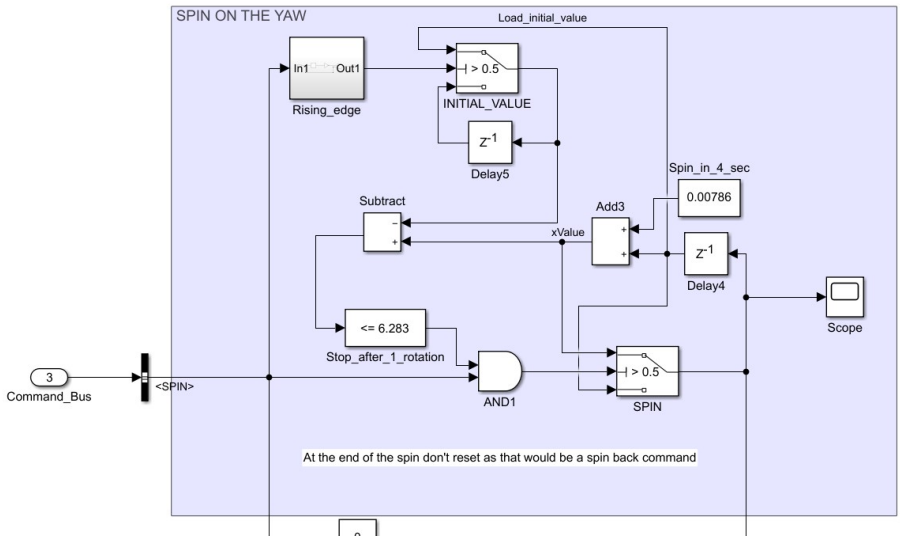

Figure 8.9: Yaw execution unit (spin)

8.2.1 Y Tracking function

Similarly to what was done by Paolo [20], Y is controlled to be over the white line by defining some gains to be added to the previous desired Y position (see Figure 8.11). This solution will be enough to balance positional estimation divergences.

8.2.2 Yaw Control function

In the same way as Y, a control was set up for the Yaw using the results of the algorithm of chapter 6.1.2.6. However, as described in chapter 6.1.2.11, it is not possible to precisely control both Yaw and position at the same time. For this reason, this control was decomposed into steps to be applied each second (see Figure 8.12): all samples coming from the "Tracking_Yaw" function are summed together and released together when the periodic signal decides so. They are not released in one sample only but a slightly wider interval must be used for not being

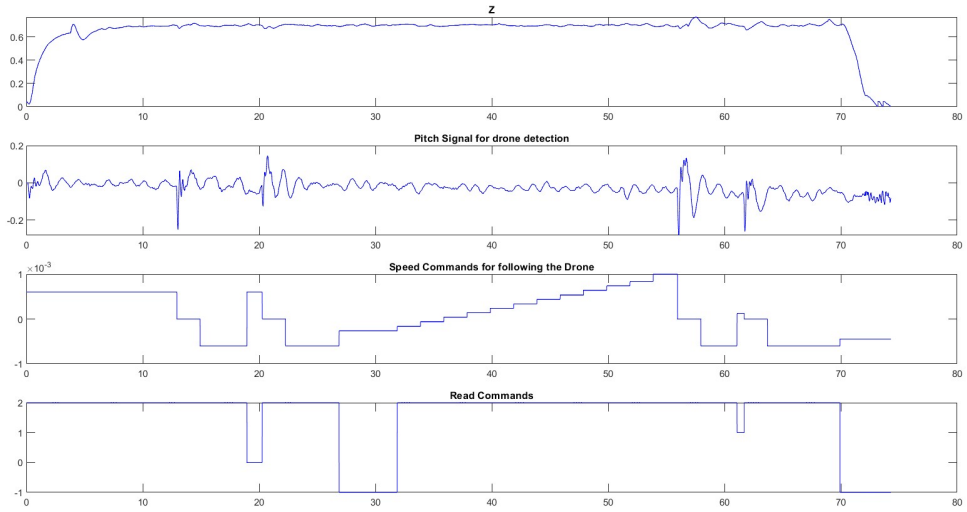

Figure 8.10: Evidence of the drone ability to correctly detect the "Land" command

completely filtered out. Furthermore, these step signals (they are not spikes because returning edges would cause anti-yaw corrections) are multiplied by a gain much higher than $2 \cdot \pi$. This is not a problem as they are only indicating the rotation speed and the drone will only move a little.

8.2.2.1 Yaw Errors Corrections

As the yaw divergence is not too big, it was defined that if there are more than "n" consecutive corrections in the same direction, the next one will be forced to be in the opposite way. This is because too many corrections on the same side will eventually take the drone outside the working range.

8.3 Complete Path Planning Logic

To conclude this chapter, the final path planning logic is shown in Figure 8.13.

The final elements included in that section are:

- **Jerk detection logic**: this operation takes as input the Z command signal, derivates it 3 times and outputs it to the "State Estimation" block. This will be needed to eventually cancel invalid Z data coming from the engines.
- **Calibration commands Logic**: These 2 blocks are in charge of starting the sinusoidal movement while calibrating and are in charge of enabling the normal movements.
- **Calibration Circuit**: this element is useful for finding the rotor's biases. It is a hovering circuit, and it can be connected to the "pos_ref" output signal.

