POLYTECHNIC OF TURIN

Master's Degree in Aerospace Engineering

Master's Degree Thesis

Autonomous Precision Landing for UAVs on a Mars-like environment in ROS/Gazebo

Supervisors

Prof. Giorgio GUGLIERI

Dott. Stefano PRIMATESTA

Dott. Simone GODIO

Candidate

Gabriele Angelo PELISSERO

October 2022

Abstract

Space exploration and UAVs have been undergoing exponential growth in recent years, and with them the related technologies deployed. More than a hundred space launches are planned for 2023, and it is expected that UAVs will be increasingly associated with these types of missions. The process leading to their success is very complex and expensive, both in terms of economics and schedule, and for this reason, simulators are often employed within the implementation process. As a result of these, it is possible to test algorithms without causing damage to the physical hardware, extend safety and operability, and definitely have a less environmental impact.

The work in this thesis aims to create a simulation environment to reproduce a possible Mars mission performed through the use of two robots: a rover and a UAV. Specifically, this work will mainly focus on implementing and simulating a precision landing algorithm of the UAV on the rover, exploiting technologies that could potentially be deployed on the red planet. This has been achieved first is simulation using the Software In The Loop (SITL) framework, and later, the same configuration has been implemented and tested on real hardware, to validate the development precision landing approach in reality. A comparison of current technologies mainly adopted for this purpose has been made, after which the combination of UWB and AprilTag was chosen as the solution. The graphical user interface has been created with Gazebo, while the Robot Operating System (ROS) has been used to develop the software architecture. Gazebo and ROS are used to simultaneously generate a simulation environment with a rover (with an AprilTag on it) and drone on the Martian ground, as well as to estimate the relative position between drone and rover using a Kalman filter, and the last part responsible for the robot control algorithms.

The precision landing algorithm was tested and analyzed in three different situations: using the UWB only, using the UWB and the AprilTag as inputs to the Kalman filter for more precise position estimation, and, finally, using the UWB and the AprilTag as inputs in the control part.

All the three configurations share the same control strategy consisting of two phases. First, a Proportional (P) controller is used when the UAV is far from the target. Then, within a certain distance, a more performing PID (Proportional-Integrative-Derivative) controller is used to get the drone closer to the rover. In the approach to the ground touch phase, the drone will no longer use data from the UWB, but only data from the AprilTag that will be clearly visible from the camera.

In order to perform experimental tests, the drone has been adapted to install the RaspiCam used for the precision landing. The CAD of a support for the RaspiCam has been created and, then, 3D printed using an FDM technique so that it would be as efficient as possible in terms of weight, strength, supports, and interface. Subsequently, the mono-camera has been calibrated, and it has been tested first that the AprilTag detection was working properly, and then that the presented algorithm was efficacious by performing tests on a drone in flight.

Acknowledgements

I would first of all thank my advisor Giorgio Guglieri for enabling me to pursue such an interesting and challenging thesis work, in which I have had a great interest for several years.

Special thanks are especially due to my supervisor Stefano Primatesta, who has been for me a mentor always ready to provide me with the appropriate guidance at every stage of the realization of my work, with his endless helpfulness and kindness, and to my supervisor Simone Godio, for constantly following me and offering me useful tools, improvements and suggestions.

I would like to thank the entire PIC4SeR, for hosting me in their structure where I found a very stimulating and welcoming environment, and in particular Marco, Gianluca and Leoluca, for offering me a valuable help for any of my needs during my thesis period. Thanks also to the Draft team for all the knowledge imparted.

I would like to thank my family for supporting me during all these years, and especially my mother, for always being there for me in front of all the difficulties that life has presented to me and for inspiring me to never put myself down.

Special thanks go to Sibilla, my partner, the person who has been closest to me throughout this path, capable of understanding me, encouraging me and motivating me in all my choices, putting my mind at ease in the dark moments and rejoicing in the happier ones.

I would also like to thank all my friends who supported me, particularly Gregorio, a special person I could always count on, and a special thought goes to all those course colleagues who always believed in my particular path, making the study sessions less stressful and contributing to the realization of wonderful projects.

I also wish to thank the volunteer association Contardo Ferrini, for giving me the opportunity to grow personally and achieve this objective.

Finally, a big thanks to myself for never giving up, always prepared to get back up after every defeat, the me of ten years ago would be very proud of the person I've become.

$\sim$ *Never give up*

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

UAV
Unmanned Aerial Vehicle

ROS
Robot Operating System

UWB
Ultra-wideband

MIL
Model In The Loop

SITL
Software In The Loop

HIL
Hardware In The Loop

HW
Hardware

SW
Software

API
Application Programming Interfaces

PID
Proportional Integral Derivative

RTPS

Real-time Transport Protocol Service

DDS

Data Distribution Service for Real Time Systems

RtPS

Real Time Publish Subscribe)

uORB

Micro Object Request Broker

UDP

User Datagram Protocol

KF

Kalman Filter

IMU

Inertial Measurement Unit

HQ

High Quality

FDM

Fused Deposition Modeling

GUI

Graphical User Interface

Chapter 1

Introduction

1.1 Thesis Objectives

The objective of this thesis work is to create a simulation environment, using ROS and Gazebo software, to reproduce a possible Mars mission performed through the use of a drone and a rover. Specifically, this work will mainly focus on implementing a precision landing algorithm of the drone on the rover, using ultra-wideband technology and an AprilTag. This will be achieved first in simulation, using only the software, and later, the same code will be implemented and tested on a real drone. So the objectives can be summarized as:

- SITL simulation phase: the creation of the simulation environment where the algorithms will be tested, using only and exclusively the software.
- Testing phase: implementation of the software tested in simulation on hardware to validate its effectiveness in reality.

1.2 Thesis organization

The thesis is divided into seven chapters, which can be summarized as follows:

1. **ROS Environment/Gazebo:**. in this first chapter, the general features of these two software programs that are necessary for the creation of any simulation environment are presented.
2. **Drone-Rover Simulation:** based on the chapter preceding this one, this section initially presents the various types of simulation (MIL, SITL, HIL) and then those of the autopilot, finally choosing the configuration that best adapts to the objective of this thesis.

3. **Simulation environment software architecture:** this chapter outlines the software architecture of the starting simulation environment.

4. **Precision Landing Algorithm:** this section describes in detail the process followed so that the drone could perform a landing accurately and efficiently. In addition, the final simulation system is presented with the implementation of an adaptive PID controller.

5. **Hardware architecture:** description of the hardware structure of the drone. Also described is the process of implementing the camera, which is required for AprilTag detection, and the resulting 3D model of the support connected to the drone.

6. **Test:** phase in which the algorithms tested in the simulation were implemented on physical hardware.

7. **Conclusions and future works:** final results and possible future implementations to improve the system as a whole.

Chapter 2

ROS Environment/Gazebo

2.1 Introduction to ROS

The acronym ROS stands for Robot Operating System and it is an open-source platform that mediates between an operating system and the software that uses it. It mainly provides:

- Typical services of an operating system
- Abstraction of hardware
- Multiprocess communication
- Process and packet management

Figure 2.1: ROS overview [1]

It also provides visualization and simulation tools, libraries, and other useful conventions for implementing robotic applications [1].
Since its inception at the Standford Artificial Intelligence Laboratory in 2007, it has been constantly updated and changed until it currently has two versions (ROS1 and ROS2) and is used in a lot of robots, both in academic and corporate settings. ROS is officially supported by the Linux-based Ubuntu and Debian operating systems, while for other operating systems such as Windows, Fedora, and Android there is an experimental version, as visible in the figure 2.1 . Lastly, ROS is not a programming language, but integrates codes written in Python, C++, and Lisp [2].

2.2 Architecture

The main components of ROS can be divided into:

- **ROS Node:** a node is a process inside the ROS network that performs a specific task, carrying out the main functions in the system. Nodes interact with each other by exchanging messages using topics, services and actions [3].

- **ROS Topic:** topics are the main communication channels in a network, where nodes can publish or receive data. The former are called *publishers* while the latter *subscribers* [4].

- **ROS Message:** topics exchange data that are encoded in messages. These can be simple (boolean, integer, string type) or they can have a complex structure (e.g. messages related to navigation or geometry) [5].

- **ROS Master:** that is a unique node that not only initializes a ROS application but also manages the communication between all the various nodes that will be executed. In ROS1 it must always be initialized manually from the terminal, while in ROS2 this is done automatically [6].

- **Service:** These are communication channels that allow, in contrast to topics, synchronous communication. Nodes that publish to the service, in this case, are called *Client*, while those that receive are called *Server* [7].

- **Actions:** These are complex communication systems used to perform long tasks.

- **ROS Launch:** It is a tool to enclose multiple nodes in a single file called *launch* file.

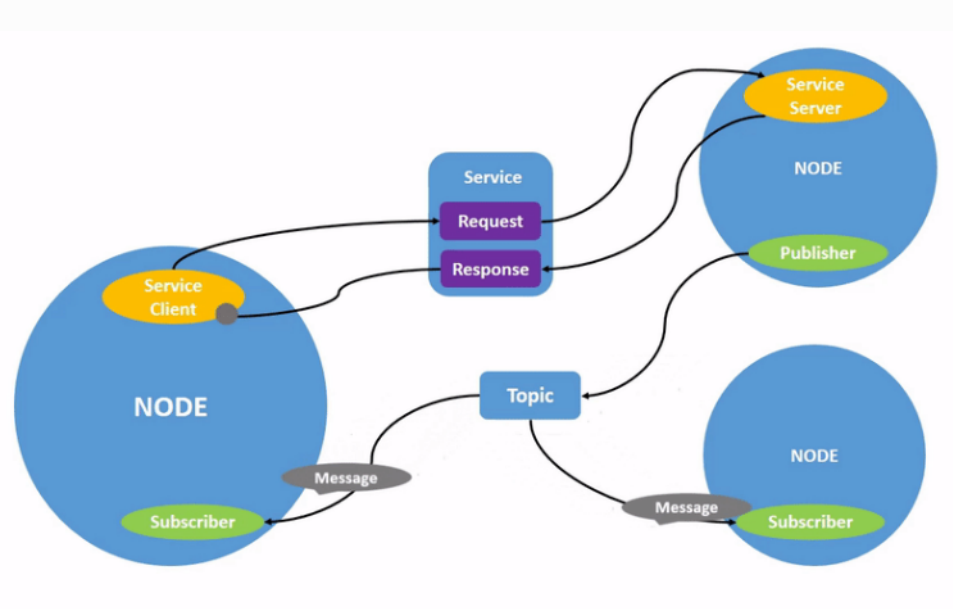


Figure 2.2: Communication between nodes and topics [8]

2.3 ROS Tools

As written in the first section of this chapter, ROS offers several very useful tools for analysis and debugging. These are very important and useful for getting visual feedback on what is happening within the system. The main ones are:

2.3.1 rqt_graph

This tool allows to graphically display the correlation between, in this case, the processes (including therefore nodes and topics) that are active at the time of execution. In this way, in addition to immediately visualizing which nodes are acting as publishers and which are acting as subscribers, debugging analysis can be performed to verify that the nodes are correctly executed [9].

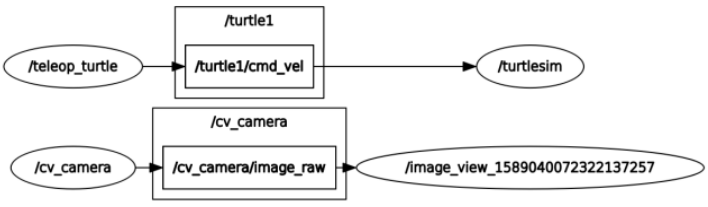


Figure 2.3: Example graphical user interface of rqt_graph [1]

2.3.2 RViz

RVIZ is the best-known and most versatile of the tools, it allows to visualize within a three-dimensional space any data the software publishes through its topics.

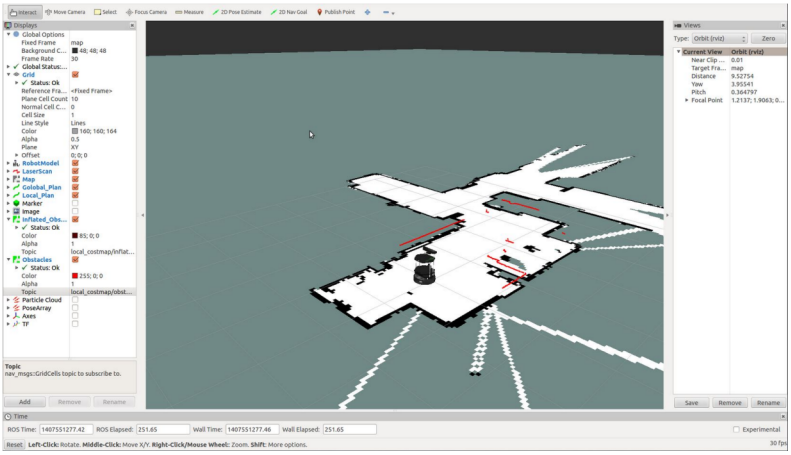


Figure 2.4: Example graphical user interface of RViz [1]

2.3.3 rqt

On the other hand, this tool is useful for showing, using a graph, how a certain feature varies over time. Based on these plots, it is possible to derive conclusions about the algorithm's effectiveness.

