Institut Polytechnique
des Sciences Avancées

IPSA Paris

UAV CONTROLLED BY HAND GESTURES

YONDU PROJECT

AU513 - Prototypage rapide
School Year 2022-2023

Submitted by:
Guillaume ARNAUD, Pierre HOUSSEAUX, Samuel HUITELEC,
Malak KOJOK, Sebastien LEBEGUE, Marco TOMMASI,
Robinson VAITILINGOM

Class: AERO5 - SAA

Professors:
M. Achour DEBIANE | M. Yamine SELLAMI
achour.debiane@ipsa.fr | yamine.sellami@ipsa.fr

January 17, 2023

Contents

Introduction

Since World War One, remote controlled flying machine have enjoyed a remarkable expansion from military field with prototypes torpedo, to civilian applications with ground medical assistance UAVs. In essence, drones are specialized flying robots that carry out tasks like taking pictures, recording films, and gathering multi-modal data from their surroundings. Multi-rotor drones can operate in places where humans cannot, collect multi-modal data, and intervene when necessary thanks to their maneuverability, adaptability and small size. They have particularly become a useful and playful tool for those who wish to step back, enjoy landscapes and feel as free as a bird. What has made a big difference with the variety of existing drones, apart from their shapes and the technologies used, is their ability to be controlled in different ways.

Whether they are controlled through joysticks, mobile apps, artificial intelligence, or embedded computers, drones offer today a wide and imagination-free remote control spectral. Drones control is usually restricted by the range of electromagnetic radiation and is sensitive to interference noise, to name a few main challenges with these systems. We look at using computer vision approaches to provide an easy manner of agent-less communication between a drone and its operator.

The idea with our project called **Yondu Project** would be to adapt and imagine a way to control a drone by hand gestures. This will make it easier to control, due to its user and all ages - friendly interface, and will allow the drones market to reach out to a wider and more varied audience.

For instance, it should become more common among both younger and older generations, and try to propose an simple and fun alternative/interface to start getting them familiar with drones. improve their daily life affordable for younger generation and people with learning difficulties.

PART I

Systems Requirements

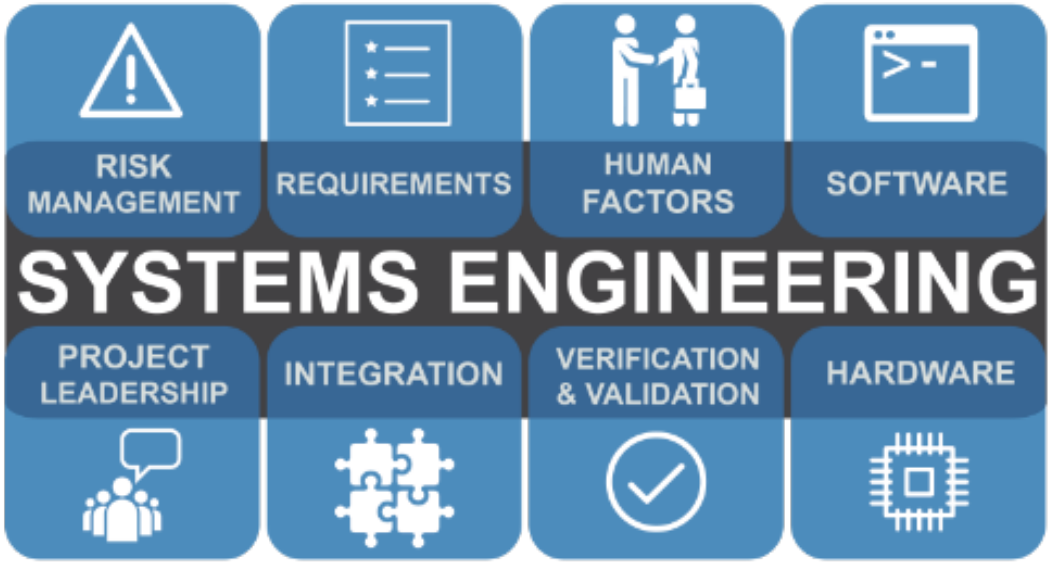

Figure 1: Systems Engineering Components

This part will define our project as we go through the activities to be performed.

Systems requirements

1 Functional analysis

In this part, we will focus on the functional analysis of this project. The idea is to respond to the question : what will our system do ? What features should it implement and why ?
In this project, we want to control an unmaned aerial vehicle (UAV) using hand signs. We decomposed all the necessary functions of our system as followed :

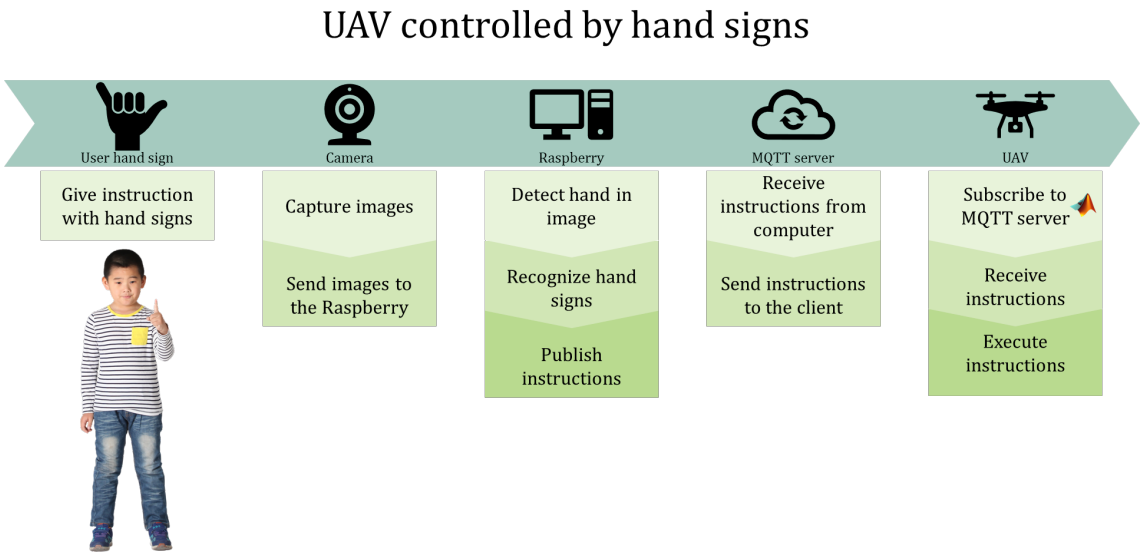

Figure 2: Functional analysis of Yondu project

The process is divided in five distinct tasks. The first corresponds to the hand signs made by the user that will initiate the process. The user designates the person who wants to control the drone and has the authorization to do it. These hand signs will be the instructions we want to transmit to the drone. The setting of those instructions will be seen later.

Then, we decided to capture the hand signs with a camera. Indeed, it is the most flexible way of capturing them with the minimum of equipment. In our case, we decided that the camera will be connected to a Raspberry Pi. It is a very light weight computer which can be brought almost everywhere. So the camera will capture the images and the Raspberry Pi will retrieve them.

The third step will only focus on the image processing that the Raspberry will be doing. From the retrieved images, it has to detect the presence of the user hand, then recognize the gesture made and associate it to an instruction to send to the drone. This instruction will then be sent to a MQTT server.