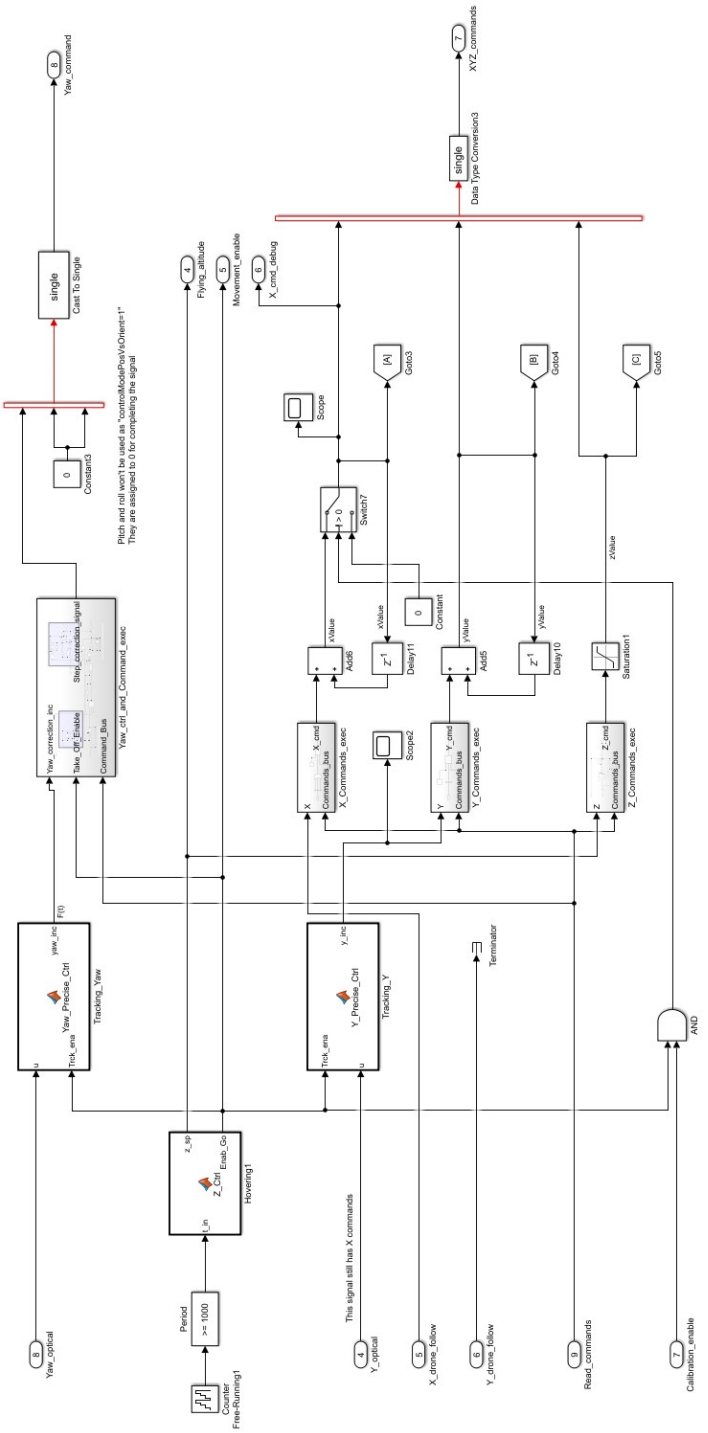

Figure 8.11: Central Planning Unit block

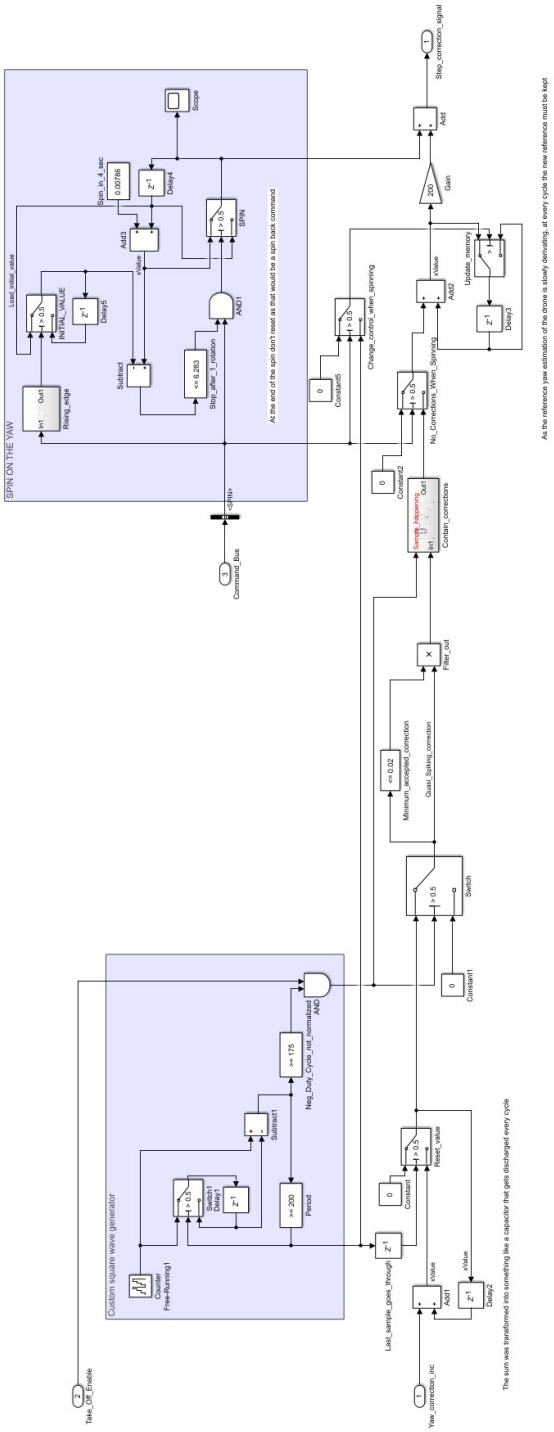

Figure 8.12: Yaw_ctrl_and_Command_exec block, where commands are applied in a spiking behavior