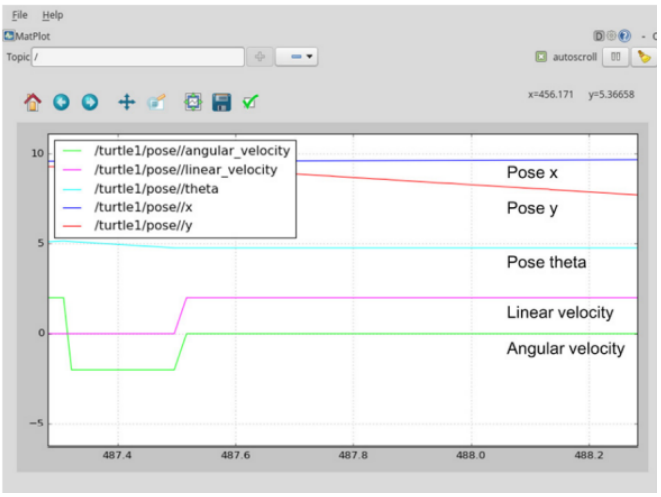


Figure 2.5: Example graphic interface of rqt [1]

2.4 Simulation tools

2.4.1 Overview

Simulators are becoming increasingly essential nowadays, just consider those for aircraft that have become an integral part of flying qualities for aircraft certification. Various tools have therefore been created in the world of robotics to enable the most realistic simulation possible, useful for testing algorithms and designing robots. The most commonly used ones are:

- Gazebo
- FlightGear
- JSBSim
- jMAVSim
- AirSim

All these simulators have different features, but the most widely used and recommended is Gazebo [10].

2.4.2 Gazebo

Gazebo is a powerful 3D simulation environment that enables to interface with ROS and reproduces vehicular dynamics leading to the extremely coherent simulation of the environment. It has many supported vehicles (quadcopter, VTOL, rover, aircraft, submarines...) and can also be used for multi-vehicle simulation in complex indoor and outdoor environments. Gazebo offers a very high degree of fidelity, this is due to the integration of: a wide variety of sensors, a rich model library, robot environments, and convenient graphical interfaces. These features just outlined allow extremely accurate testing of robotics algorithms and execution of tests with realistic scenarios. The main peculiarity of this software is actually that it allows a system to be developed on a robot without having the physical hardware and, therefore, without damaging the robot in the case of incorrect algorithms. Once it has been ensured that the system works properly, it can then be implemented on the physical robot.

In addition, it is an open source program that uses advanced 3D graphics by relying on OGRE (Open-source Graphics Rendering Engines), which enables the generation of realistic environments using textures, lights and shadows [11]. Some of his additions are:

- numerous sensors, such as:
  - two-dimensional and three-dimensional camera
  - depth camera
  - contact sensors
  - force and torque sensors
- different robot models, such as:
  - Turtlebot
  - IRobot Create
  - Drone Iris
  - Rover
- different scenarios, such as:
  - empty world
  - road
  - runway

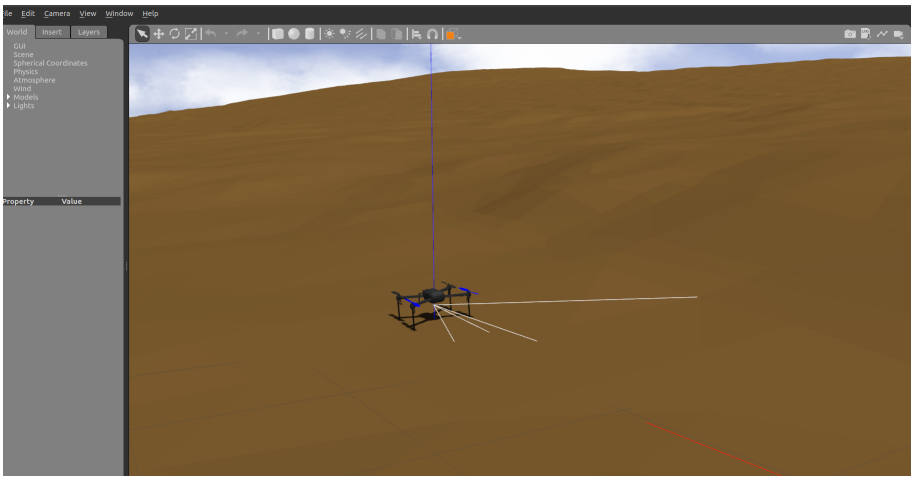


Figure 2.6: Gazebo custom environment: Iris drone on Martian ground

In addition to these, however, it is possible to develop custom sensors, models, and scenarios through the use respectively of plugins, files in .SDF format, and files in .WORLD format. [12]. For example, in figure 2.6, it is possible to see a custom model where a Martian environment has been implemented and above it implemented the iris drone. It is very important to properly manage the spawn dependencies between the two models to prevent them from getting stuck and causing mismatches.

Simulated sensors and physical data can be transmitted from Gazebo to ROS, so the robot's actuator commands can be sent between the two software. As a result, the robot software can be represented identically on both the real robot and the simulator. [13].

Chapter 3

Drone-rover simulation

3.1 Model Based Design Approach

Generally in the software world, model-based design is the best way to meet cost, quality, and time requirements. It allows every feature of the software to be tested during development, and its feasibility can be evaluated precisely due to simulation. This latter can be mainly of three types:

- **MIL simulation:** the entire system is modeled (e.g., on Simulink) to simulate the complete environment and test the control laws. The aim is to obtain functional validation.
- **SITL simulation:** by SITL simulation is meant the testing of software, written for a particular system, that is made operational and simulated in its behavior in an equivalent software environment with the physical conditions it will have in flight. This is extremely effective in that it enables debugging of the software module and evaluation of latencies, delays, and failures. What most characterizes a simulation of this type is that it can simulate the mission and evaluate its anomalies without damaging the physical hardware, which is often very expensive. The aim then is to obtain validation of the generation of the programming code.
- **HIL simulation:** in the HIL simulation, the software runs directly on the flight hardware. So the software does not run on a generic computational platform (pc or workstation), but it is run by the same hardware that will be used on board the aircraft. The aim is to obtain validation on HW-SW integration.

A certain control law is then typically developed and tested on a MIL model. Based on the results, code is generated through a programming language that is tested

with a SITL model. Finally, this is validated with a HIL simulator to verify that the performance still meets the requirements. [14]

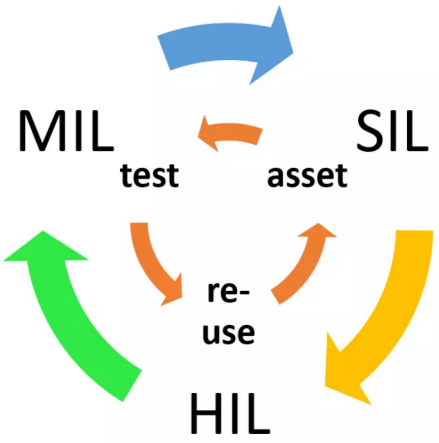


Figure 3.1: Model-Based design approach [14]

3.2 SITL simulation

SITL simulation is the one used in this thesis work because it is the most useful when testing a code change in the controller. The latter, more commonly referred to as an autopilot, can be distinguished from both a hardware and a software perspective. Concerning the former, several electronic boards can host the controller code, but the most famous is definitely Pixhawk. On the software side, instead, there are several possibilities: dRonin, Betaflight, Librepilot, Ardupilot, and PX4. The last two are the most widely used and have excellent documentation to be able to set up the simulation environment correctly. Thus, SITL simulation takes advantage of the fact that the autopilot can be run and built on the PC or workstation platform, allowing it to not necessarily have the physical autopilot hardware itself. In addition, it supports a large number of vehicles [15]:

- multi-rotor aircraft (including the four-rotor)
- fixed-wing aircraft
- ground vehicles (such as the rover)
- underwater vehicles
- gimbals for cameras
- a wide range of optional sensors, such as Lidar and optical flow sensors

These configurations can be chosen directly in the flight controller's GCS interface, as seen in the figure 3.2.

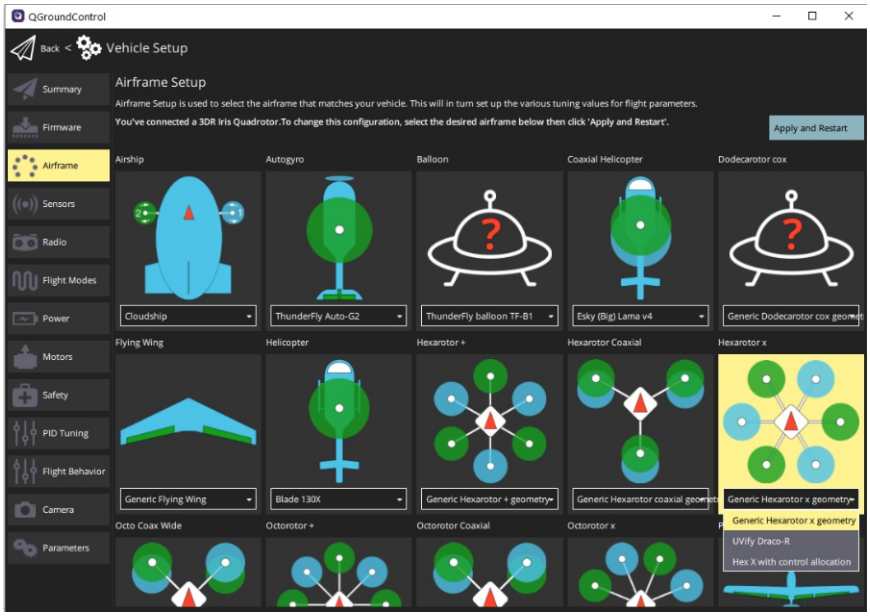


Figure 3.2: Autopilot Airframe configuration [16]

The purpose of this SITL simulation is therefore to exclude the control given by the radio control, which goes through the GCS and uses only the computer to set parameters and send commands. To do this, it is essential to use the MAVlink communication protocol and other Mavlink-compatible libraries to control both the rover and the drone. A widely used tool is MAVROS, often use to enable the communication between the ROS environment and the Autopilot. Using these tools, it is possible to create a communication bridge between the computer, the GCS, and the autopilot [17].

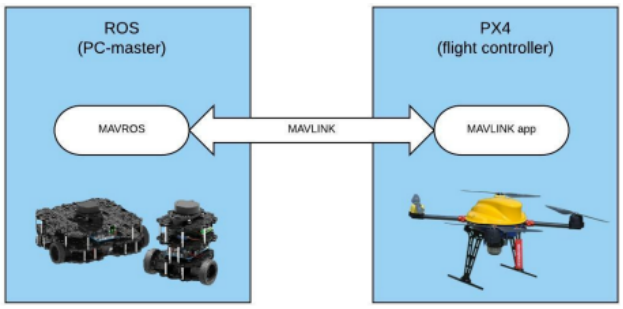


Figure 3.3: MAVLink and MAVROS Architecture[17]

3.3 Autopilots

As written in the previous section, the most commonly used autopilots from the software point of view are PX4 and ArduPilot. They are both widely used and have very good features, and for the purpose of this thesis, both configurations were developed for the initial phase only.

3.3.1 Ardupilot

The generic architecture referring to the SITL environment of ArduPilot is shown in Fig. 3.4, where the link to the simulator GUI (in this case FlightGear instead of Gazebo) is also presented.

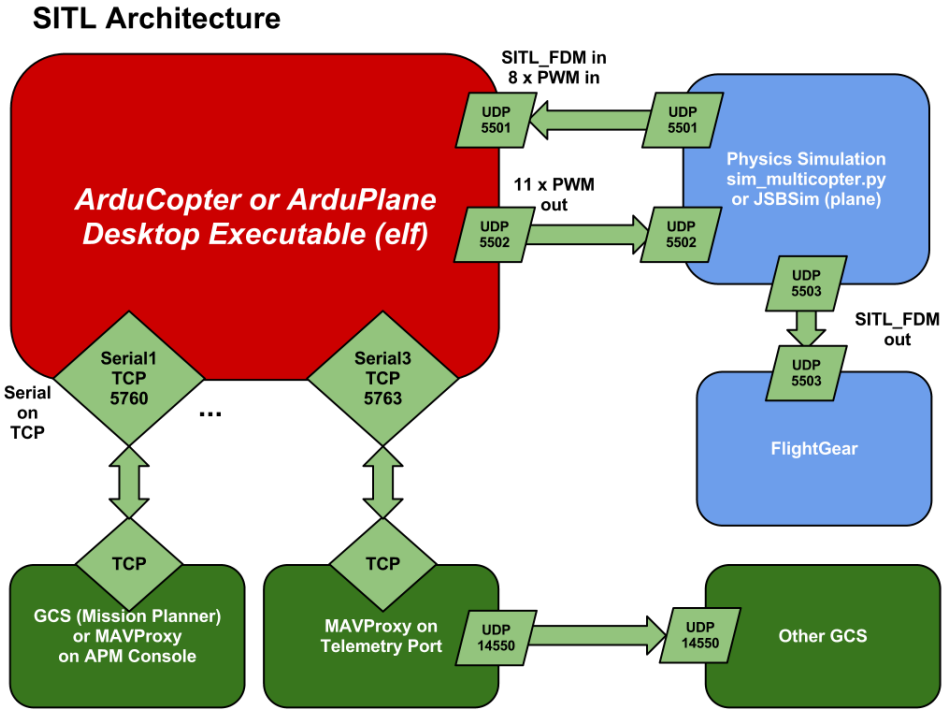


Figure 3.4: SITL simulation architecture with ArduPilot [15]

In order to run a SITL simulation with ArduPilot, mainly two steps are required:

- **Install ArduPilot plugin**: once installed both ROS and Gazebo, it is necessary to install the plugin via the ArduPilot Github repository [18]. Referring to figure 3.5, it can be seen that in the simulation environment, on the left, there is a console through which commands can be given:

– GUIDED: command needed when users want to fly the drone without setting up a specific mission.