The fourth step concerns the MQTT server, also called MQTT broker. We decided to use MQTT protocol to make the communication between the drone and the user. MQTT is a communication protocol which follows the Publisher/Subscriber design pat-

tern. Users, or actors, can publish and subscribe to topics which are kept by the MQTT broker. The MQTT broker is in charge of transmitting the information published on a topic to the subscribers of this topic. In our case, the actors are the Raspberry Pi, which published instructions, and the drone or the computer which will receive the instructions. But why using MQTT in our case ? The idea is that, as long as the drone is connect to internet, the user can be anywhere in the world. It does not have to be next to the drone to control it.

The ultimate step is about the drone that the user tries to control. It will receive the instructions from the MQTT server, process them and the drone controller will compute the commands which will make the drone execute them. To receive the instructions, the drone must be connected to internet.

2 Technical specifications

In this section, we want to give a better description of the technical aspect our project will have to be compliant with.

2.1 Instruction processing

First, hand signs must be retrieved from an image, then processed in order to extract the pilot's instruction.

This section includes all the elements before the MQTT server in the functional analysis done in the previous part.

The following flow chart has been made in order to describe the strategy which will be implemented :

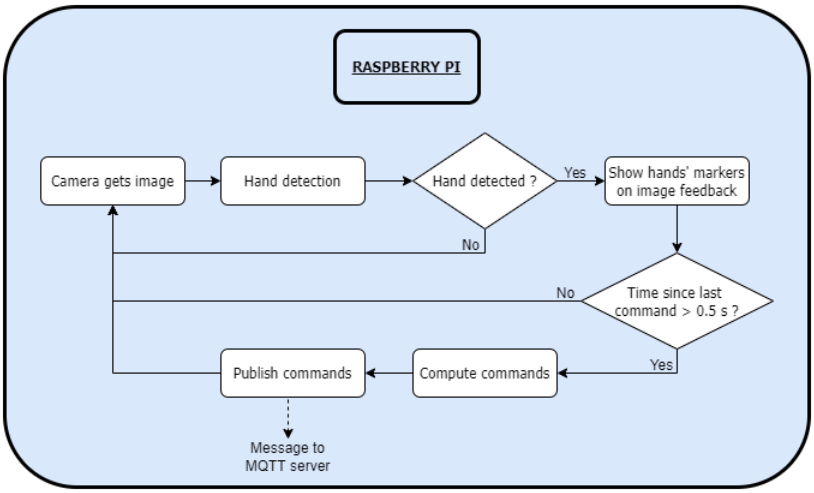

Figure 3: flow chart of the code running on the Raspberry Pi

First, the Raspberry Pi or the computer running the code will have to capture an image with its camera. Then, a hand detection will have to be performed. If no hand

is detected, we capture another image. Otherwise, we show the hand markers and if the duration since the last command send is greater than 0.5 second, then we recognize the hand sign and associate it to its instruction, creating a command which will be send via MQTT protocol to the broker.

2.2 Instruction transmission

In this section, we will discuss how the instruction will be sent from a Raspberry Pi or a computer to the drone.
This will be done through the use of a MQTT server, or MQTT broker. Each agent will have to connect to this server, so an internet connection is required. With MQTT protocol, each agent is a publisher and/or a subscriber. In our case, the Raspberry Pi or computer which is sending the instruction is a publisher and the drone is a subscriber which has subscribed to the topics on which instructions will be sent.

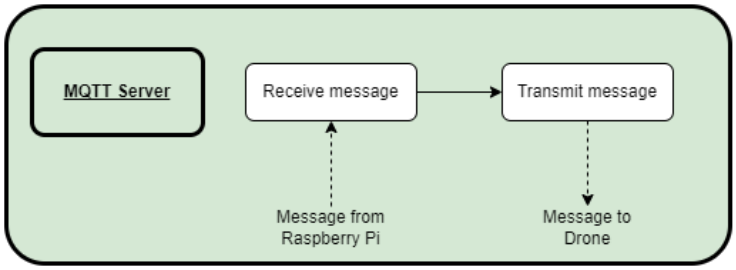

Figure 4: MQTT server flow chart diagram

We do not need to provide more information about the transmission as we do not create the server by ourselves, it is already existing.
However, we can provide the benefits of using MQTT communication over another communication protocol :

- **Scalability :** the publish/subscribe model scales well in a power-efficient way.
- **Reliability :** MQTT uses mechanisms such as quality of service tokens to ensure the delivery of data.
- **Packet agnostic :** any type of data can be transported
- **Decoupled Design :** decouple the device and the subscribing server
- **Low energy consumption :** MQTT is designed to work on low-power devices such as sensors and actuators
- **Low bandwidth consumption :** MQTT uses very little bandwidth as it only transmits data that has been changed
- **Fast response time :** MQTT is designed to transmit data in real-time, so that applications can quickly react to data changes.
- **Portable and flexible :** easy to use, well-suited for remote sensing and control

2.3 Instruction execution

In this section, we want to describe how the drone will execute the instruction it will receive from the MQTT server. To do so, we draw the following flow chart:

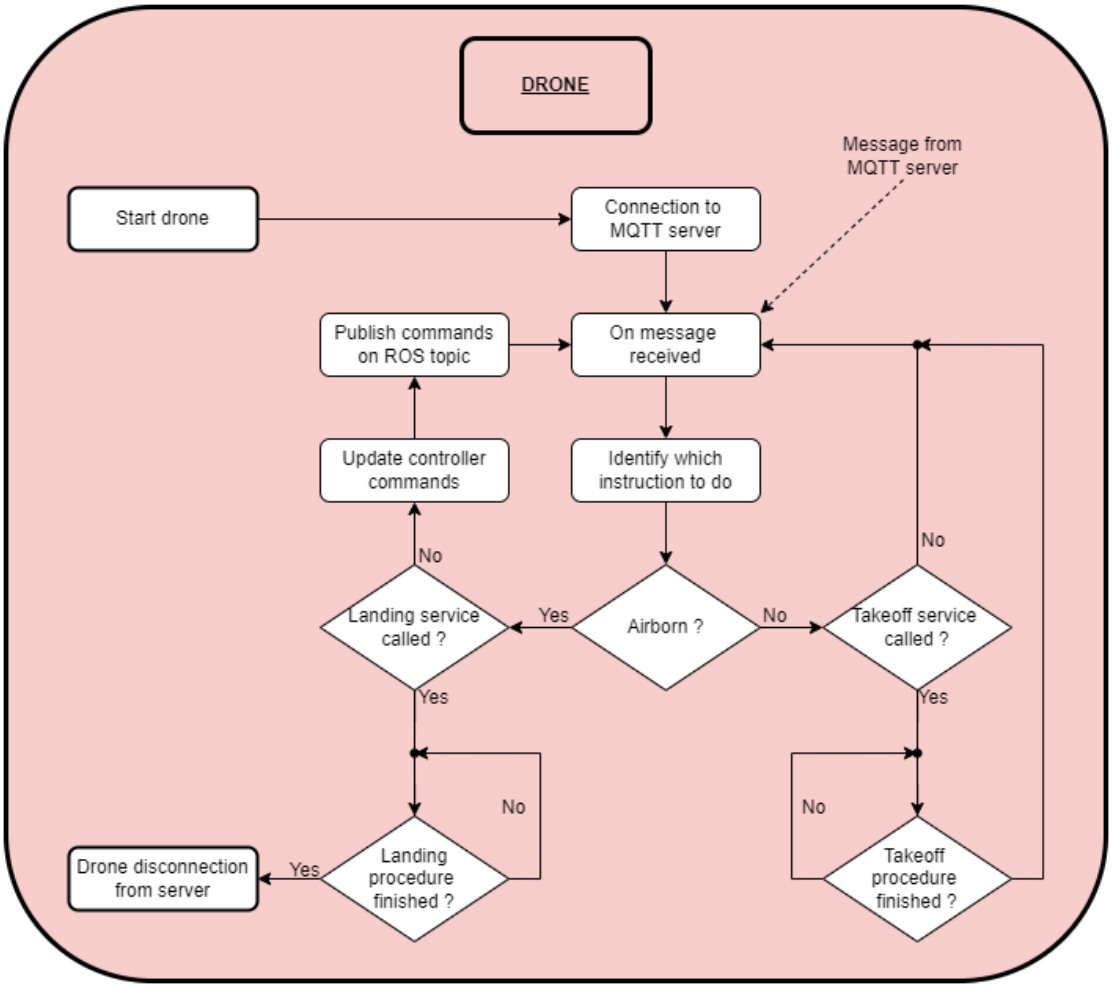

Figure 5: Drone's flow chart

When starting, the drone must connect and subscribe to the MQTT server and the commands topics. Then, it waits for messages, process them and execute the received instructions.
We must be sure that the instruction can be executed. For instance, it is impossible to ask the drone to move forward if it is not in the air. We need to ask him to takeoff first. Finally, if a landing instruction is received, the program should land the drone then stop itself.

PART II

System Design

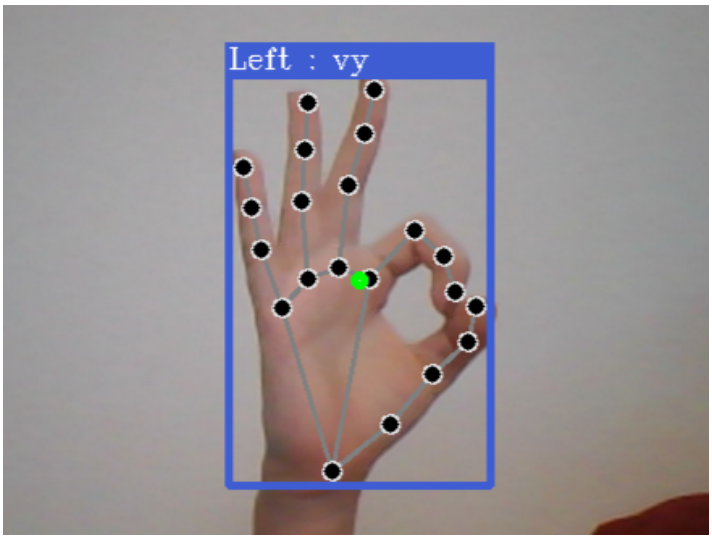

Figure 6: Hand gesture recognition

This part focuses on the implementation of the requirements described in the previous section.

System Design

In this part, we will present all the implementations made for this project. It will be divided in 3 parts :

- The hand signs recognition
- The MATLAB/Simulink drone simulation
- The ROS/Gazebo simulation

You can find all the codes developed in this section on RobKenobi's GitHub.

1 Hand sign instructions

In this section, we will describe how we manage to detect a hand in a video feed and interpret its sign as an instruction, the whole as a real time process.

1.1 Hand detection

First, we need to detect a hand in the picture. To do so, we used the MediaPipe library with its Hands solution API.
We created our own python module (*handtracking/GestureRecognition/HandDetector.py*). This file contains a class *HandDetector* which has the following methods:

- *get_result()* : returns the hand's markers if a detection has been made, *None* otherwise.
- *hands_detection()* : detects hands in a given image, returns *True* if a detection has been made.

1.2 Hand's signs detection

Now that we are able to detect hands in an image with *HandDetector*, we need to use the hand's markers it returns to determine the pilot's hand gesture.

To do so, we created *HandProcessing* class (*handtracking/GestureRecognition/HandProcessing.py*). It has the following methods:

- *find_handedness()* : returns the handedness of a given detected hand.
- *find_position_on_image()* : returns the position of each hand's markers in pixel in the image frame (origin : left top corner).

- *find_gesture()* : based on the handedness and the hand's marker positions, it determines the hand gesture using artificial intelligence (AI) models (see next subsection).
- *create_hand_commands()* : creates a *HandCommand* object for each hand detected.

The *HandCommand* class is another class we defined. It stores information about a detected hand such as its handedness, the position of its markers in the hand frame and the image frame and the gesture detected. It also allows to convert the position of the hand's markers into numpy array as they are stored in a specific type by *MediaPipe* during detection.

1.3 Artificial intelligence models for hand gesture classification

To make *HandProcessing* objects work, we need an AI models for gesture recognition. These models are used in the *HandProcessing.find_gesture()* method.

Gesture recognition is not a very hard machine learning problem. In our case, it is a classification problem : from input values, we want to provide a label. In our case the inputs are the hand's markers and the output is the gesture the hand is doing.

1.3.1 Dataset creation

The first step before training an AI is to get a training dataset. We did not have it so we had to create it.

To do so, we created the *handtracking/ModelBuilder/datasetBuilder.py* script. This script is using the *HandDetector* and *HandProcessing* we defined in the *handtracking/GestureRecognition* module to get the hand's markers. The goal of this script is to build a CSV file which contains the normalized hand's markers with the associated gesture. The normalization is made using the wrist markers as reference, so the position of the hand in the image and the hand size do not have any influence on the gesture to predict.

To capture hand's markers and associated label, we simply had to show the gesture to the camera and press the key which indicates which gesture has been made. Here are the key associations we defined :

- **<h>** or **<enter>** : show key associations
- **<ESC>** : exit the program
- **<o>** : gesture is open hand
- **<c>** : gesture is closed hand
- **<i>** : gesture is index up
- **<p>** : gesture is thumb up
- **<v>** : gesture is index and middle finger up (V sign)
- **<k>** : gesture is index and thumb joined (OK sign)

When a gesture key is pressed, the hand's markers are saved with the label corresponding to the key pressed. We had to create two datasets : one for the left hand, the other one for the right hand. Therefore, the saving process is done on the dataset corresponding to the handedness of the detected hand.

With this script, we were able to create two datasets of **1800 examples for each hand**, so a **total of 3600 examples**. It represents 300 examples for each hand for each sign. To show these statistics, you can run *handtracking/ModelBuilder/datasetStatistics.py*.

1.3.2 Models creation

Now that we have our datasets on which we will be able to train an AI, we need to define which AI model we are going to use. In our case, we opted for a Classification And Regression Tree (CART) model. CART models are binary classifications trees. Hence, they are very light and fast which is perfect for real time embedded hand gesture recognition. Moreover, they are trained very quickly unlike a neural network which can take some time.

The *handtracking/ModelBuilder/CARTClassifier.py* allows to create the CART models for the right and left hands. These models are saved in the *handtracking/GestureRecognition* as they are used by the *HandProcessing.py* file. They are the CART_left.sav and CART_right.sav files.

The *CARTClassifier.py* script starts by loading the datasets we created like explained in the previous section. Then it creates a training and a test partition. The complete dataset is shuffled before partitioning it. We use 20% of the available data for the test set. Then, the model is trained with the training data (80% of the available data). After the training, it is evaluated wit the test dataset. Then it is possible to chose keeping the newly created model or not. This can be useful if the validation test is not good enough. In our case, the test dataset is predicted with an accuracy above 99%, which is a very good result.