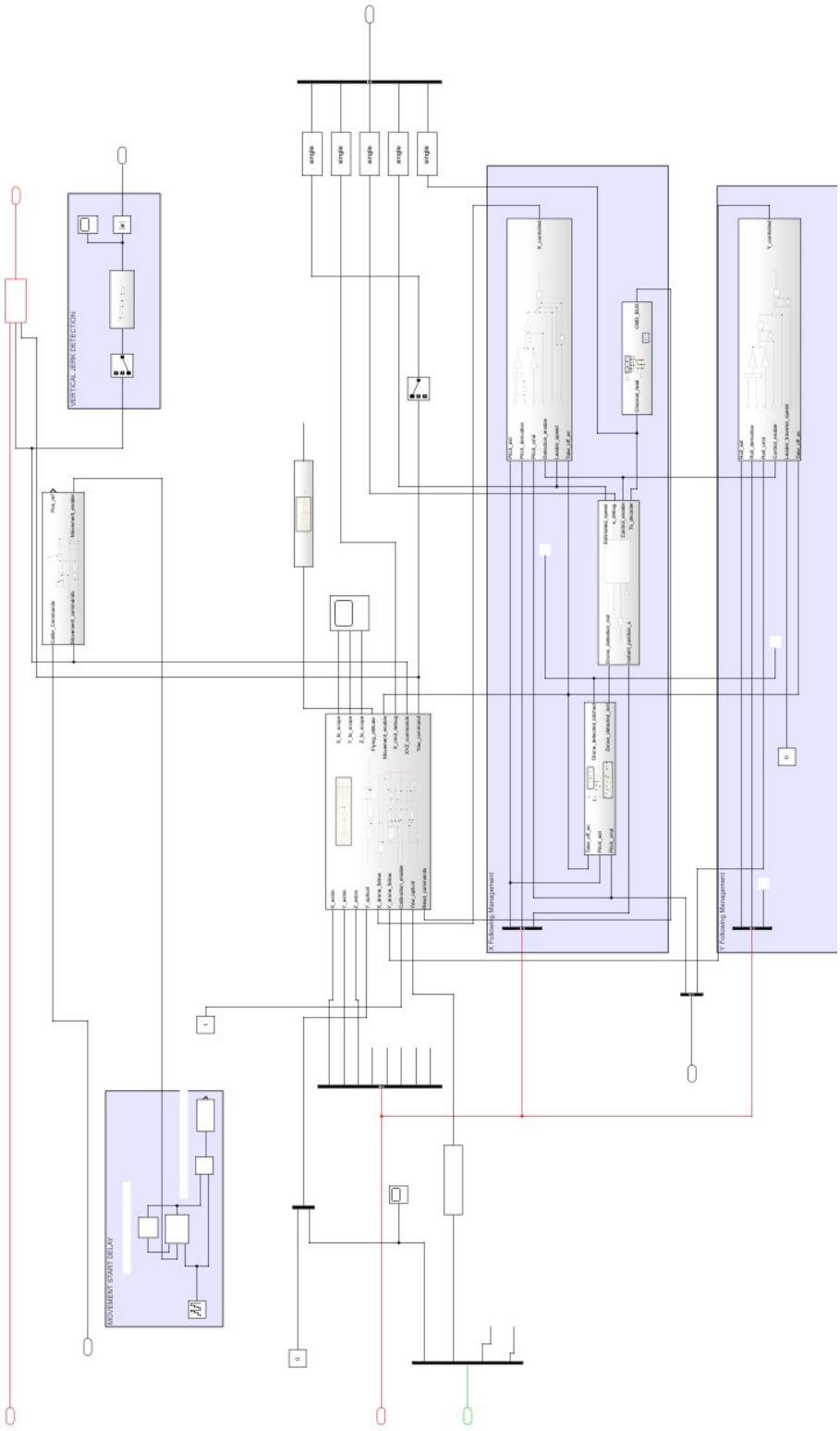

Figure 8.13: Full path planning logic

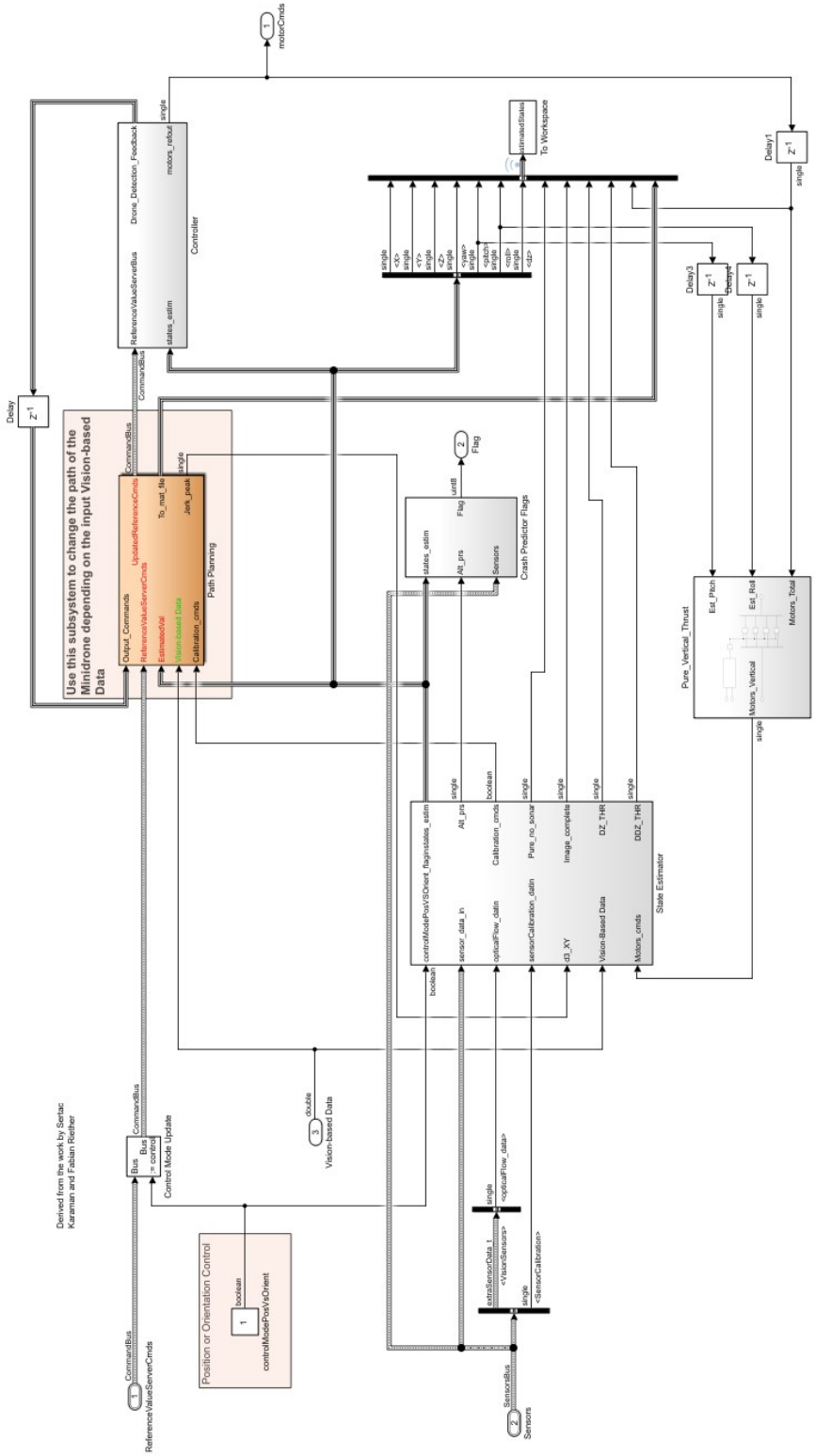

Figure 8.14: Full Drone Control System. To observe signals in the .mat output file connect them to the "to Workspace" block