– arm throttle: control that allows the motors to be able to turn. It is necessary to give it before giving the takeoff command.

– takeoff: command that allows the motors to be able to turn at a speed that the propellers can create lift and the drone can move off the ground. This command is followed by a number indicating the desired altitude.

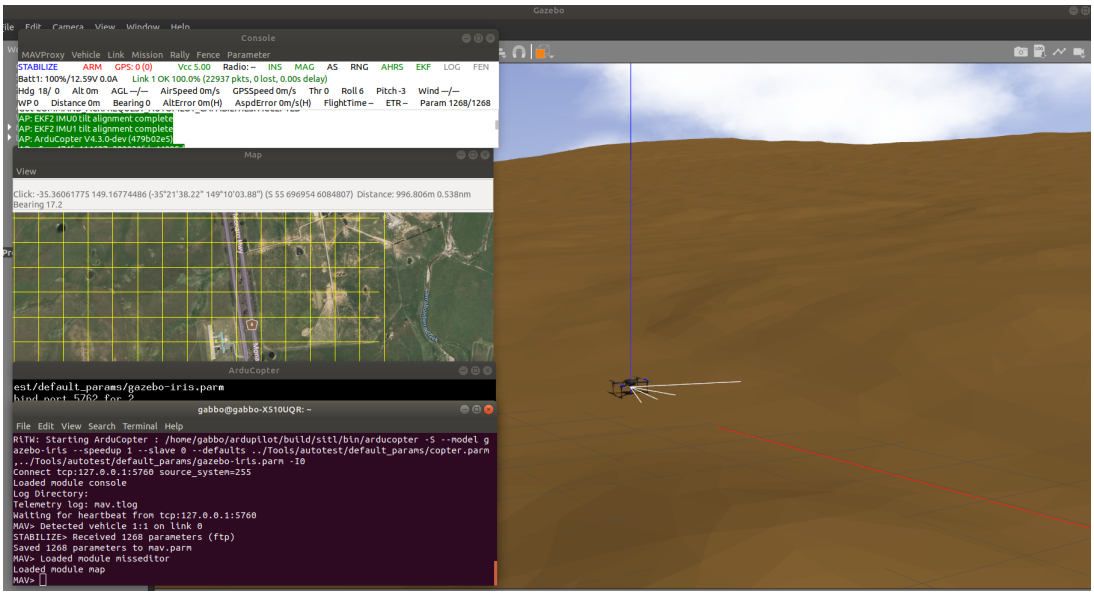


Figure 3.5: Drone simulation with ArduPilot console

- **Install MAVROS package:** this is essential so that it is possible to control the robot directly from the terminal via ROS. Compared with the previous point, as visible in Figure 3.6, it is possible to give commands directly from the terminal and no longer via the console. Thanks to the MAVROS package, it is possible to make speed and/or position messages simply by publishing a message on the respective topics.

This configuration is great but compatible only with the first version of ROS; in fact, there is still a lack of documentation that can interface Ardupilot with ROS2. For this reason, it was preferred to develop another simulation environment using this time the latest version of ROS (ROS2) with the PX4 controller.

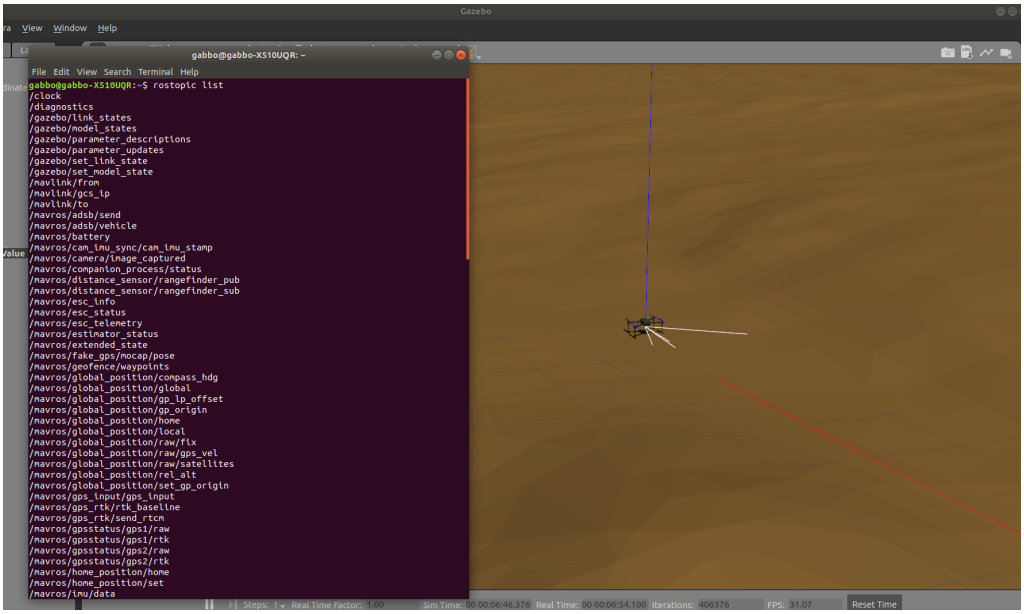


Figure 3.6: Drone simulation with MAVROS package

3.3.2 PX4

In this case, the generic architecture referring to the SITL environment of PX4 is depicted in figure 3.7, where it can be seen that the communication with PX4 is made using the simulator's MAVlink API. The latter defines a series of MAVlink messages, which provide the flight controller with sensor data from the simulated world and return engine and actuator values from the flight code that will be applied to the simulated aircraft [19].
This controller, unlike its previous one, is fully compatible with the latest version of ROS and, thanks to the documentation[20], it is possible to perform SITL simulation easily. In fact, thanks to the middleware (DDS/RTPS) it is possible to add an RtPS interface to the PX4 autopilot, allowing the exchange of uORB messages between the various PX4 internal components and DDS applications in real-time. This allows the creation of publisher and subscriber nodes that interface directly with PX4 uORB topics as shown in Figure 3.8.
As this is the best configuration, the simulation environment used in the following chapters will be presented.

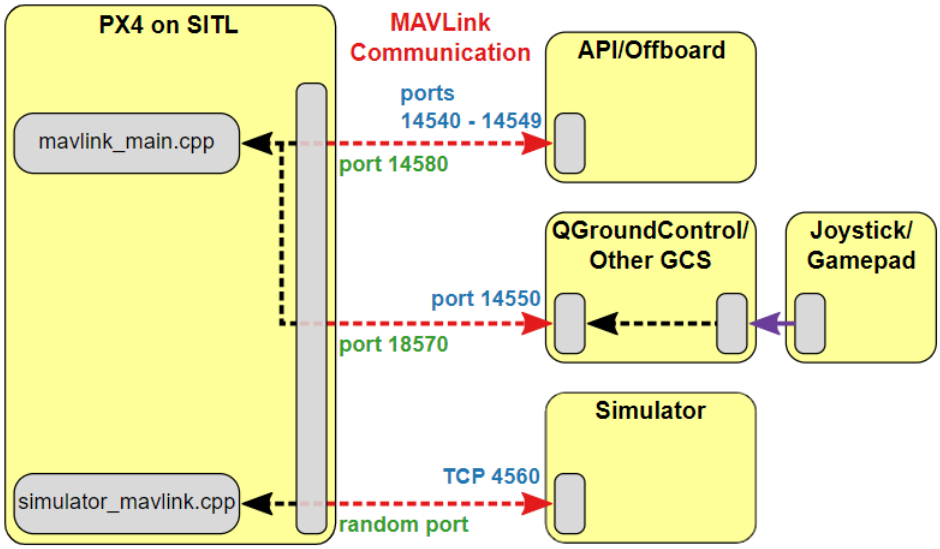


Figure 3.7: SITL simulation architecture with PX4 [19]

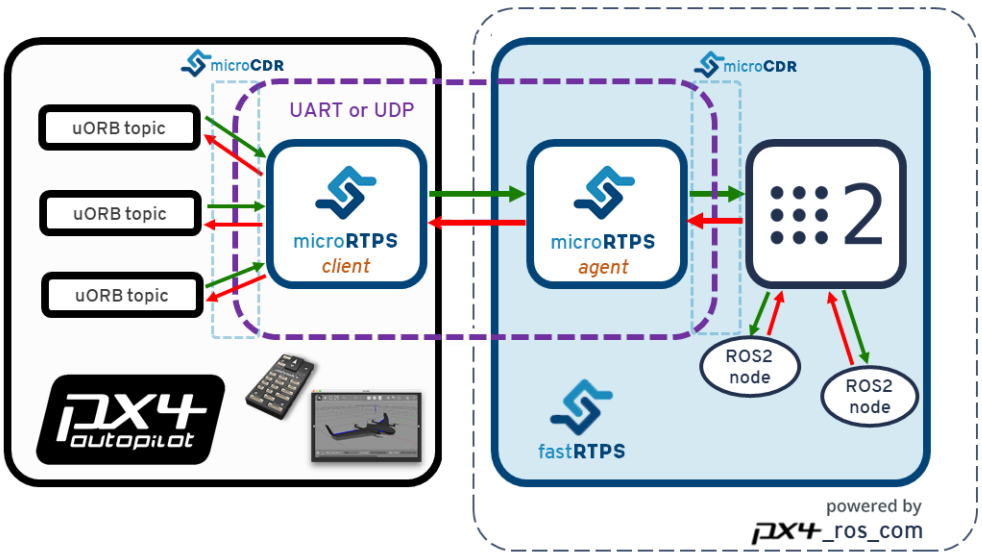


Figure 3.8: Interface between PX4 And ROS2 [20]

Chapter 4

Software architecture simulation environment

4.1 Overview

With the assumptions of the previous chapters, and configured the Linux environment with ROS, Gazebo, and PX4, this chapter presents the software architecture of the simulation environment that allowed the objective of this thesis to be achieved. The work presented in this chapter refers partially to the following source [21], which gave me an excellent starting point for setting up the appropriate environment and for developing the precision landing algorithm that will be discussed more specifically in the chapter 5. The software simulation environment can be divided into three main parts, related to:

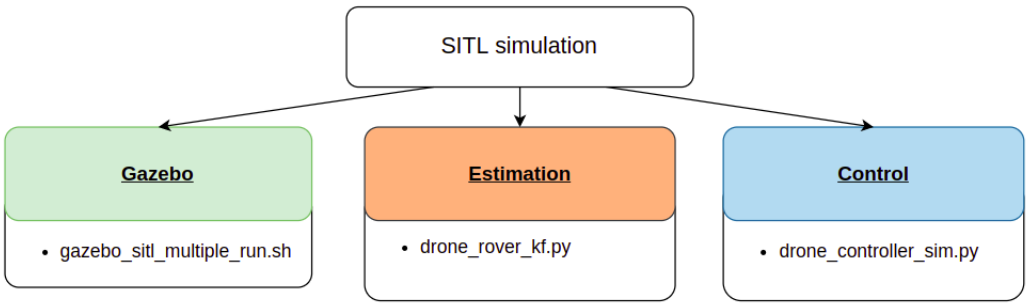


Figure 4.1: SITL simulation organization

- **Gazebo**: generation of the simulation environment using Gazebo. It includes the spawning of three models:

  - custom drone
  - custom rover
  - martian ground

- **Estimation**: it contains the system state estimation algorithms.
- **Control**: it contains the precision landing and control algorithms.

4.2 Environment configuration on Gazebo

4.2.1 Multiple vehicles with ROS and Gazebo configuration

Using a multi-vehicle approach, the environment must be set up correctly to avoid inconsistencies. In addition, a *launch* file 2.2 should be generated to ensure that all vehicles, within the Gazebo model, are developed with a single command on the terminal. Therefore, it is necessary that, for each simulated vehicle, is defined [22]:

- **Gazebo model**: this model, in *.urdf* or *.sdf* format, in addition to being contained in a specific folder in the workspace, must contain an argument called *mavlink_udp_port* that defines the UDP port on which Gazebo will communicate with the PX4 node. The latter must be defined for each vehicle and placed within the launch file.
- **PX4 node**: due to the previous step, it is possible to connect the PX4 simulated flight controller by setting the parameter *sitl_udp_prt* and matching it to the *mavlink_udp_port*.
- **MAVROS node**: to make the connection with ROS, it is necessary that within the file *.urdf* or *.sdf* a MAVLink stream is contained on a unique ports stream and these ports must match those in the launch file.