1.4 Hand sign instructions

Now, we are able to detect a hand and associate a sign to the detected hand. The only remaining element is to assign an instruction to the hand sign. We defined the following parameters which will be used for controlling the drone :

- **vx** : the longitudinal speed
- **vy** : the lateral speed
- **vz** : the vertical speed
- **v_yaw** : the rotation speed around the z axis.
- **landing** : request drone to land
- **takeoff** : request drone to takeoff

- **stop** : request drone to stop moving

Each of these instruction will be a topic on which the computer processing the command will publish a value which will be the instruction associated to the parameters. This value will be 0, 1 or -1 and will be determined by the hand position in the picture. Indeed, we divided the image feedback in 3 vertically stacked area with the following associations:

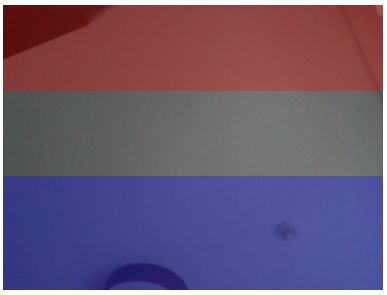

Figure 7: Camera feedback

- Top area (red) : 1
- Center area : 0
- Bottom area (blue) : -1

Finally, we have to make the association between the hand signs and the parameters to play on :

Parameters	**Hand signs**
vx	Left hand index up or right hand Ok sign
vy	Right hand index up or left hand Ok sign
vz	Left hand V sign
v_yaw	Right hand V sign
landing	Left or right hand thumb up
takeoff	Left or right hand opened
stop	Left or right hand closed

For instance, if an opened hand is detected in the red area, it will publish 1 on the topic *takeoff*. The interpretation of the instruction will be made by the drone itself.

2 MATLAB/Simulink simulations

In this part we will demonstrate all *MATLAB* simulation that have been made until now. Furthermore we will discuss of their implementation and the link they will have with other parts of the project.

2.1 Simulink implementation

The content of this section is divided in two main parts: the first one, concerning on the presentation of the used physical model, and the second one showing the Simulink implementation.

2.1.1 Physical model

The problem statement is presented in the following image, showing the adopted configuration, along with all the state variables.

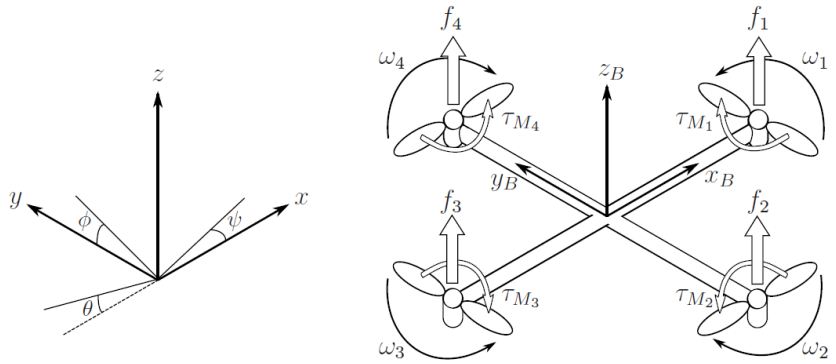

Figure 8: Problem Statement

Some reference frame will be used, in particular:

$$\xi = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad \eta = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \qquad q = \begin{bmatrix} \xi \\ \eta \end{bmatrix}$$

Here, ξ represents the inertial linear positions frame (x, y and z coordinates), while η represents the inertial angular positions frame (Euler angles). q is the composition of ξ and η. Linear and angular velocities in the body frame are expressed with the two following, where (p, q, r) are the pitch, roll and yaw rate.

$$V_B = \begin{bmatrix} v_{x,B} \\ v_{y,B} \\ v_{z,B} \end{bmatrix} \qquad \nu = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Other useful matrices are the inertia matrix (I), the rotation matrix from the body frame to the inertial one (R) the Thrust vector (T^B), the Torque vector (τ_b), and I_M the inertia moment of the rotor.

$$\mathrm{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

$$\mathrm{R} = \begin{bmatrix} cos(\theta)cos(\psi) & sin(\psi)sin(\theta)cos(\psi) - cos(\phi)sin(\psi) & cos(\phi)sin(\theta)cos(\psi) + sin(\phi)sin(\psi) \\ cos(\theta)sin(\psi) & sin(\psi)sin(\theta)sin(\psi) + cos(\phi)cos(\psi) & cos(\phi)sin(\theta)sin(\psi) - sin(\phi)cos(\psi) \\ -sin(\theta) & sin(\phi)cos(\theta) & cos(\phi)cos(\theta) \end{bmatrix}$$

$$\begin{cases} T = \sum_{i=1}^{4} f_i = k \sum_{i=1}^{4} \omega_i^2 \\ T^B = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \end{cases} \qquad \begin{cases} \tau_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} lk(-\omega_2^2 + \omega_4^2) \\ lk(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^{4} \tau_{Mi} \end{bmatrix} \\ \tau_{Mi} = I_M \dot{\omega}_i \end{cases}$$

2.1.2 Motion's Equations

The first step is to compute the angular velocities in the body reference frame, then convert them in the absolute reference frame angular velocities (and Euler angles) and, in the end, get the linear positions in the inertial reference frame, respectively:

$$I\dot{\nu} + \nu \times (I\nu) + \Gamma = \tau \tag{1}$$

$$\dot{\eta} = W_\eta^{-1} \nu \tag{2}$$

$$m\ddot{\xi} = G + RT_B \tag{3}$$

Where:

$$\Gamma = -I_r \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} (\omega_1 - \omega_2 + \omega_3 - \omega_4) \tag{4}$$

$$W_\eta^{-1} = \begin{bmatrix} 1 & sin(\phi)tan(\theta) & cos(\phi)tan(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & \frac{sin(\phi)}{cos(\theta)} & \frac{cos(\phi)}{cos(\theta)} \end{bmatrix} \qquad G = -g \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

2.1.3 Block Diagram and Controller Design

Here, the complete block diagram is shown, starting from the model itself. After it, the controller design is shown, consisting of 4 PIDs controller, one for the z coordinate, and the other three for the Euler angles. In this way, by applying some combinations, we are able to compute the needed forces (resulting in the total thrust) and torques (consequences of the imposed angles) to perform a particular manoeuvre.

The final motor's speeds equations are reported after the two plots With this, we are able to control all the degrees of freedom, by applying different inputs, described in the next section.