CHAPTER 9

SYSTEM SETUP

To successfully set up an environment for running this process using Parrot Mambo minidrones, a sequence of steps must be followed. To configure the Matlab environment with the drone, first download all the required add-ons (see chapter 5.7), then go to the "Manage Add-Ons" list and search for "Simulink Support Package for Parrot Minidrones". This one will be the only extension (between the ones used) that will have an icon with the "settings" wheel on the right: clicking it, will start a guided procedure for setting up the environment.

1. If the project is starting from an external already existig one a few additional steps must be followed:

   (a) Download the "parrotMinidroneCompetition" project.

   (b) Set up the project: Simulink compiler was programmed in an optimized way so that if no project change is detected or if a file already exists, some code generation files won't be updated or created. Between these untouched files, there are some (like the make file) that contain full directory paths for the target building folder or for other scripts to be launched. When the project is started in a new computer or in a new directory path, when launching the code generation there will be a make target error. To force the creation or update of these files, there are 2 main procedures:

i. Create a brand new "parrotMinidroneCompetition" project, replace the file "flightControlSystem.slx" found in the "controller" folder and the file "asbQuadcopterWorld.wrl" in the "support" folder with the ones found in the existing project. In a brand-new project, target folder paths are not defined yet, and Simulink will immediately generate the correct ones

ii. Use the downloaded project, but make sure that the "work" folder (if it exists) is empty and the folder structure looks like Figure 9.1. This will eliminate all the wrong directory paths and force Simulink to replace them with new ones.

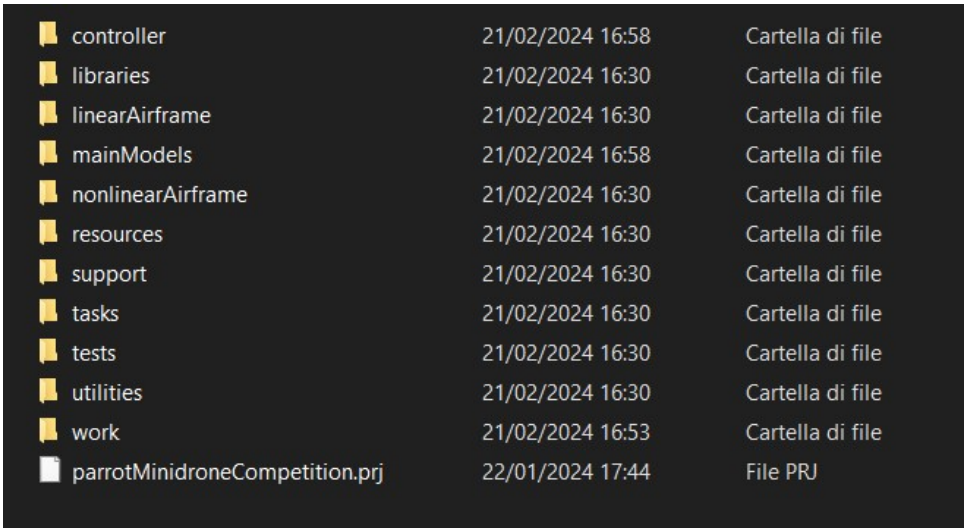

Figure 9.1: Working Project Folder Organization

2. Calibrate the drones: Connect the calibration block in the path planning to the output pos_ref signal to put the system in calibration mode. Once in calibration mode, save, generate, build, and deploy the model to the drone, start the flight, and wait for the

drone to stabilize at a random point. When the drone has been looking stable for around 5-10 seconds, stop the flight immediately and download the ".mat" file. Finally, open the result in Matlab and run the script in A.3.2 to obtain the calibration results and copy them to the block Figure 9.2. Notice that for this step, it is needed to connect to the .mat file block the image signal not rescaled (as this link was removed in the final version). This step is necessary for ensuring a smooth take-off on the spot and must be performed on every drone independently.

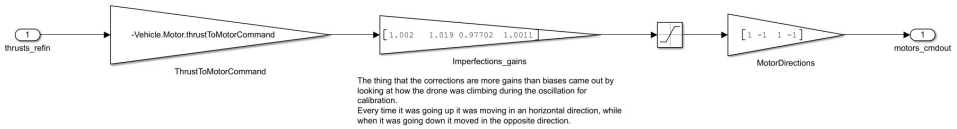

Figure 9.2: Here the rotors biases must be inserted for a more precise take-off

3. Set up a ground environment as described in chapter 6.1.2.3.
4. Launch the final project: once the pre-flight calibrations are done, resume the "normal mode" of the project and launch it. Recall that the drone will perform an initial calibration in the first part of the flight (see 6.1.2.8), so do not expect it to move forward immediately.

9.1 In case of not using a Parrot Mambo drone

In case a new drone model is used, some more steps must be executed. Firstly, it must be measured again the pitch signal to find the correct peak level for detection as it was explained in chapter 7. Secondly, the 2 constants Z_{ref} and P_{ref} of the image must be measured again following the procedure explained in chapter 6.1.2.2. Furthermore, a different drone might not

need the same ground environment, but in this case, the solution will need to adapt to the situation.

9.2 Other Rarer Encountered Problems' Solutions

While working on this program, some occasional problems came up. They are usually correlated with Simulink bugs and are usually solved by clearing the work folder, restarting Simulink, or restarting the whole PC. In some other cases these actions wouldn't solve the issue and the ones found are listed below.