4.2.2 SITL simulation environment

Once all these steps have been performed, the three models mentioned above can be generated simultaneously through *gazebo_sitl_multiple_run.sh*. However, these models will still not be controllable until the nodes discussed in 4.3 and 4.4 are executed.
Both the rover and the drone are not default Gazebo models, but custom models created ad-hoc so that the simulation would be as similar as possible to the situation recreated in the tests, which can be seen in the chapter 7.
The *drone* has been customized based on Gazebo's default "Iris drone" model, on which it has been mounted the Holybro X500 quadcopter black landing gear.

Furthermore, the overall structure contains one white ultra-wideband tag module and a RasPi Cam single camera.
The *rover*, on the other hand, is schematized as a grey cube supported on three dark blue wheels, and above it, there are four white ultra-wideband anchor modules.
The *Martian terrain* is characterized by bumpy and uneven ground. This is to validate the effectiveness of the algorithm even under adverse conditions.

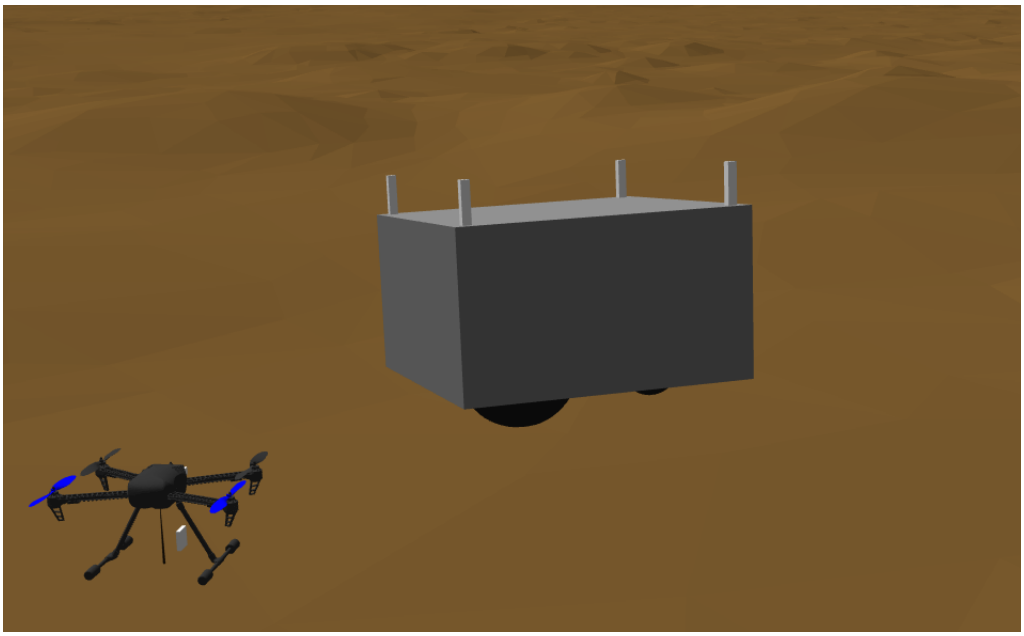


Figure 4.2: Gazebo environment for the SITL simulation

4.3 System state estimation

This part is fundamental for the drone to be able to orient itself and, subsequently, locate the position of the rover to approach the precision landing phase. In fact, without this part, the drone could only implement a simple land and take off and the rover only move, but without orienting themselves in space. As a result, it is important to employ several sensors to estimate the location and relative speed of the drone with respect to the rover. This estimation may then be made using a particular type of filter.

4.3.1 Filters overview

For statistics and control theory, it is necessary to deploy an algorithm that uses a series of measurements and produces an estimate of unknown variables, which tend

to be more accurate than those based on a single one. The measurements include statistical noise and other inaccuracies, while the estimation is done using a joint probability distribution over the variables for each time interval [23]. These best estimate algorithms are called filters and, according to their characteristics, they have different names. The main ones are presented in the table 4.3.1.

Kalman Filter theory

Linear dynamical systems discretized in the time domain are the basis of the *Kalman Filter* and are modeled on the basis of linear operators that, after each time increment, are responsible for generating a new state state of the system perturbed by errors (including Gaussian noise). This can be visualized in figure 4.3, where it can be seen the discretized disposition in time in the steps $k-1, k, k+1$. In addition, the state *hidden* can be visualized, which refers to the target measurement to be obtained, which is precisely hidden since it cannot be measured directly. When the data is *supplied by user* an additional linear operator will be used which is mixed with other noise in order to generate measurable outputs defined as *observed* [23].

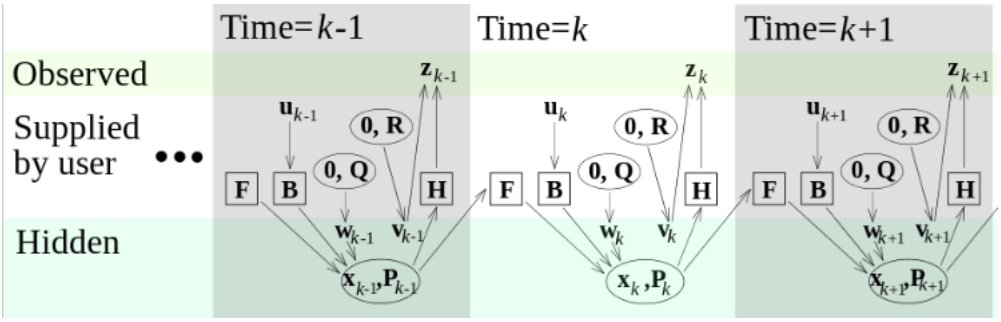


Figure 4.3: Kalman Filter Model [23]

According to this theory, the state system evolves from time $k-1$ to time k following this equation:

$$x_k = F_k \cdot x_{k-1} + B_k \cdot u_k + w_k \tag{4.1}$$

where:

- $\mathbf{F}_k$: the state transition model is what is used to apply the current state to the prior state x_{k-1}
- $\boldsymbol{u}_k$: it is the control vector and represents the input to the *input-control model* B_k.

- $\boldsymbol{w}_k$: it is the process noise assumed with a normal distribution N with zero mean and covariance $\mathbf{Q}_k$: $w_k \sim N(0, Q_k)$. This means that with many measurements, the noise in the readings will take values with most of them located close to the zero mean and less far from it, generating a Gaussian distribution.

Furthermore, at the time k the output *observed* z_k of the state *hidden* x_k is made according to

$$z_k = H_k \cdot x_k + v_k \tag{4.2}$$

where:

- $\mathbf{H}_k$: it is the observation model which charts the *hidden* state space into the *observed* one
- $\boldsymbol{v}_k$: it is the measurement noise that is added and it is assumed to be a Gaussian white noise with zero mean and covariance $\mathbf{R}_k$: $v_k \sim N(0, R_k)$

In addition, it is necessary to build a model of the reference robot so that the KF can combine the measurements and predictions in order to find the final optimal estimation. The cycle that characterizes the Kalman filter involves an initial phase of *prediction* of a current state, based on the estimation of the previous state, a phase in which the state measurements are obtained and, finally, a phase of *update* of the prediction using the observation noise [24].

State estimator	Model	Gaussian distribution	Computational cost
Kalman Filter (KF)	Linear	Yes	Low
Extended Kalman Filter (EKF)	Locally linear	Yes	Medium
Unscented Kalman Filter (UKF)	Non-linear	Yes	Medium
Particle filter (PF)	Non-linear	No	Hight

Table 4.1: State estimator comparation [25]

The Kalman filter model is great for linear systems, but nonlinear modeling is often necessary to accurately describe a certain type of situation. If the state transition and measurement functions are linear, then the distribution maintains its Gaussian property, but if it were to be non-linear then the state distribution may also no longer be Gaussian and thus the Kalman filter algorithm may not converge. So in this case an Extended Kalman Filter is used that linearizes to a non-linear function around the mean of the current state estimate. However, it has drawbacks because it has a higher computational cost and it doesn't provide a good approximation for highly non-linear systems. To solve these problems it would be possible to switch to either a UKF or a PF, the latter is the only one that works for any arbitrary distribution, but this obviously incurs an even greater computational cost.

4.3.2 Estimation with Kalman filter in the simulation

Based on what was written in the previous subsection, the KF was the best as the first iteration in this application because of its simplicity compared to the others. Figure 4.4 represents a scheme of how the KF operates, through the algorithm *drone_rover_kf.py* written in Python, within the simulation presented in this thesis. When the input command "land_on_target" is given (which means that the drone should land on the rover), it will be managed by the part of the control algorithms that will later go on to communicate with the simulated PX4. The latter will then proceed to provide the drone's sensor parameters, which, together with the data provided by the UWB and the rover's compass, provide the state estimation algorithm with the necessary inputs to be able to calculate the drone's position and velocity relative to the rover.

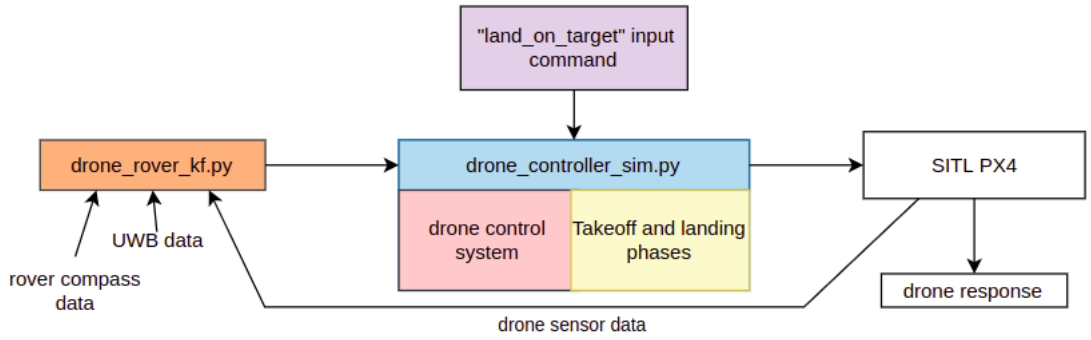


Figure 4.4: Software architecture operation scheme

Specifically, the algorithm *drone_rover_kf.py*, based on the considerations made in 4.3.1, implements a prediction model that allows the drone and the rover to be modeled via differential equations. The assumptions made in order to accomplish this are:

- the drone model is described by a constant acceleration model
- the rover is described by a constant velocity model

These measurements are published asynchronously on several ROS2 topics. Consequently, modeling many observation models is required. Every timestep dt is used to run the prediction phase, and whenever the corresponding data are available, the update phase runs [21]. The data from the drone's sensors (such as range sensor and IMU), those from the UWB, and those from the rover's compass then represent the observation models, which will allow the following outputs:

- drone position, velocity, and acceleration

- rover yaw angle, position, and velocity

This data will be provided to the control algorithms part, managed by the code *drone_controller_sim.py*, so that the drone and rover can communicate with each other in order to be able to control a precision landing phase.

4.4 Control Algorithms

In this chapter, the control part, already mentioned in reference to the figure 4.4, is explained in more detail. After the environment has been opened on Gazebo, it is necessary to launch algorithms that are responsible for controlling the two robots and, until this part is executed, it will not be possible in any way to operate on them. In the functional flowchart 4.5, it is possible to visualize the correct flow of actions that will allow the achievement of a precision landing of the drone on the rover. The algorithm *drone_controller_sim.py* will then consist of two main parts:

- the drone control system (marked in light pink color)
- takeoff and landing phases (marked in light yellow color)

4.4.1 Takeoff and landing phases

In the flowchart, it is possible to see the effect of two input commands: *takeoff* and *land_on_target*. The former is responsible for arming the drone and allowing it to reach a predefined hover altitude. This will be maintained until the latter input command is given. When this is given, the drone will go on to perform a series of actions fully automatically. The drone will begin the precision landing phase which starts with the chase of the rover. Thanks to the algorithms of *system state* 4.3 that remain active at all times, both the relative position and velocity between the drone and the rover are calculated. When both of these two values are below a certain threshold, the *touch* phase is then possible. This phase consists of a condition in which the drone will have a very low descent speed and, upon reaching a certain height, it will automatically turn off its motors so that it can land on the platform while also avoiding the ground effect. It is necessary that even one of the two values (p or v) does not meet the requisites that the drone returns to the chase phase. Setting limit parameters is important so that there is confidence that the drone will land smoothly and safely. To explain the control part, it is first necessary to introduce the generic operation of a PID controller so that the use of such choices can be properly justified.