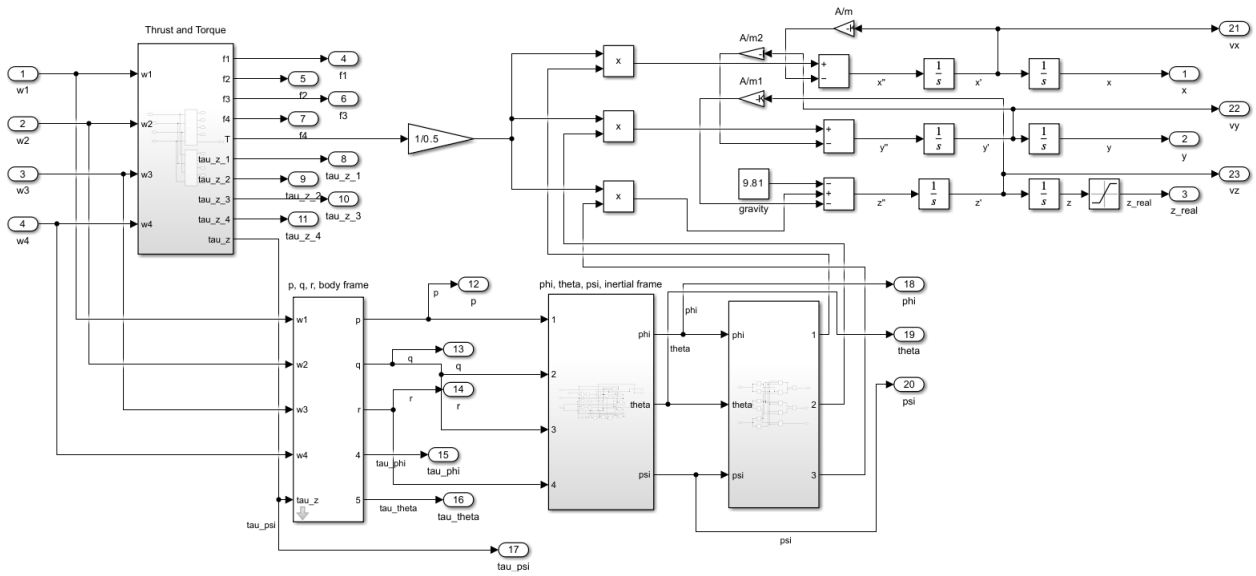

Figure 9: Model Block Diagram

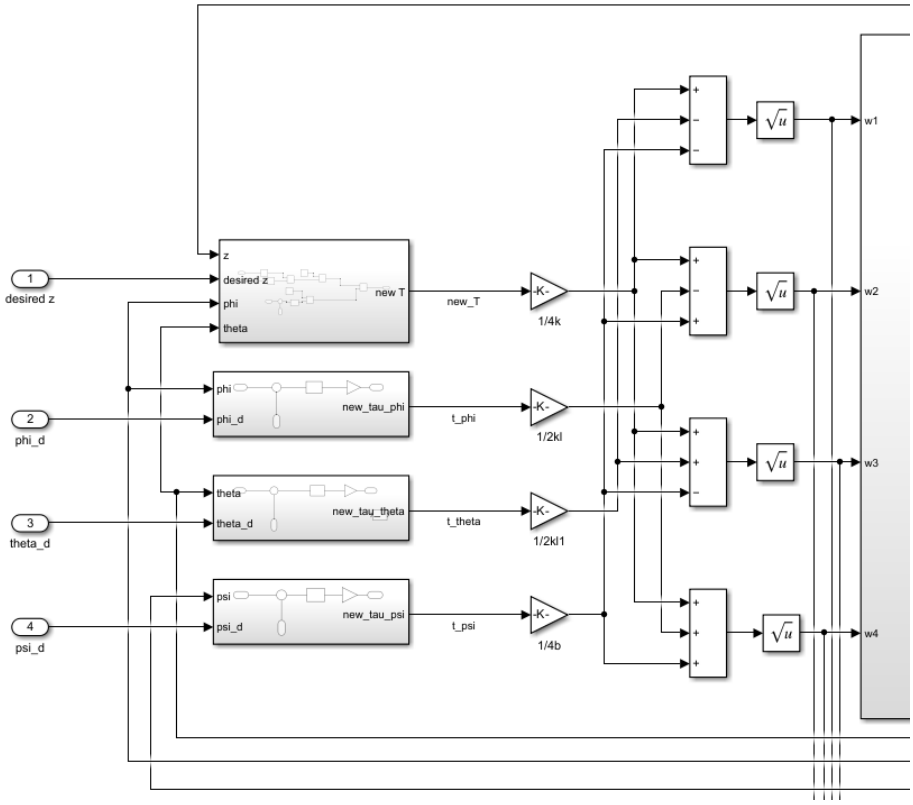

Figure 10: Controller Block Diagram

$$\begin{cases} \omega_1^2 = \dfrac{T}{4k} - \dfrac{\tau_\theta}{2kl} - \dfrac{\tau_\psi}{4b} \\ \omega_2^2 = \dfrac{T}{4k} - \dfrac{\tau_\phi}{2kl} + \dfrac{\tau_\psi}{4b} \\ \omega_3^2 = \dfrac{T}{4k} + \dfrac{\tau_\theta}{2kl} - \dfrac{\tau_\psi}{4b} \\ \omega_4^2 = \dfrac{T}{4k} + \dfrac{\tau_\phi}{2kl} + \dfrac{\tau_\psi}{4b} \end{cases}$$

The last part is about the input generator, shown in the next figure, which allows to chose between some different actions, and reports all the possible outputs.

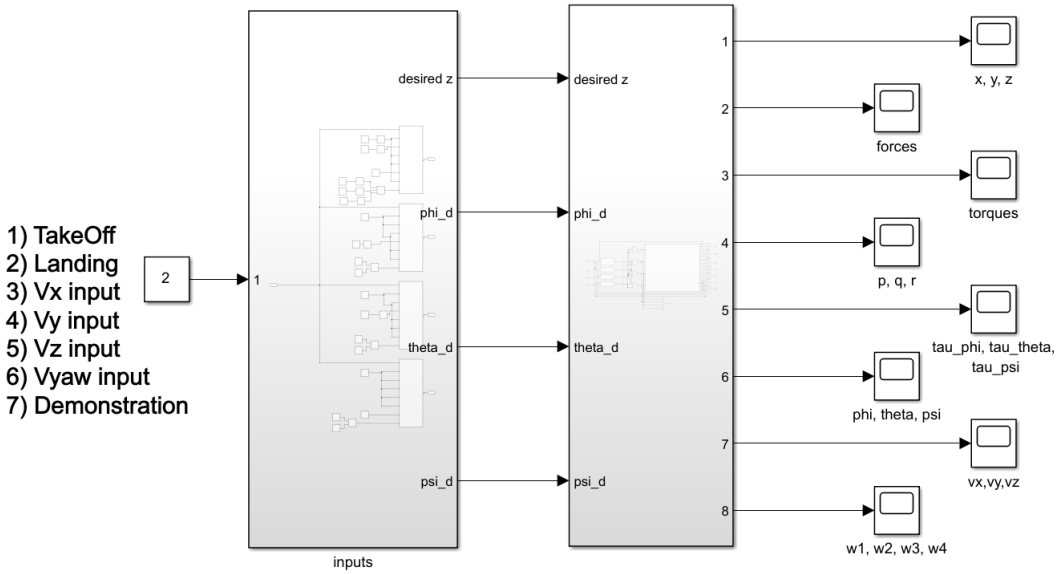

Figure 11: Complete Block Diagram

2.1.4 Drone modelling

In order to make a simulation we had to build a full complete model that could run the tests on it. To do so we designed and modelled a simple quadcopter using the software Catia V5.

We first had to decide on the dimensions of our drone. While looking at the objectives of our project, we did not find it necessary to have a big drone in order to perform some simple commands. That's why we opted for a 30 cm quadcopter since we needed it to be highly maniable and also lightweight. Thus, our quadcopter does not exceed 1 kg.

We did not find it necessary also to model the motors themselves but focus on modelling the shafted motor shells with the two types of propellers.