9.2.1 Flight Data File Not Updating

This issue is related to an incomplete code generation problem (it is easy to notice that the code generation takes less time) and results in the drone always presenting the same ".mat" file after any flight. To solve it, a number of steps must be performed:

1. Check if, in the hardware specification of the project, there is the correct drone board indicated and set it if not.
2. Press consistently all project shortcuts (except code generation) such that a big pending instruction queue is formed.
3. The last 2 pressed project shortcuts of the sequence must be in order "Clear up project" followed by "generate code". This will obviously generate an error as the code generation won't be able to find any initialized variable. If everything went correctly at this point most of the code generation and model data would be invalidated.

4. Launch the project shortcuts "Initialize variables" and "Set up project" followed again by the "generate code" one. At this point, if everything goes correctly, there will be an error saying, "variable size arrays disabled."
5. As a consequence of the previous error, go to "modeling; model settings; Code generation; Interface; Software environment" and enable the "variable size signals".
6. Launch again the code generation: this time it should get the same error as before.
7. Without changing anything, launch the code generation again; this time, it should work.
8. As it wasn't a default project setting, disable the "variable size signals" that were enabled in point 5 and launch the code generation again. This time, everything should work, and the drone should be able to update again the ".mat" file.

9.2.2 Small Room only Available

This issue is related to the sonar signal bouncing on a small room's walls. In this case, a sponge was used to filter out, side reflections, allowing the drone to fly at higher altitudes without this issue. In the room used to develop the algorithm, the drone would start having side wall issues when flying higher than 1 meter. By modifying its body with the sponge, it was able to reach up to 1.5 meters without any issue when flying in the whole test area. This problem is correlated to the sonar beam angle, and it was described in chapter 5.1.1.2.

9.2.3 Software Vulnerability

In Simulink, for delaying a signal, the "delay" block (Figure 9.3) is available. This block is equivalent to a shift register FIFO in normal digital circuits but with the clock (at simulation frequency) always enabled.

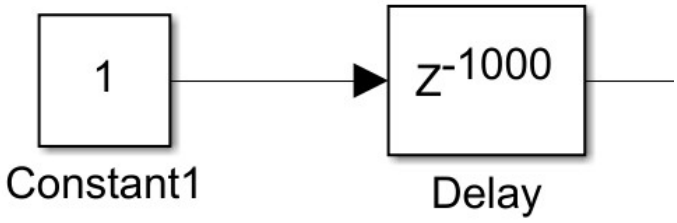

Figure 9.3: Delayed enabled signal using the delay block (1000 memory cells)

Choosing the delay value, a consequent number of registers is instantiated, and since sometimes it is required to have enable signals happening after more than 1000 samples, memory is easily filled in an inefficient way.

It was noticed that the cross compiler and Simulink do not check if memory is filled through these blocks, and this might result in a software vulnerability where some used memory parts are overwritten. As a result, it was noticed that the drone wasn't flying correctly when one of these long delay blocks was filled.

The found solution is shown in Figure 9.4, where a free-running counter is used. At the beginning (when the enable signal that needs to be delayed (It must be shaped as a pulse)), the counter's value is saved, and it is subtracted from the current value. When the difference

becomes higher than a threshold, the signal is activated. This solution is using 1 memory cell instead of one per sample. A simplified version is visible in Figure 9.5.

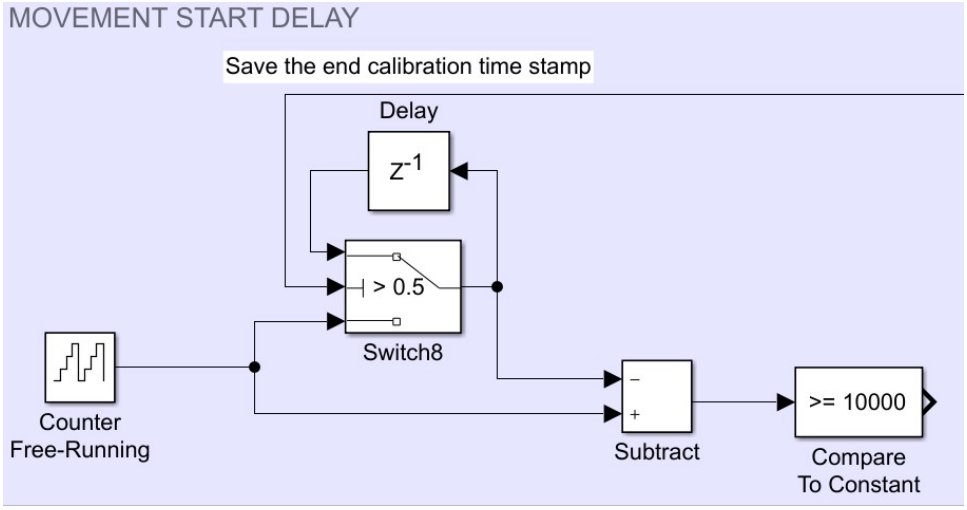

Figure 9.4: Delayed enable signal generic

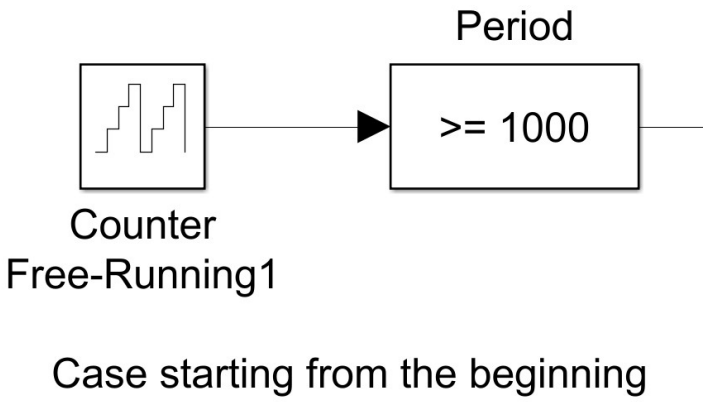

Figure 9.5: Delayed enable signal (from start time)

Thus, for better memory optimization, all long delay blocks used in the reused part of previous work [20] that was utilized were converted to this better version.