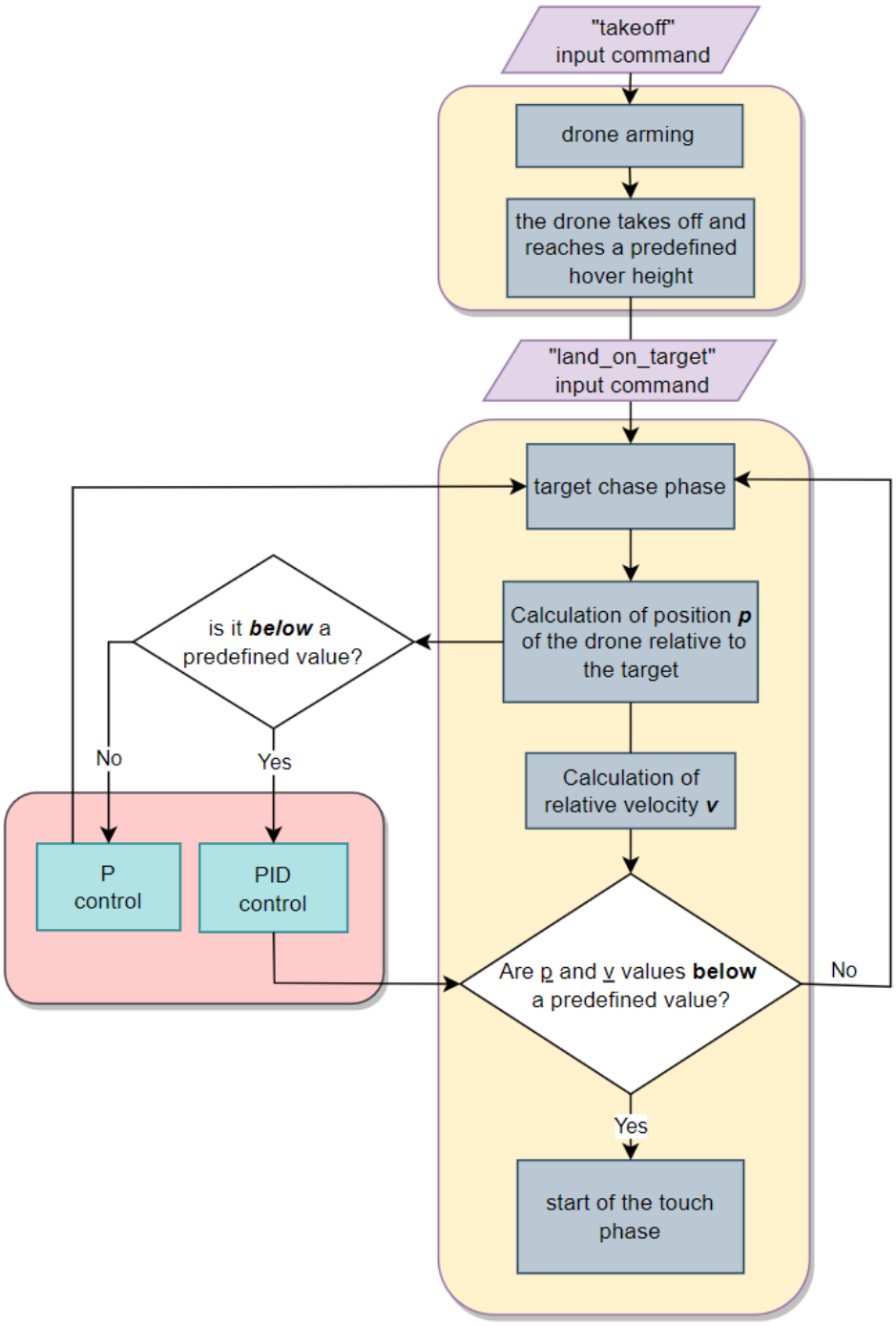


Figure 4.5: Control Algorithm flowchart

4.4.2 PID controller

The PID is a controller used for many applications in several different fields. This is because of its simplicity and effectiveness of use. Starting from the schematic 4.6 on the left, there is the desired signal $r(t)$, which is subtracted from the actual output signal $y(t)$ thus generating an error $e(t)$. The latter is then continuously computed and a term-based correction is applied:

- **Proportional**: this action is obtained by multiplying the error signal $e(t)$ by a constant K_p. This constant must always be present in a controller of this type, and in some simple cases, this may be all that is needed. Depending on how this parameter is varied will result in a control output proportional to it.

- **Integral**: by adding a control effect based on the historical cumulative value of the error, this term keeps the previous values of the error signal in memory and integrates them over time in order to remove any residual error. It is multiplied by an integrating constant K_i.

- **Derivative**: The derivative term is involved in quickly compensating for changes in the error signal. In the case where $e(t)$ is increasing too much, the derivative action tries to compensate for this deviation by reason of its rate of change, without waiting for the error to become significant. It is possible to act on this parameter thanks to the constant K_d.

The combined action of these three terms then produces a $u(t)$ given by the sum of the three terms given above.

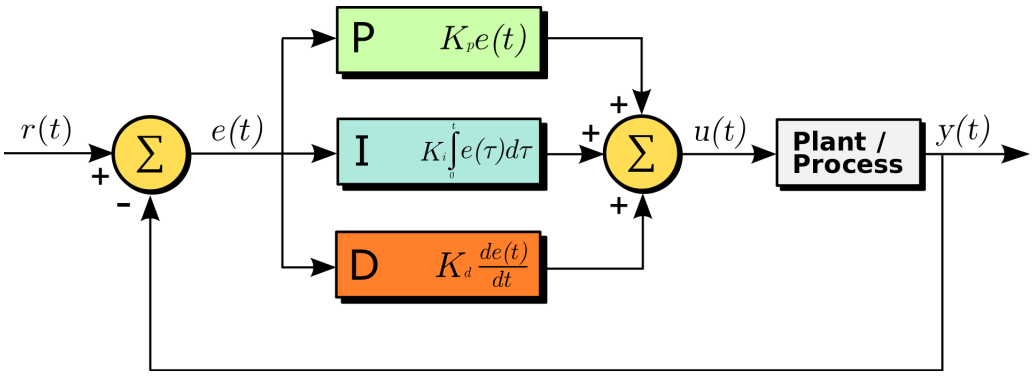


Figure 4.6: PID controller scheme [26]

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)\,d\tau + K_d \cdot \frac{de(t)}{dt} \tag{4.3}$$

The constants K_p, K_i, K_d require tuning that will be different for each application. This is because they depend on several factors including the behavior of the measurement sensor and any delays in the control signal [26].

4.4.3 Drone control sysyem

This section explains the part located to the left in the flowchart 4.5, related to the control system of the drone (the one outlined by the light pink box). It can be observed that if the relative position between the drone and the rover is below a certain value, there will be a different control that can be either a PID or just a P. Based on the references in the previous section 4.4.2, it is useless to use an integrative and derivative control when the UAV is far away from the target. In fact, if it is not at a close enough distance from the rover, it will start again with the chase phase. On the other hand, as soon as it reaches a close enough position, the controller will become a PID, thus allowing the integrative and derivative action to come into effect and therefore be able to reach the landing pad much more accurately. In this case, the *proportional* provides the drone with a greater response, the *integrative* on the other hand is responsible for causing the error that is generated between the drone's desired position and the actual position to tend to zero, while finally, the *derivative* term is responsible for damping the drone's oscillations in reaching the target.

Chapter 5

Precision landing algorithm

5.1 Methods to perform precision landing with a drone

This thesis aims to improve precision landing as much as possible from what has already been outlined in the 4.4 chapter. To do this, it is useful to understand what technologies are currently being used.

	Landing accuracy	Necessary HW	Price	Implementation in simulation
IR	± 15cm	Rangefinder	∼400€	Medium
Marker	± 10cm	Raspi-cam	∼20€	Easy
GPS RTK	± 20cm	None	∼350€	Difficult
UWB	± 10cm	1 target and 4 anchors	∼25€ per module	Easy

Table 5.1: Precision landing methods comparison

5.1.1 IR Sensor/Beacon

This infrared device allows communication between an IR-LOCK sensor, mounted on the drone, and an IR beacon as a target, mounted on the rover. This solution, in addition to allowing a very accurate landing in the range of 10-15 cm, has other advantages including being plug and play and being able to see even in the dark or in fog. However, the price is very

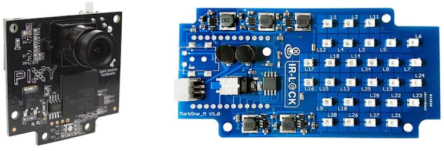


Figure 5.1: IR Sensor/Beacon

high due to the use of two devices, and it is a solution that is poorly suited to very bright light. In a possible real implementation, it needs a rangefinder (e.g., a lidar), while a possible mockup realization would be of medium difficulty since plugins have already been developed to interface this device with Gazebo, but they need to be configured [27].

5.1.2 Marker

The marker is definitely the most widely used method of precision landing as it is really very functional and cost-effective. In fact, all that is needed is a simple Raspberry Pi cam, costing a few tens of euros, and a 2D marker easily printable. The function of the camera is to visualize the marker, which is typically square in shape to allow for more accurate calibration and detection, however, it will be necessary to develop a script that can recognize it and extrapolate the required data to be able to make a precision landing. Fortunately, a number of solutions have already been developed in this regard, and thanks to the use of OpenCV, it is undemanding possible to implement such a marker in a simulation environment. This solution, nevertheless, has several critical issues including:

- limited visualization of the marker based on the size with which it is printed
- a bright light may not make the marker visualization correctly
- if the landing pad is moving, the marker visualization might be distorted
- the drone must be exactly over the marker in order to be able to have it properly displayed

Figure 5.2: Markers example [28]

This option is excellent because it can be used for indoor applications, where there are fewer light reflections and ambient conditions that won't interfere with accurate detection and identification.
There are several markers used for this purpose, including Aruco, ARTag, CALTag and AprilTag. In each case, they all involve two stages of detection: determination of unique features and identification. The first stage scans the image because of its square-shaped feature, while the second validates the interior of the image to determine whether it is a marker or another object. Depending on the situation, it may be useful to use one marker rather than another, this is because each one has different detection and recognition algorithms that determine its strengths and weaknesses. Dealing more specifically with the marker **AprilTag**, its detection process involves several steps:

- scan for linear segments used to form a square
- detection of squares
- calculating orientation and position of the tag
- decoding the barcode

For the internal image identification part, Apriltag uses the lexicode system, which allows increasing accuracy and reducing the number of false positives [28].

5.1.3 GPS RTK

GPS RTK (Real Time Kinematics) is a measurement method that achieves centimeter-level positioning accuracy. This remarkable difference in accuracy from a normal GPS can be achieved because, it is not enough to simply use GPS satellites and a receiver on earth, but this is a different positioning method that uses:

- Satellites: these still remain essential, but in this case information is acquired from both the drone receiver and the base station, thus achieving greater position accuracy.
- Base station: this station is the additional component that characterizes an RTK system. This remains fixed at a defined point on the ground and continuously communicates the GPS position with the drone.
- RTK receiver: this device, also called a rover, is mobile in that it is part of the drone's hardware equipment.

The innovation consists in using two receivers that communicate data to each other using a GSM phone signal or radio transmission. Thus knowing the position of the base, which is always fixed, the onboard SW will apply corrections to the RTK receiver determining a more accurate position [29].
In the figure 5.3 it is possible to see the *base station* in the middle and two different types of *RTK receiver* that will be implemented on the drone. The one on the left is better performing, but bulkier and heavier (about 58g more than the one on the right).
This solution is great in that this way with one device it can perform both navigations (due to the fact that it is possible to travel high ranges) and precision landing, however, it remains very limited to applications where there is no noise or interference (so open fields for example) in order to avoid signal loss.
Implementation in simulation is challenging as it is difficult to find open source plugins and in any case, the simulation would be limited to open scenarios, as it would be difficult to simulate the interference from an urban environment that could generate noise on the drone.

Figure 5.3: GPS RTK base station and on-board receiver[30]

5.1.4 UWB

Ultra-wideband is a transmission technology developed to send and receive signals by the use of radio frequency energy pulses of extremely short time duration and therefore with very wide spectral occupancy [31].
Each UWB module can be set by software as either *anchor* or as *tag*, the former being fixed devices that emit a precise position (e.g., they can be mounted on the vertices of a rover-mounted landing platform), while the latter are mobile devices (e.g., it can be mounted on the drone). Both of these devices exchange information to determine the distance between them, so the more anchors there are, the more the tag can communicate with them to increase the accuracy of the estimate. With specific reference to precision landing, this method is definitely the most innovative and promising and the advantages of using such a solution are many:

- due to the low spectral power, it allows to resist the phenomena of multipath and jamming, which well affect a GPS-based solution instead
- the UWB modules are cost-effective, small and low power consumption, thus easily configurable in an existing HW

- it allows location applications because of its strong time domain resolution.

Figure 5.4: UWB Model Decawave EVB1000

This solution achieves an accuracy of about 10 centimeters and has already been developed in simulation, however, it should be kept in mind that an appropriate algorithm must be used to estimate the position between tags and anchors, and the range of use is limited to about 60 meters.

5.2 Implementation of Apriltag in the simulation

According to the references in the previous section 5.1, it can be seen that all these methods may be more or less effective depending on the specific application. In

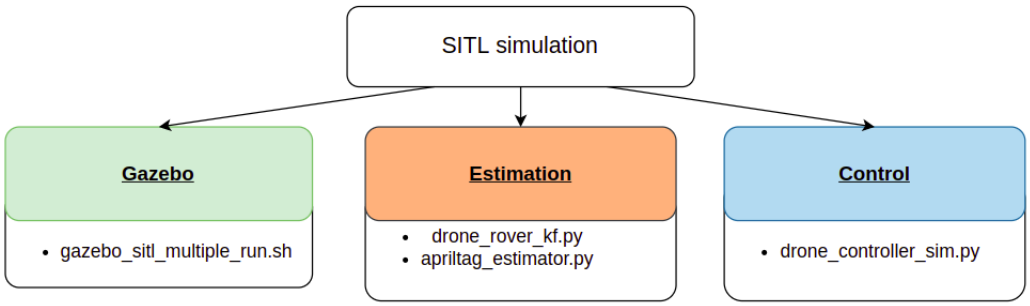


Figure 5.5: SITL simulation organization with AprilTag implementation

a Martian environment, the use of GPS would be impossible since there is still no satellite network like the Earth's, while better adapted is the use of UWB. Instead, both IR 5.1.1 and Marker 5.1.2 can be used as add-ons to improve the

accuracy of precision landing even more. In this case, the choice was made to use an AprilTag associated with the UWB; in this way, higher efficiency, lower weight, and lower computational and economic cost can be achieved. Therefore, this chapter will present the solution of the AprilTag embedded in the architecture outlined in chapter 4.