After modelling and customizing our drone we obtained the following model to play with for the simulation:

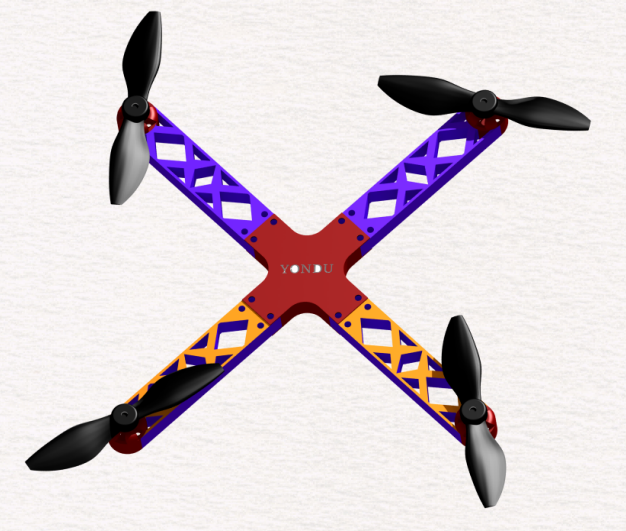

Figure 12: Quadcopter model designed for simulation

2.2 Simscape model built for simulation

2.2.1 Environment

Thanks to our built model using Catia software, we simply needed to import this model inside Simscape. In order to make the full design more real, along with the interactions between them, we rather preferred importing them separately and manage their position inside the Simscape environment.

To do so, we first built the simulation environment that could represent a classroom for example. Thus, we built two by two meters walls as using rigid transformations and solid blocks as below :

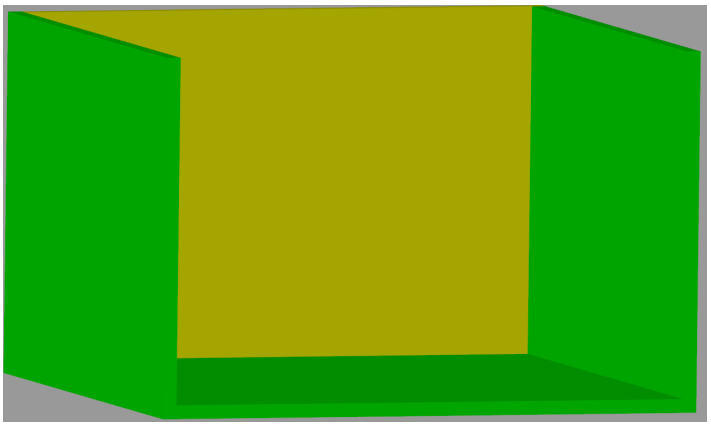

Figure 13: Appearance of the environment walls

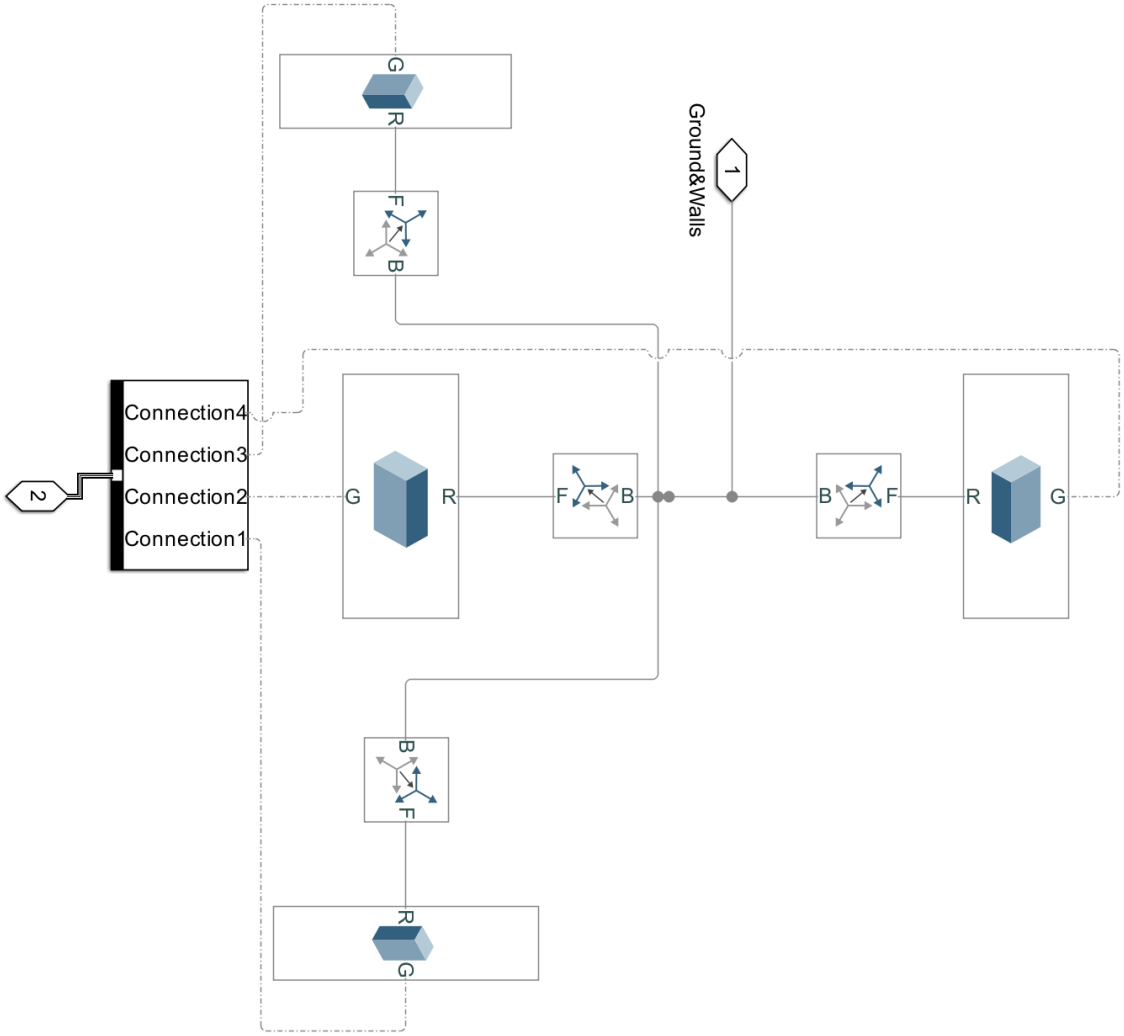

Figure 14: Environment structure

So this will be our environment to test our robot's movements responding to our input commands.

2.2.2 Drone body

To continue with our Simscape system, we built our quadcopter's body by importing pieces one by one and organizing them with rigid transformations as below:

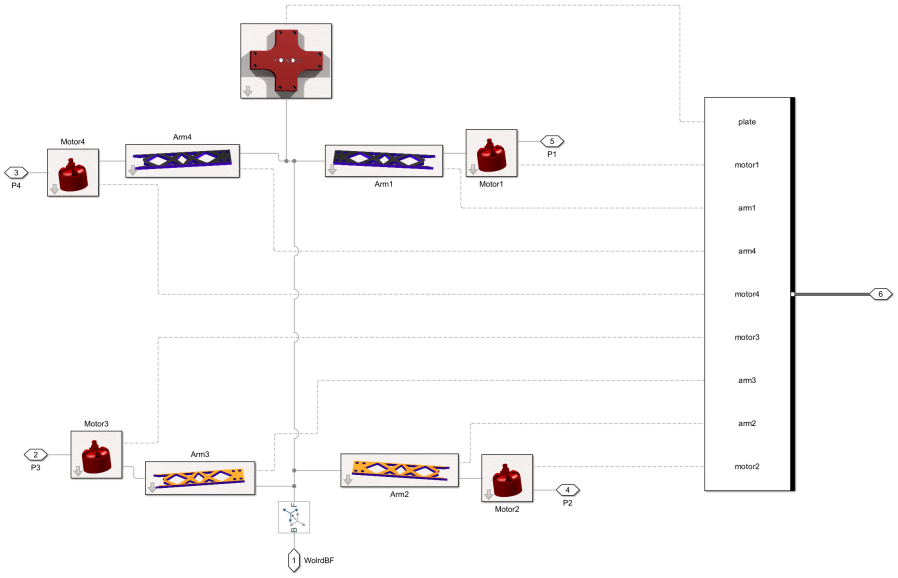

Figure 15: Main quadcopter body structure

With this environment set, we implemented the physical interactions and contacts (right block) between each elements to make the drone stay on the ground for example or even colliding with the walls.