CHAPTER 10

CONCLUSION AND FUTURE DEVELOPEMENTS

The work done in this thesis resulted in a big step forward in the drone swarms research.

10.1 Main Found Results

As well visible by the experiments, it was possible to make a drone able to follow another one by only sensing its blade's wind. However, it was not possible to perform a continuative detection, but multiple instant detections had to be performed to estimate the leader's speed.

Success in this aspect put the basis on a completely new communication protocol: detected speed was used as a received signal, and with a correctly set up decoder and codification, it was possible to receive commands.

10.2 Systems Fault Tolerance Application

The developed communication algorithm is a new method for dealing with communication faults in a drone swarm: in case of disturbed communications or damaged devices, it will still be possible to give orders to such damaged elements. As an example, some commands can be given such that the damaged vehicle is steered out of a dangerous place and be safely recovered.

10.3 Secondary result

In this thesis, it was tried to design a brand-new altitude estimator to avoid sonar beam disturbances. However, the low computational power and the increasing complexity of the algorithm demonstrated that the best solution in terms of time spent is to use optical sensors

as they are about the same price and weight as ultrasonic sensors but don't suffer from the same problems.

10.4 Future Developments and applications

This algorithm can be improved if a drone with a down-facing laser is available. This is an example of how the leader might detect the follower and set up a bi-directional communication protocol. Another idea is to implement the "become leader" command for such bidirectional communication, even if it is probably less reliable.

APPENDICES

Appendix A

MATLAB SCRIPTS

A.1 Image Processing System (Optimized), Matlab Codes

A.1.1 Black Binarization (Code generated by the Color Thresholder app)

```
function BW  = createMask(R, G, B)
%  createMask  Threshold RGB image using auto-generated code from
    colorThresholder app.
%  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB
%  using auto-generated code from the colorThresholder app.
%  The colorspace and range for each channel of the colorspace
%  were set within the app. The segmentation mask is returned
%  in BW, and a composite of the mask and original RGB images
%  is returned in maskedRGBImage.

% Auto-generated by colorThresholder app on 06-May-2024
% (and modified since it was not working)
%------------------------------------------------------

% Define thresholds for channel 1 based on histogram settings
```

Appendix A (continued)

```
channel1Min = 210.000;

channel1Max = 255.000;


% Define thresholds for channel 2 based on histogram settings

channel2Min = 210.000;

channel2Max = 255.000;


% Define thresholds for channel 3 based on histogram settings

channel3Min = 210.000;

channel3Max = 255.000;


% Create mask based on chosen histogram thresholds

sliderBW = (R(:,:) >= channel1Min ) & ...

    (R(:,:) <= channel1Max) & ...

    (G(:,:) >= channel2Min ) & (G(:,:) <= channel2Max) & ...

    (B(:,:) >= channel3Min ) & (B(:,:) <= channel3Max);

BW = sliderBW;

end
```

Appendix A (continued)

A.1.2 Track Center Search and Yaw measure

```
function [Y_mov,Yaw] = fcn(edgedBW)


% Input Image Dimensions
Width = 160;
Height = 120;


White_border = zeros(1, Width); %The matrix composed of only
   vertical lines can be represented as a single vector (that
   represent one of its horizontal lines)


sumColumns_top = sum(edgedBW(1:Height/2,:),1);
sumColumns_bot = sum(edgedBW(Height/2:Height,:) ,1);
%sumColumns = sumColumns_top + sumColumns_bot;


left_top = 0;  %sum(sumColumns_top(1:Width/2));
left_bot = 0;  %sum(sumColumns_bot(1:Width/2));
right_top = 0; %sum(sumColumns_top(Width/2:end));
right_bot = 0; %sum(sumColumns_bot(Width/2:end));


%Counting the number of significatively white hal-columns
```

Appendix A (continued)

```
for i=1:Width/2
    if sumColumns_top(i) > 30
        left_top = left_top + sumColumns_top(i);
    end
    if sumColumns_bot(i) > 30
        left_bot = left_bot + sumColumns_bot(i);
    end
end



for i=Width/2:Width
    if sumColumns_top(i) > 30
        right_top = right_top + sumColumns_top(i);
    end
    if sumColumns_bot(i) > 30
        right_bot = right_bot + sumColumns_bot(i);
    end
end

dY_top = right_top - left_top;
dY_bot = right_bot - left_bot;
```

Appendix A (continued)

```
Y_mov = (dY_top + dY_bot)/2;

Yaw = dY_top-dY_bot;
```

A.1.3 Vertical lines detection (Vertical_Positions)

```
function Vert_lines = fcn(edgedBW)
%Tuning parameters
Vert_line_detect = 5;
%column_width = 1;


% Output Image Dimensions
Width = 160;
%Height = 1;


Vert_lines = zeros(1, Width); %The matrix composed of only
   vertical lines can be represented as a single vector (that
   represent one of its horizontal lines)


sumColumns = sum(edgedBW,1);


for i=1:Width-1    %(Width-column_width)
```

Appendix A (continued)

```
    %if abs(sumColumns(i)-sumColumns(i+column_width)) >
       Vert_line_detect1
    if abs(sumColumns(i)-sumColumns(i+1)) > Vert_line_detect %If
       there is a jump on the object pixel from a column to the
       next one, it means that there is a border
        Vert_lines(i) = 1;
    end
end
```

A.1.4 Z estimation Image (Without_vertical_lines)

```
function Z_img = Distance_estimation_inst(edgedBW)

% Input Image Dimensions
Width = 160;

%Distance average on the horizontal lines (Vertical lines
   measures are not needed)
summation = 0;

previndex = find(edgedBW ~= 0, 1); %This function will result in
   a 1 dimensional vector (It always gives a vector)
```

Appendix A (continued)

```
count = 0;


if isempty(previndex) == 0
    previndex = previndex(1); %even if this will be a vector of a
        single element, for better software  robustness I
       redefine it as a scalar


    if previndex < Width
        while edgedBW(previndex+1)~=0 && (previndex+1)<Width %
           This is to prevent consecutive ones (0 distance)
            previndex = previndex+1;
        end
    end
    index = find(edgedBW((previndex+1):Width) ~= 0, 1); %Here I
       obtain already the distance with the next edge
    %I'm searching on a vector of dimension "Width-previndex"


    while isempty(index) == 0 %While some elements at 1 are found
        index = index(1); %even if this will be a vector of a
           single element, for better software  robustness I
           redefine it as a scalar
```