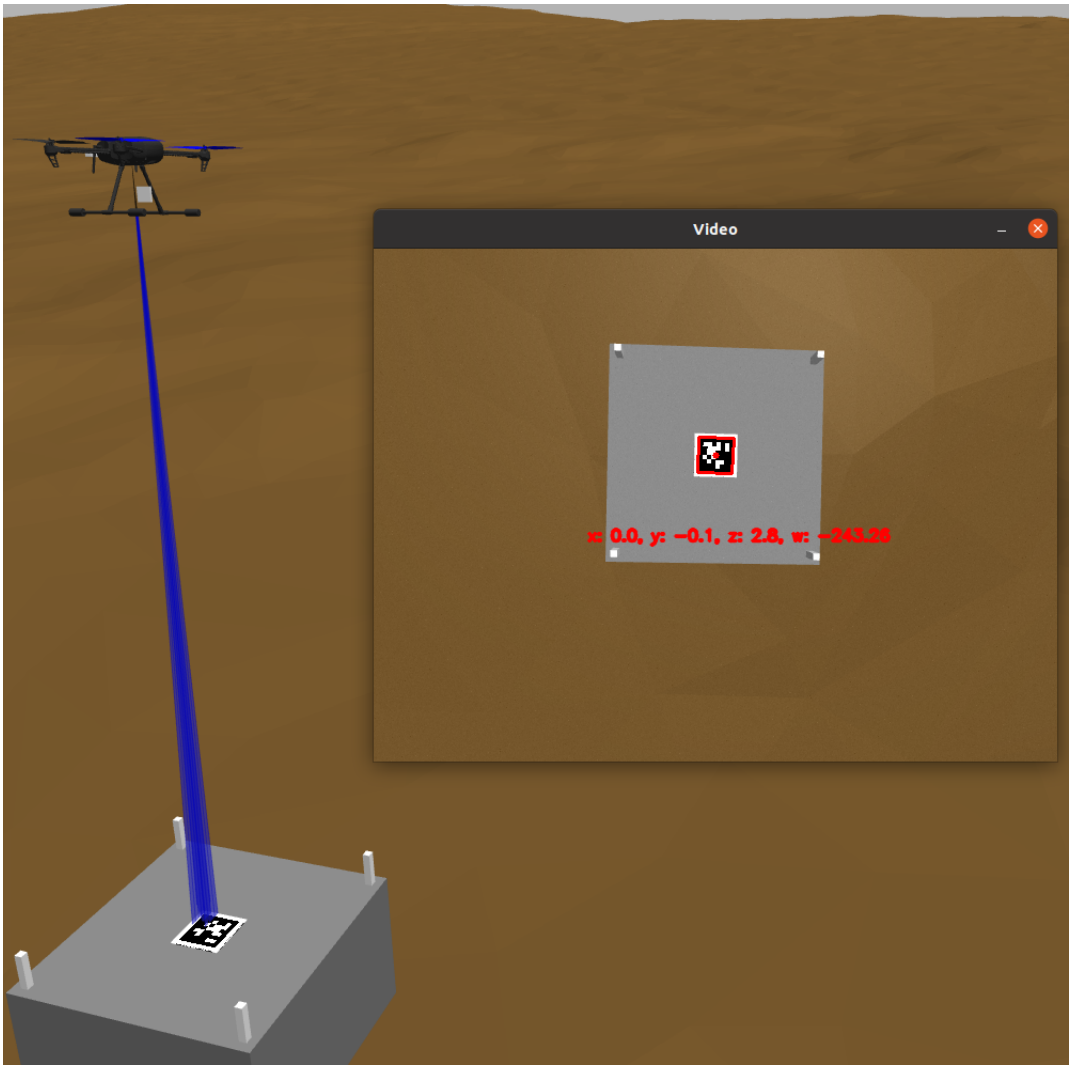


Figure 5.6: AprilTag implementation on Gazebo environment

From the scheme 5.5, compared to the one already presented in 4.1, it can be seen that the script *"apriltag_estimator.py"* has appeared, which will take care of performing the AprilTag detection thanks to the implementation of the OpenCV library directly in the Python script. Indeed, it will be possible to display a tab dedicated to the detection of the AprilTag, as visible in the figure 5.6 and,

thanks to this script, it will then be possible to have an estimate of the position x, y, z and orientation w of the drone with respect to the Marker. In addition, the implementation of the AprilTag within the simulation environment is done due to the file *gazebo_sitl_multiple_run.sh*, in which the files *.sdf* of the drone and rover. The latter was modified by adding a piece of HTML code to insert the AprilTag above it. Data from this execution can be implemented within the simulation with two different methodologies:

- **Apriltag implementation in the control algorithm**: in the section 5.3 data from AprilTag detection are implemented directly in the control part of the simulator.
- **Apriltag implementation in the Kalman filter**: in the section 5.4 data from AprilTag detection are filtered by the Kalman Filter so as to provide a more accurate calculation of relative position between drone and rover.

5.3 Apriltag implementation in the control algorithm

This section explains the operation of the 5.7 diagram in which, compared to the diagram 4.4 presented in the previous chapter, it is illustrated how the script *apriltag_estimator.py* acts within the simulation. The received data can obviously be used only when the Marker is visible from the mono camera, and in this case, it is used directly in the controller part. In this way, the relative position between rover and drone is calculated exactly as presented in chapter 4, but in the final part of the landing on the platform, it will use additional data that will allow the system to be more precise. In order to do this, it is necessary to create a publisher and a subscriber with respect to a topic (2.2) so that the information can be communicated correctly.

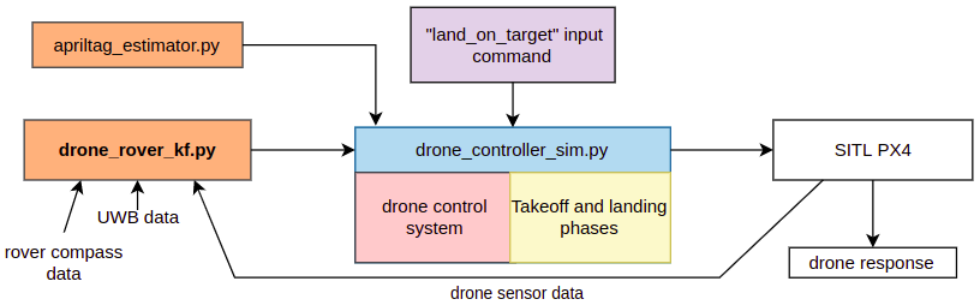


Figure 5.7: Software architecture with AprilTag implementation

5.3.1 Publisher and Subscriber implementation

Within *apriltag_estimator.py*, a *publisher* node has been created that will release data (of position x, y, z and orientation w of the drone with respect to the Marker) on a topic called */AprilTag_estimator*. Instead, as part of the *drone_controller_sim.py* code, a *subscriber* node was created to subscribe to that topic and take that data so that it can be used directly in the control system. Each time a subscriber is introduced, it is necessary to introduce, again within the same code, a *callback* function that will take care of creating a vector containing the data of interest.

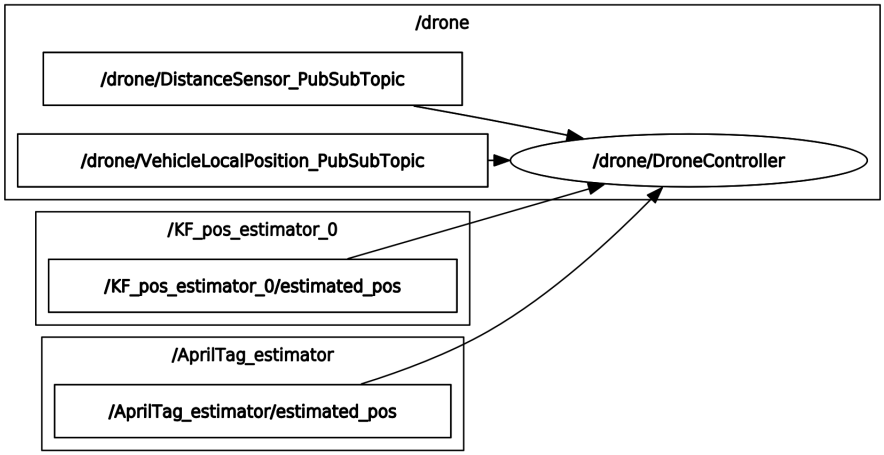


Figure 5.8: rqt_graph using AprilTag directly in the control part

Due to rqt_graph, a tool provided by ROS, it is possible to graphically visualize, in the figure in 5.8, what has been explained above. As previously detailed, the UWB modules, the rover's compass, and the sensor data from the PX4 flight controller are filtered through the KF, resulting in an estimate given by */KF_pos_estimator_0*. The latter and the topic related to the AprilTag, will both be in communication with the general node that is concerned with the control part */drone/DroneController*. The */drone/DistanceSensor* and */drone/VehicleLocalPosition* are given directly from the PX4 flight controller.

5.3.2 Pseudo-code for AprilTag implementation

From the pseudo-code 1 it is possible to see that with this implementation, the drone will always use the data coming from the Kalman Filter, but if the AprilTag provides a clearly visible output then that data will be used which will be more accurate. In case the drone is farther than the *PID switch position* then it will use only proportional control without obviously using the data coming from the Marker.

Algorithm 1 Precision landing algorithm with AprilTag implementation

Data: Relative position between UAV and rover = X
PID switch position = 2m

Result: UAV will use AprilTag information directly in the PID controller if that is visible, otherwise it will use the data coming from the Kalman Filter Estimation

```
if X is less than PID switch position then
    if The AprilTag can be seen by the camera then
        position data with respect to relative x and y provided by the Marker
         detection are employed
    end
    PID controller application taking as input the Kalman Filter Estimation or
     AprilTag position data
else
    P controller application that will use only data that comes from the Kalman
     Filter Estimation
end
```

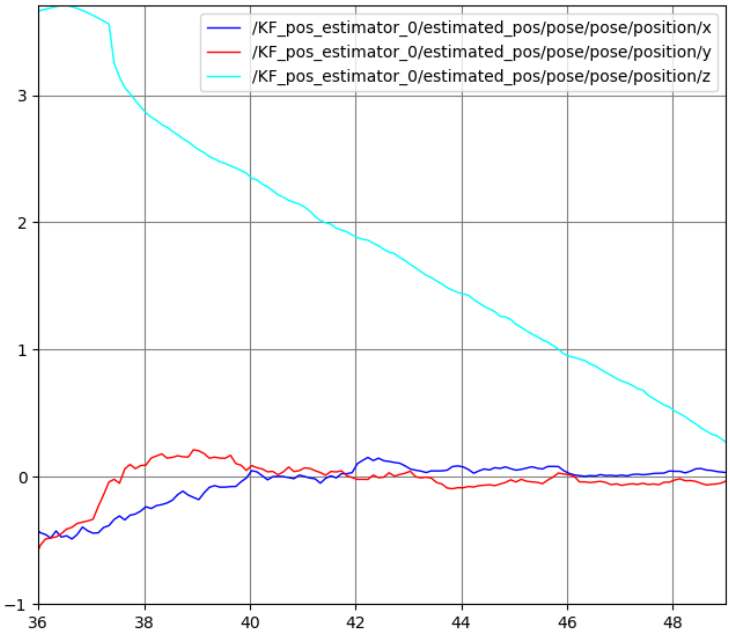
5.3.3 Achievements with rqt demonstrating the benefits of this implementation

This section analyzes the comparison between a solution with only UWB and with the AprilTag implementation. The tool *rqt* is used for analyzing the response over time of the two topics:

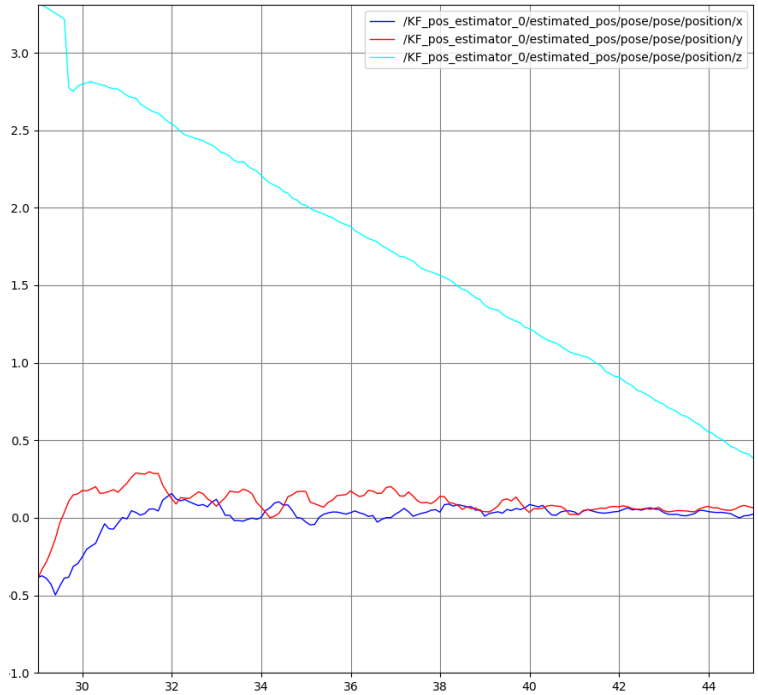
- */KF_pos_estimator_0* referred to the pose/pose/position message in x,y,z. It is useful to display the relative distance between the drone and the rover. This topic will start (and can therefore publish information) at the same time as the estimation algorithms are executed. In the following representations, a start time equal to that at which the AprilTag is detected has been considered. It is possible to visualize its progress over time in the graphs 5.9 and 5.10.