2.3 Electro-mechanical system design

2.3.1 Electronical system

Once the environment is set, we focused on designing the electrical part of our quadcopter. To do so we created the following Simscape structure:

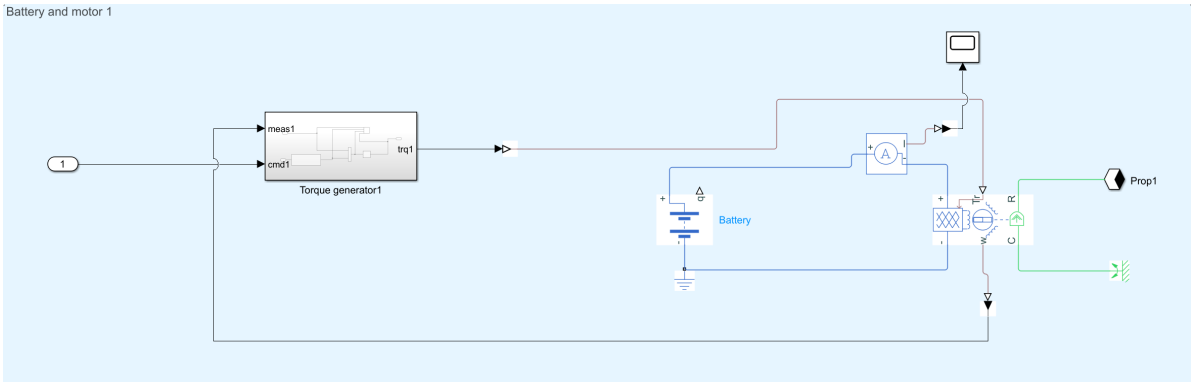

Figure 16: Electrical structure

This structure is first made of a battery on the top part and of a motor and drive system on the right part. Those elements parameters were found while searching for typical and average quadcopter datasheet. In order to make it as simpler as possible, we did not find it necessary to add any temperature influence on the battery.

The overall system is controlled by a torque generator made of a PID controller that we tuned in order to stabilize the output torque (in this example Prop1). This controller also takes as input the motor rotations per minute values coming from the flight control system.

2.3.2 Mechanical system

Once we got the electrical part regulating the torque sent to the motor, we needed to design the mechanical that will receive those torques and transform them into lift forces. To do so, we built the following mechanical system:

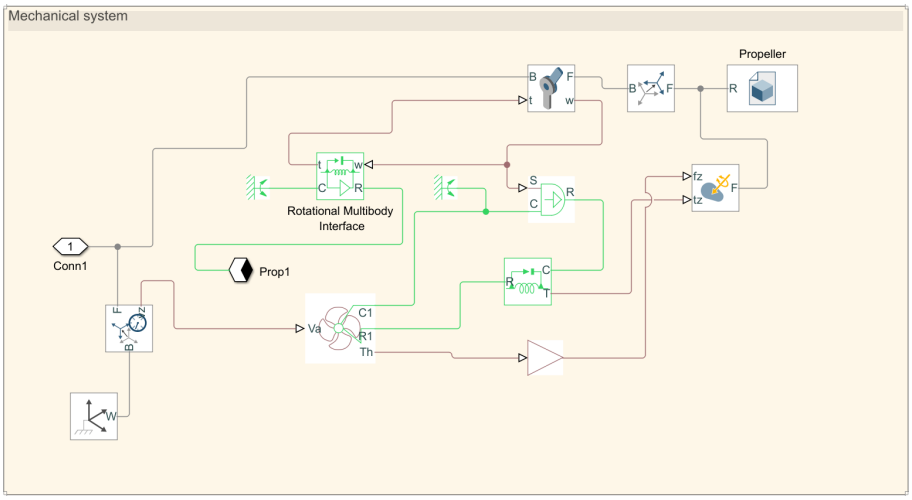

Figure 17: Mechanical structure

The first main block here is the Marine propeller block which defines the overall aerodynamics of the propellers, meaning their lift coefficient, their thrust coefficient or the air density. Although those parameters were difficult to find with respect to our quadcopter, we managed to find matching information through different documentations. Everything is detailed in the *Parameters.m* file for all of the present blocks. This Marine propeller block allows to convert rotary motion to linear thrust that will be used to lift each propeller.

The second important block is the Rotational Multibody Interface which serves as an interface between a Simscape Multibody joint and a Simscape mechanical rotational network.

In the end with this system we are able to get forces and torques to apply on propellers and rotate them while generating lift on the simulation.

2.4 Communication MATLAB - Simulink

To receive instruction from the pilot, we need to connect MATLAB to the MQTT broker. We tried to directly connect Simulink to the MQTT broker but we did not find a

way to connect it to our broker. So, we create a MATLAB script (*Utilities/udp_sender.m*) which connects to the broker, subscribes to the instructions published and then sends them to Simulink.

To send instructions to the Simulink, MATLAB uses the local network. It is a UDP communication on 127.0.0.1:25000. On Simulink, we place a receiver block from the SoC Blockset / Host I/O library to get the UDP messages send by MATLAB.

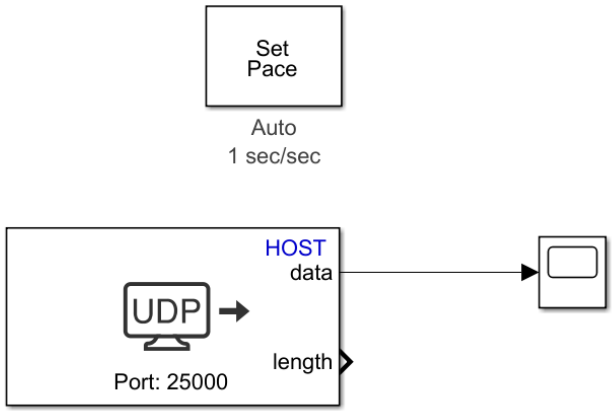

Figure 18: Simulink model for receiving UDP messages from MATLAB

This model is in the file *Utilities/communication.slx*.

3 ROS/Gazebo simulation

To simulate the drone on Gazebo, we used a model we've already used in another project (Au515 - Drones et asservissement visuel). The file structure of this part of the project is a bit complex and we did not create it. Moreover, installing all the project is long and not easy. To make it simple, we will only present the general steps of the work made to obtain the simulation.

3.1 Simulate the drone

So, for this part, we had to create our own environment which is just an empty place. That one was pretty easy to make. Then we had to write a script to launch Gazebo and make it launch this environment. Again, this was just a simple modification to make in the launching file. We need to launch the environment before trying to control the drone. Indeed, the node of the drone will connect to the environment, so if the environment is not running, we cannot use the drone.