Appendix A (continued)

```
        %Avoid small distances (width of the red tape on the
           sides) Avoid large distance (from the red tape to the
           end of the cloth and from the red tape to the thin
           track)
        if index > 7 || index < 35 %it was 15 to 90 before
            count = count +1;
            summation = summation + index;
        %else
            %do nothing (small distances don't count)
        end
        previndex = previndex+index;
        if previndex < Width
            while edgedBW(previndex+1)~=0 && (previndex+1)<Width
               %Removing cases with zero distances
                previndex = previndex+1;
            end
        end
        index = find(edgedBW((previndex+1):end) ~= 0, 1);
    end
end
```

Appendix A (continued)

```
if count == 0
    Z_img = 0;
else
    avg_distY = summation/count;

    %As the altitude is inversely proportional to the camera
       distances (higher altitude means that ground objects looks
        smaller) here it is the conversion:
    if avg_distY == 0
        Z_img = 0;
    else
        Z_img = 1/avg_distY;
    end
end
```

A.2 Path Planning Logic

A.2.1 Yaw Precise Control

```
function yaw_inc  = Yaw_Precise_Ctrl(u,Trck_ena)
```

```
%%%% Tuning parameters

Gain_yaw_dir = 0.000001;


if Trck_ena == 1

    yaw_inc = u*Gain_yaw_dir;


else

    yaw_inc = 0;


end

end
```

A.3 Data Processing Scripts

A.3.1 Image Calibration

```
clc


skip_beginning=16;

Z_ref = 0.75; %Fixed drone altitude during the measure


image_dirty = rt_estimatedStates.signals.values(skip_beginning:

   end, ??); %Insert the correct index here
```

Appendix A (continued)

```
pixel_count = 1./image_dirty;

t = rt_estimatedStates.time(skip_beginning:end);


figure;

plot(t , image_dirty, 'blue');   %Img dirty (altitude measure to
   be rescaled)

title('Measure of the altitude');

avg_img_ref = sum(image_dirty)/length(image_dirty); %This is the
   value 1/P_ref


figure;

plot(t , pixel_count, 'blue');   %Pixel count dirty

title('Average pixel count');

avg_pxl_ref = sum(pixel_count)/length(pixel_count);  %This is
   P_ref



Rescaling_constant = Z_ref/avg_img_ref;

disp(Rescaling_constant);

%Now I can copy this constant to the actual drone
```

```
figure;
plot(t , image_dirty.*Rescaling_constant, 'blue');   %Img dirty (
   altitude measure to be rescaled)
title('Actual altitude IMG');
```

A.3.2 Rotors Biases Calculator

```
% Rotors


FL_AVG = mean(rt_yout.signals(1).values(:, 1));
FR_AVG = -mean(rt_yout.signals(1).values(:, 2));
RR_AVG = mean(rt_yout.signals(1).values(:, 3));
RL_AVG = -mean(rt_yout.signals(1).values(:, 4));


AVG_Thrust = (FL_AVG + FR_AVG + RR_AVG + RL_AVG)/4;


biases = [(AVG_Thrust/FL_AVG), (AVG_Thrust/FR_AVG), (AVG_Thrust/
   RR_AVG), (AVG_Thrust/RL_AVG)]; %Calibrate Thrust


%Choose if to calibrate the thrust or the angular moment
% 0 - Calibrate the thrust (all rotors same thrust) (suggested)
```

Appendix A (continued)

```
% 1 - Calibrate the angular moment (total moment = 0)
mode = 0;

if mode == 1: %Calibrate Moment
    biases(1) = biases(1)*2/(biases(1) + biases(3));
    biases(2) = biases(2)*2/(biases(2) + biases(4));    %For a
       null total angular moment, the gains
    biases(3) = biases(3)*2/(biases(1) + biases(3));    %are
       normalized such that each diagonal sum is 2
    biases(4) = biases(4)*2/(biases(2) + biases(4));    %The
       major cause of this calibration stands in the blades, not
       the engines
end

disp(['Biases: [', num2str(biases),']']);
```

CITED LITERATURE

1. Mellace, M.: Colomba di archita da taranto, il primo drone della storia nel 350 a.c. https://www.madeintaranto.org/colomba-di-archita/ [Online; accessed 06/19/2024].

2. Museo Nazionale della Scienza e della Tecnologia Leonardo da Vinci, Milano: Macchina volante - vite aerea. http://www.museoscienza.it/dipartimenti/catalogo_collezioni/scheda_oggetto.asp?idk_in=ST070-00001arg=Modelli%20leonardeschi [Online; accessed 06/19/2024].

3. Icaros, Aerial Intelligence: Agriculture. https://icarosgeospatial.com/agriculture/ [Online; accessed 04/01/2024].

4. Mistretta, C.: Coming soon: Human-drone teams working on construction sites. University of Florida, College of Design Construction and Planning. https://dcp.ufl.edu/news/constructiondrones/ [Online; accessed 06/20/2024].

5. DIREC, Digital Research Center: Drone swarms must respond fast in case of natural disasters and drowning accidents. https://direc.dk/drone-swarms-must-respond-fast-in-case-of-natural-disasters-and-drowning-accidents/ [Online; accessed 06/20/2024].

6. Sekkappan, C.: Catch a spectacular dragon-themed drone light show at marina bay waterfront this february. TimeOut, 2024. https://www.timeout.com/singapore/news/catch-a-spectacular-dragon-themed-drone-light-show-at-marina-bay-waterfront-this-february-012624 [Online; accessed 04/02/2024].

7. Petersen, R.: Rethinking last-mile logistics: Deploying swarms of drones with self-driving trucks. flexport, 2016. https://www.flexport.com/blog/deploying-drones-self-driving-trucks/ [Online; accessed 04/02/2024].

8. EPSON: Gyro sensors - how they work and what's ahead. https://www5.epsondevice.com/en/information/technical_info/gyro/ [Online; accessed 06/23/2024].