- */AprilTag_estimator/estimated_pose* referred to the x,y,z pose/position message. It is useful to display the relative distance between the AprilTag and the drone. This topic will only start (and can therefore post information) when the AprilTag is visible from the camera. It is possible to visualize its progress over time in the graphs 5.11 and 5.12.

In either situation, the first command of *"takeoff"* is issued and, after a few seconds, that of *"land_on_target"* and an equal-parameter situation was analyzed.
In this case a value of proportional equal to $K_p = 0.6$, integrative equal to $K_i = 0.1$ and derivative equal to $K_d = 0.05$ was used. In the graphs depicted, it is possible

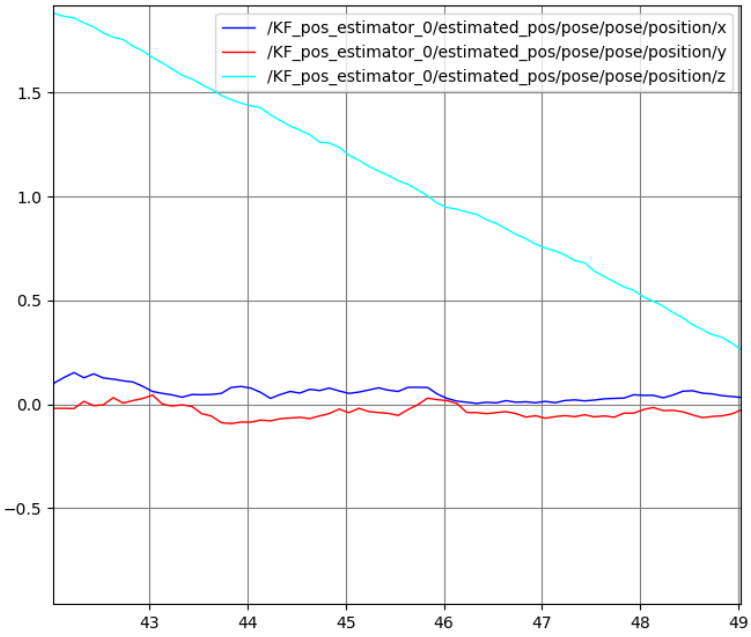


(a) *rqt* with only UWB

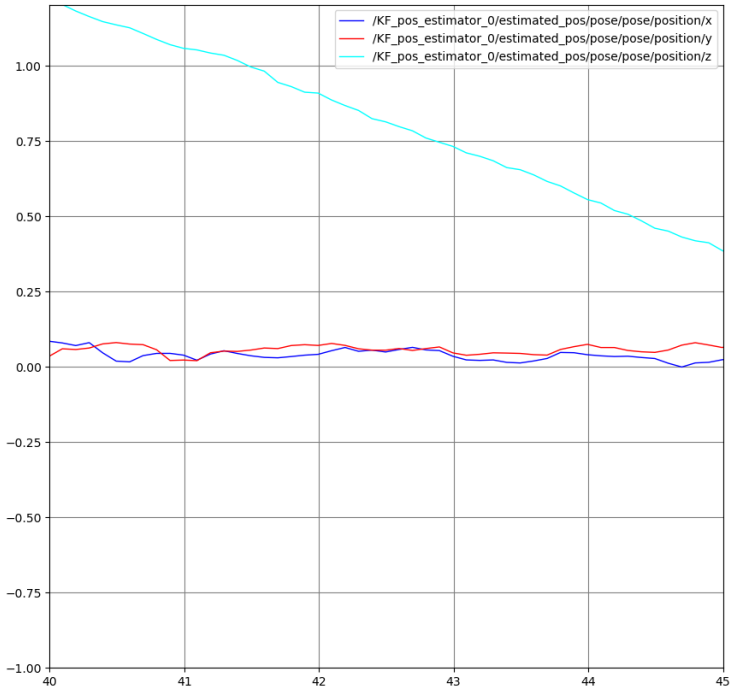


(b) *rqt* with AprilTag implementation

Figure 5.9: *rqt* of "KF_pos_estimator" topic during precision landing phase

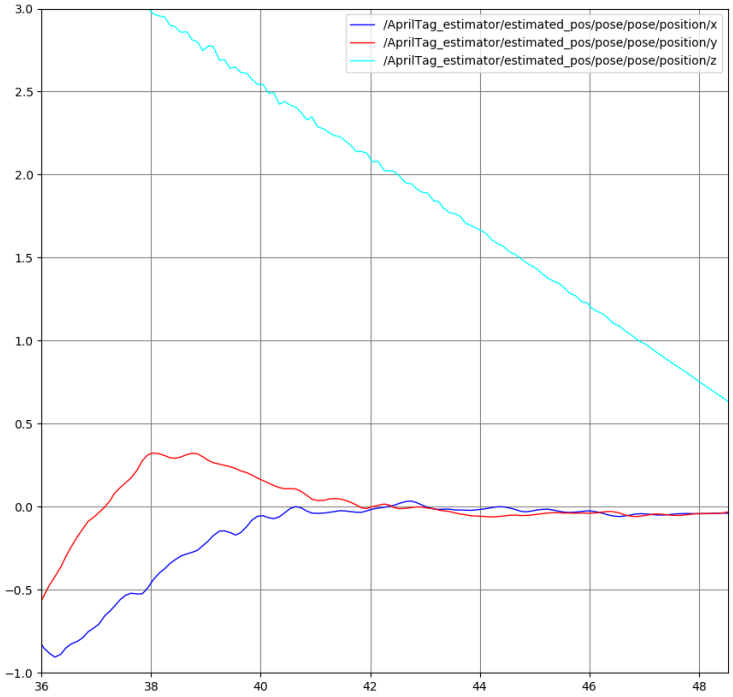


(a) *rqt* with only UWB

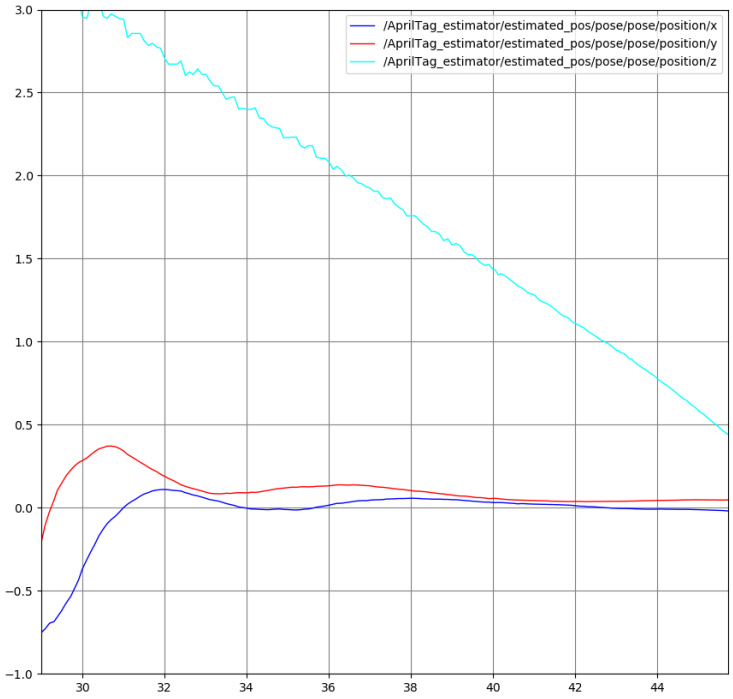


(b) *rqt* with AprilTag implementation

Figure 5.10: *rqt* of "KF_pos_estimator" topic during touch phase

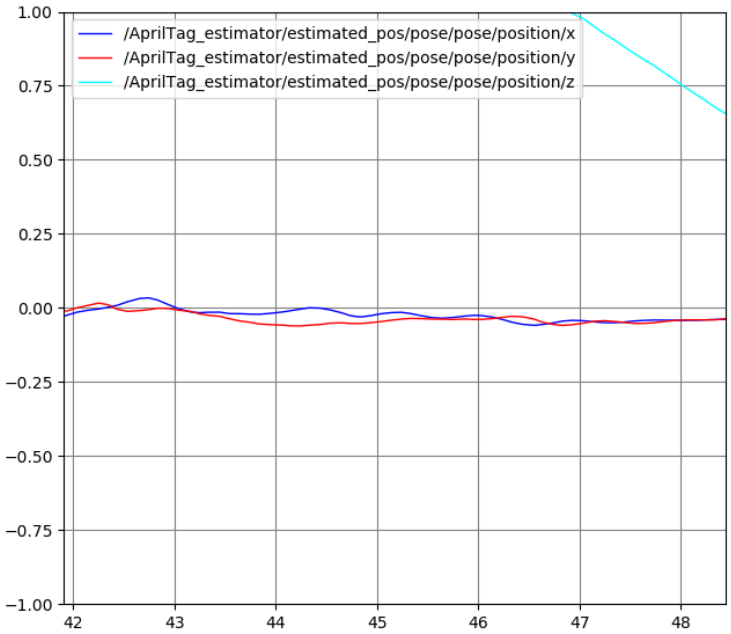


(a) *rqt* with only UWB

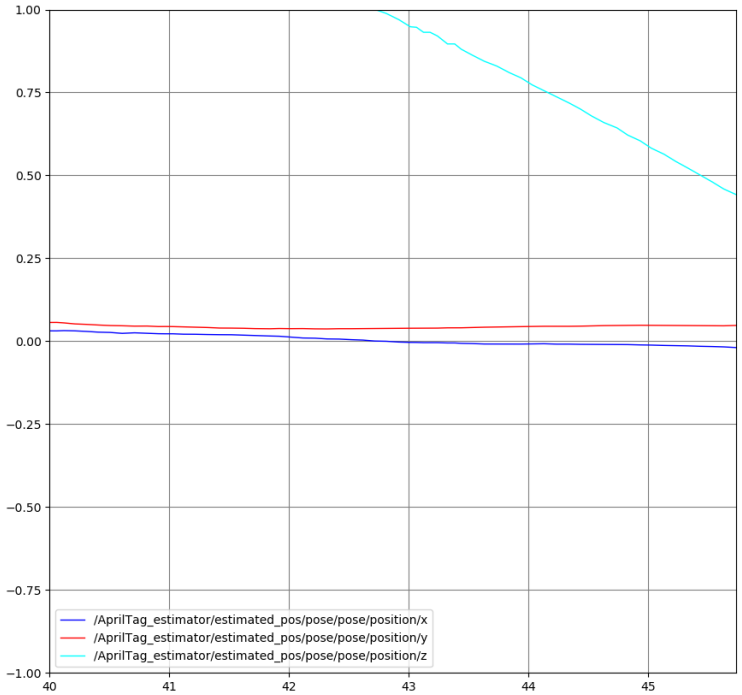


(b) *rqt* with AprilTag implementation

Figure 5.11: *rqt* of "AprilTag_estimator" topic during precision landing phase



(a) *rqt* with only UWB



(b) *rqt* with AprilTag implementation

Figure 5.12: *rqt* of "AprilTag_estimator" topic during touch phase

to visualize the improvement achieved by implementing the AprilTag within the simulation system by graphed on the x-axis the position in meters (m) and on the y-axis the time in seconds (s). In particular, figures 5.9 and 5.11 refer to the entire landing phase, while figures 5.10 and 5.12 focus on the final touch phase. It can be seen that compared to using only UWB, the time to perform the landing increased from 13s to 16s. This is due to the fact that the simulation will take longer to process the data, yet provide a more accurate output. The light blue line in the graphs represents the position of the drone relative to the ground, and it is clearly visible as it descends over time to a position of about 0.45m (the height of the rover plus the distance between the camera and the landing platform). As for the position along the x-axis (the blue line) and y-axis (the red line), a very low tolerance of about 5cm is achieved in both cases at steady state, but with the implementation of the AprilTag a much cleaner and more accurate trend is achieved. The most distinguishing characteristic of the two solutions are the oscillations generated to reach the target which, in the case of using only UWB, will be definitely more evident due to the non-receipt of the AprilTag data definitely useful to improve the overall stability. To improve the data obtained with UWB one would have to increase the value of the derivative K_d quite a bit, which would damp the oscillations. However, this choice might be inconvenient in a real application because the derivative action is very sensitive to external disturbances, affecting its use. It is for this reason that low-pass filters are often applied, but these tend to work opposite to the K_d action. Otherwise, for the purpose of this project, it was chosen to keep a very low derivative value to avoid an overly complex controller design or unpleasant drawbacks. The integral action given by the K_i is particularly important as it ensures that the error tends to zero at steady state, which is why a higher value was chosen. However, the tuning must be done properly since the calibration is related to the overshoots that can be initiated and such sudden changes in load could lead the system towards instability. In addition, an anti-wind-up filter has been included that blocks the action of the integrator in case the velocity increases too much, this is to avoid the triggering of nonlinear phenomena that were not modeled in the simulation and could generate errors. In the figure 5.11 it can be seen that the graph referring to the AprilTag implementation has much fewer oscillations with the same PID parameters, and this is certainly another important advantage. The same can be seen in the figure 5.12, where a significantly more stable position is maintained throughout the descent. The most important value is definitely that of K_p since it must always be present in a controller. This value the more is increased, the more the drone will be ready to respond, however, it is recommended to avoid increasing it too much otherwise the controllability of the drone could be compromised. A very limiting factor up to this point is the use of the same value of proportional both at the beginning of the command *land_on_target* and at the beginning of the *touch* phase, and for this

reason, an adaptive controller explained in 5.5 was used next.
Introducing the Martian environment results in a response that is slightly higher due to the particular ground. Despite this, the final accuracy remained the same and therefore, validating the effectiveness of the algorithm.