The drone representation is simple :

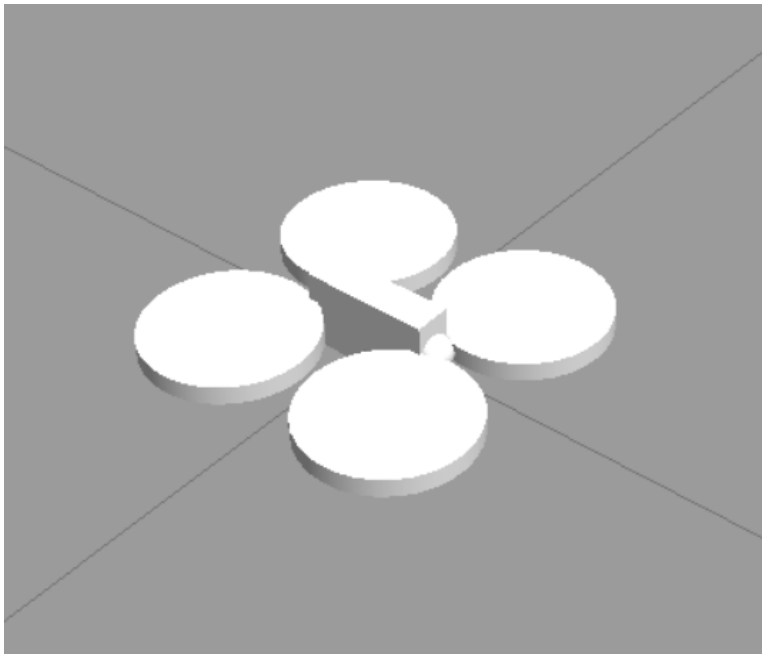

Figure 19: Drone simulated on Gazebo

3.1.1 Command interpretation

To make this drone fly and follow the instruction, we did the exact implementation of the flow chart presented in the Technical specification(Figure : 5). You can find the code in the file *yondu.py*.

The most important part of this code is how the message received from the MQTT server are interpreted. On this simulation, we are giving the drone RC commands or

calling services for landing and takeoff.

To call the takeoff service, the value published in the *takeoff* topic must be 1. In other words, at least one hand doing the open hand sign needs to be in the red area. To call the landing service, you just need to do the landing sign, the position in the image does not matter.

For the RC (Radio control) parameters, which are (vx, vy, vz, v_yaw), we need to do the sign associated to the parameter we want to change. To increase a parameter, the hand needs to be in the red area, and to decrease it the hand needs to be in the blue area. If the hand is in the center area, the parameter associated to the hand sign is set to 0. By doing the stop sign (hand closed), all RC parameters are set to 0.

You can see a video of the simulation here.

PART III

In real life demonstration

Figure 20: DJI Tello drone

In this section, we will explain how we set up the connection between the DJI Tello drone, the controller and the server to make a real life demonstration.

In real life demonstration

In this section, we are going to describe how we manage to make our project work in real life using a DJI Tello drone. So this is the integration part.

1 Setup

Tello drone software is completely closed, meaning we cannot change it. However, an API is available to make the robot execute some actions. You can access the documentation here. So we had to develop a code which will receive the instructions from the MQTT server and send them to the drone. This code will be running on a computer which will have to be connected to internet to receive the instruction and at the same time connected to the drone in Wifi. So, the setup is the following :

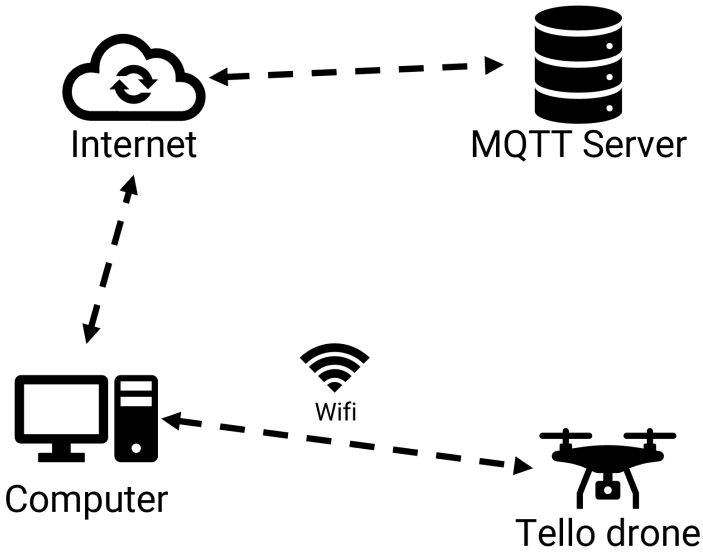

Figure 21: Connections between the drone, the computer and the MQTT server

Once this setup has been made, we were able to run the experiment.

2 Code to make the drone fly

We developed two versions of the code to make the drone fly. In this section, we are going to explain why and what are the differences.

2.1 Displacement instructions

In this first version, instructions are interpreted as displacements. The code associated to this part is *drone_main.py*.

In this version, we defined a reference angle and a reference distance. When a message is received, we identify on which topic it has been sent and we retrieve the value associated (-1, 0 or 1).

If the topic is a RC parameter (vx, vy, vz or v_yaw), then the displacement or rotation is equal to the reference distance or angle time the value sent. For instance, if the topic on which the message arrived is *vx* and the value associated is 1, the drone will move forward.

With this method, we can make the drone fly but it has a jerking motion, which is not really convenient. Therefore, we wrote another code where we are directly controlling the RC parameters. Plus, this method cannot allow mixed movements such as moving up and turning on itself. Indeed, we are sending a command at each message which can be moving forward, backward, up, down, turn on yourself clockwise and counter clockwise, land or takeoff. Any combination is impossible.

You can see a video of the test of this code here.

2.2 RC commands

To remove the jerking motion and allow mixed movements, we decided to write a code where we directly send RC commands. The code associated to this section is *rc_drone_main.py*.

In this code, we defined a reference linear and rotational speeds. The computation of the desired speeds follows the same process as in the previous section with distance instructions. Another difference is that we wait 0.5 second before sending the RC command. In that way, we allow different RC parameters to be updated before being sent to the drone. This allows combined movements. Moreover, we removed the jerking motion as the drone is not trying to stabilize around a given position like with the previous method. However, we need to be careful while controlling the drone because it will not stop moving if we do not tell it to stop.

We do not have a video for this code, but it was demonstrated on the presentation day.

Conclusion

Through this Yondu project, we managed to control a drone by hand gestures. In terms of applications, it could be useful in the fields of education and military among others.

The development of this group project was based on the V-cycle. Indeed, we started from the definition of the system requirements (functional analysis and technical specifications).

Then, we dealt with the system design : the hand signs detection and recognition thanks to MediaPipe library on Python and artificial intelligence (Classification and Regression Tree model). Besides, we implemented the model (created on Catia V5) on Simulink with the environment and the physical interactions between the different elements on Simscape for simulation. Then, we designed the electro-mechanical part of our quadcopter. After that, we wanted to connect Simulink to the MQTT broker in order to get the instructions sent by the pilot but in vain. But we found a way to make Matlab connect to the MQTT broker and transmit received instructions to Simulink using UDP communication on the local network. In parallel to the work done on Matlab/Simulink Scimscape, the model has also been developed on ROS/Gazebo. So, we were able to control the drone in a simulated environment, using MQTT server and the computer camera.

Finally, the time for real life experimentation has arrived. For that, we used the DJI Tello drone that we managed to connect to the computer via the Wi-Fi and the server.

To sum up, the objectives have been met. But some improvements remain to be considered. We could mention a faster response time, a task planification commanded by a precise hand sign, a higher number of commands associated with other signs and last but not least, the fact that the drone follows the movements of the hand using Recurrent Neural Networks algorithm for example.