9. Thakkar, R. R.: Electrical equivalent circuit models of lithium-ion battery. IntechOpen, 2021. https://www.intechopen.com/chapters/78501 [Online; accessed 06/23/2024].

CITED LITERATURE (continued)

10. Collins, D.: What are coreless dc motors? MOTION CONTROL TIPS, 2024. https://www.motioncontroltips.com/what-are-coreless-dc-motors/ [Online; accessed 06/18/2024].

11. Fiorillo, F.: Magnetic materials for electrical applications: a review. Technical report, iNRiM (Istituto Nazionale di Ricerca Metrologica), 2010.

12. Parrot: Parrot Mambo Fly. . https://www.parrot.com/us/drones/parr mambo-fly [Online; accessed 01/25/20], in Paolo Ceppi Model-based Design of a Line-tracking Algorithm for a Low-cost Mini Drone through Vision-based Control.

13. HASAKEE: Q9s Drones for Kids. https://www.hasakee.com/products/q9s-drones-for-kids-rc-drone-with-altitude- hold-and-headless-mode-quadcopter-with-blue-green-light-propeller-full-protect-2-batteries- and-remote-control-easy-to-fly-kids-gifts-toys-for-boys-and-girls [Online; accessed 06/18/2024].

14. Radians: Radians MD1-20-13 MetSafe MD1 Safety Glasses - Blue Frame - Clear IQuity Anti-Fog Lens. https://www.fullsource.com/radians-md1-20-13/ [Online; accessed 06/23/2024].

15. Techkey: Bluetooth Adapter for PC USB Bluetooth Dongle 4.0 EDR Receiver. https://www.mytechkey.com/collections/frontpage/products/bluetooth-adapter-for-pc-usb-bluetooth-dongle-4-0-edr-receiver-techkey-wireless-transfer-for-stereo-headphones-laptop-windows-10-8-1-8-7-raspberry-pi-linux-compatible [Online; accessed 06/23/2024].

16. HAINSWORTH, F. R.: PRECISION AND DYNAMICS OF POSITIONING BY CANADA GEESE FLYING IN FORMATION. The Company of Biologists Limited 1987, 1986.

17. Dalamagkidis, K.: On Integrating Unmanned Aircraft Systems into the National Airspace System. Springer Science+Business Media B.V., 2012.

18. MAJ Andrew W. Sanders, U. A.: Drone Swarms. Master's thesis, School of Advanced Military Studies United States Army Command and General Staff College Fort Leavenworth, Kansas, 2017.

19. Canis, B.: Unmanned aircraft systems (uas): Commercial outlook for a new industry. Congressional Research Service, 2015.

CITED LITERATURE (continued)

20. Ceppi, P.: Model-based Design of a Line-tracking Algorithm for a Low-cost Mini Drone through Vision-based Control. Master's thesis, University of Illinois at Chicago and Politecnico di Torino, 2020.

21. Orsag, M., Korpela, C., Oh, P., and Bogdan, S.: Aerial Manipulation. Springer, 2018.

22. Quaternion. Wikipedia, https://en.wikipedia.org/wiki/Quaternion [Online; accessed 06/14/2024].

23. rotmat. https://www.mathworks.com/help/nav/ref/quaternion.rotmat.html [Online; accessed 06/14/2024].

24. Loizou, S. G. and Constantinou, N.: Implicit communication in multi-robot systems with limited sensing capabilities. 60th IEEE Conference on Decision and Control, 2021.

25. MathWorks: MathWorks Minidrone Competition. https://www.mathworks.com/content/dam/mathw dot-com/academia/student-competitions/minidrone-competition/mathworks-minidrone-competition-guidelines.pdf [Online; accessed 06/14/2024].

26. Smoot, J.: Understanding ultrasonic sensors. Digikey, 2021. https://www.digikey.com/en/articles/understanding-ultrasonic-sensors [Online; accessed 06/23/2024.

27. Upadhyay, J.: How do accelerometers and gyroscopes work? CircuitBread, 2022. https://www.circuitbread.com/ee-faq/how-do-accelerometers-and-gyroscopes-work [Online; accessed 06/23/2024].

28. Leishman, J. G.: INTRODUCTION TO AEROSPACE FLIGHT VEHICLES. EMBRY-RIDDLE Aeronautical University, 2023. DOI https://doi.org/10.15394/eaglepub.2022.

VITA

NAME	Leonardo Cerruti
EDUCATION	
	Master of Science in "Electrical and Computer Engineering, University of Illinois at Chicago, Jul 2024, USA
	Specialization Master's Degree in " Embedded Systems Electronic Engineering ", Jul 2024, Polytechnic of Turin, Italy
	Bachelor's Degree in Electrical and Electronic Engineering, Oct 2022, Polytechnic of Turin, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2022 - IELTS examination (7/9)
	A.Y. 2023/24 One Year of study abroad in Chicago, Illinois
	A.Y. 2022/23. Lessons and exams attended exclusively in English

WORK EXPERIENCE AND PROJECTS	
2023	Design procedure of a full DLX processor
	Starting from the classic DLX pipeline structure decide the desired instruction set to be implemented, modify the datapath such that it can accomplish the correct execution. Write all the structure in VHDL together with the control unit and debug it with testbenches. Now, complete the synthesis, optimization, and physical design.
2023	Design of a Linux driver

VITA (continued)

	Design procedure of a Linux driver for managing the SHA256 crypto core that must be set up on an FPGA via system calls. Using a special board with an integrated FPGA and able to run Peta Linux set up the system and the driver. After that, write generic C codes able to perform the SHA256 operation by accessing the crypto core via normal system calls.
2022	Measurement analysis of magnetic materials characteristics (INRiM) Using an Epstein circuit, measure the hysteresis cycle of different magnetic materials at different temperatures and magnetic field intensities. As the measuring circuit was running high current quantities, especially when a high magnetic field was needed, overheating was an issue. Since to ensure low measurement uncertainties the circuit couldn't be modified, design an external circuit that measures the temperature and activate an alarm in case of close overheating.

MISCELLANEOUS SKILLS

Advanced level	Playing Violin