5.4 Apriltag implementation in the Kalman filter

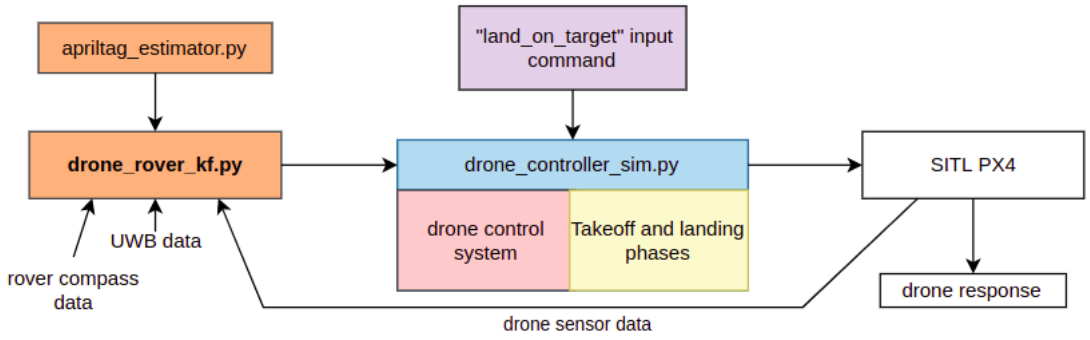


Figure 5.13: Apriltag implementation in the Kalman filter algorithm

In the diagram 5.13 is shown the addition of the code *apriltag_estimator.py*. It can be seen that this will act directly within the KF algorithm so that an extra input data item is used. This implies that, with reference to the Kalman filter theory presented in 4.3.1, the vector x_k must increase in size and, as a result, the sizes of the matrices F_k and Q_k will change (B_k is null). In addition, since additional input data has been added, there will be four different observation models, one for each sensor. It is consequently necessary to construct the vector z_k referring to the AprilTag with the resulting matrices R_k and H_k. Due to the use of *rqt*, it is possible to plot, as a function of time, the data that are posted on the topics related to the AprilTag pose and the relative position between rover and UAV.
An improvement over a solution with only UWB can be visualized here as well, however this solution, in addition to being more difficult to implement, finds it more difficult to be applied on Martian ground. Both graphs depicted in 5.14 were obtained using a completely flat terrain, but if this were to have varying inclinations, as the Martian terrain can be, the camera might struggle more to detect the Marker. These disturbances could then be amplified by the Kalman filter, which could generate a less accurate relative position signal than the original. By using this data directly in the control part and only when this is visible, the relative position will still always be calculated by not considering it, and only when it is clearly visible, it will start generating control signals to better direct the drone to the landing pad.

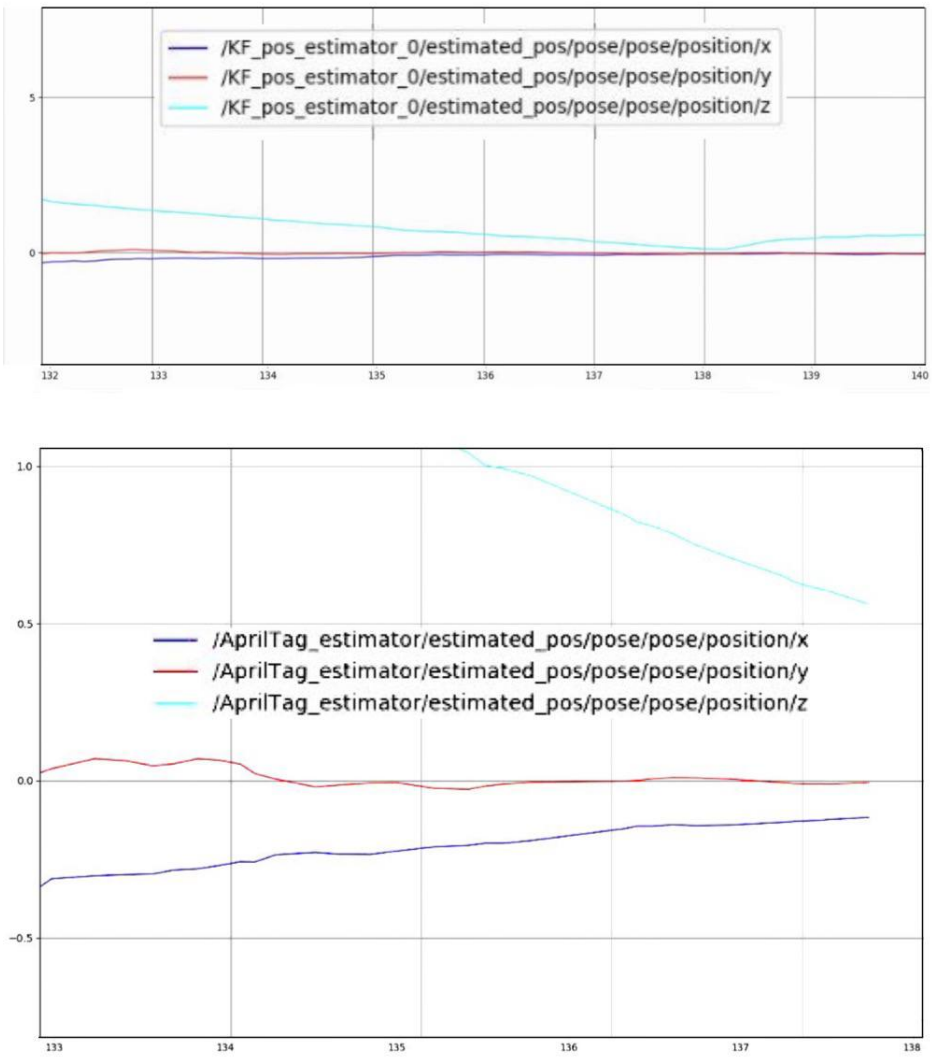


Figure 5.14: *rqt* of "AprilTag position estimator" topic

5.5 Adaptive PID controller

In the previous sections, only the precision landing phase in which the AprilTag is detected was discussed. However, it is also important to analyze the behavior of the drone from the first moment when the command *land_on_target* is given and in detail only the situation of the AprilTag directly acting in the control part will be analyzed 5.7.

Once imparted, the drone will perform a target tracking phase; however, it is

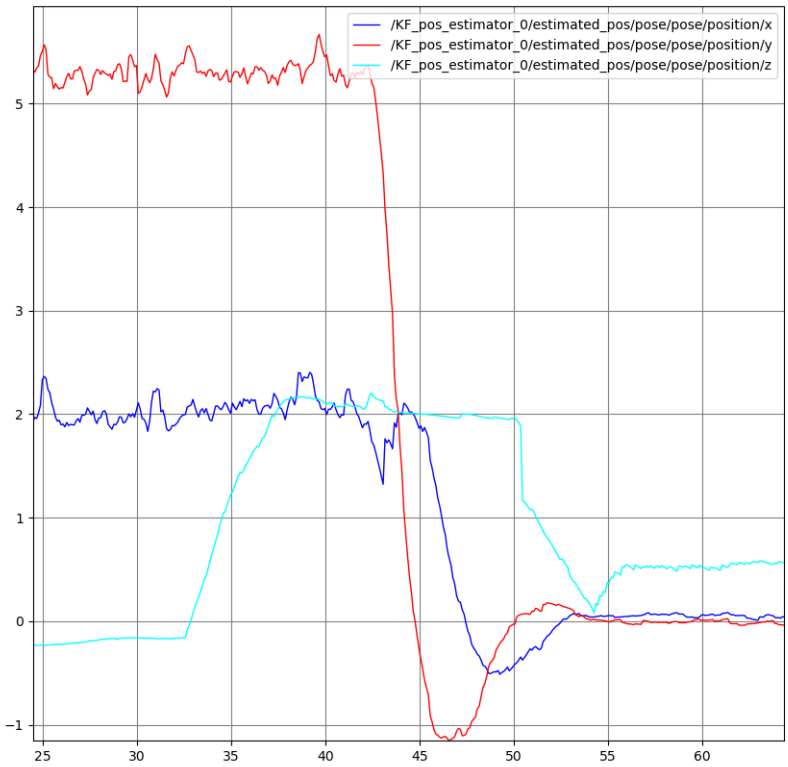
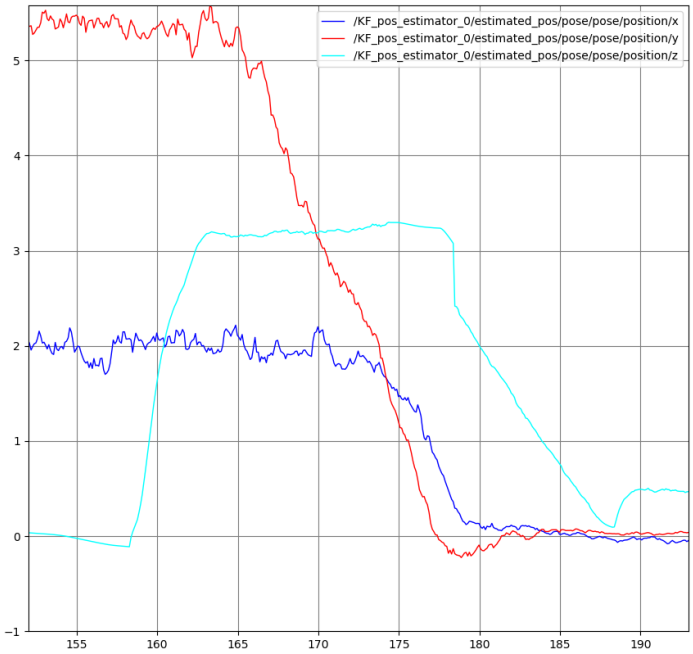
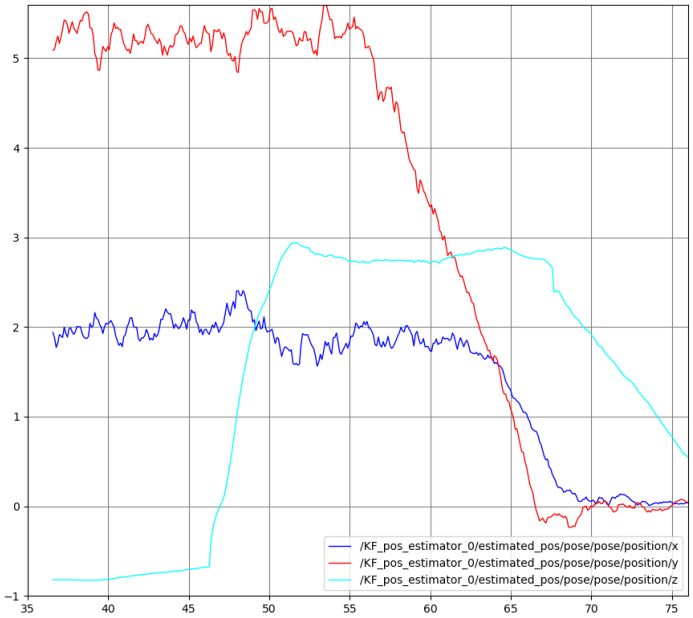


Figure 5.15: *rqt* using only UWB in the whole mission

necessary to vary the values of K_p gradually, so that the drone does not start from the very beginning with a too sudden oscillation that could lead to instability. As seen in the previous section, a value of $K_p = 0.6$ is optimal for performing the touch phase, but this same value would be too high for a tracking phase. For this reason, an adaptive controller was developed that changes the value of the proportional according to the distance from which it is located relative to the target. The principle of operation is based on the relative distance between the drone and the rover, if this is greater than 3 meters then it will use a value of $K_p = 0.1$ and as soon as it reaches this distance every 0.5 meters the value will increase by 0.1, until it reaches the value of $K_p = 0.6$ in the final landing phase. In this way, as visible in the simulation depicted in 5.19, the drone will have a much smoother and more precise behavior. In addition, from the graphs depicted in figure 5.16, it is possible to observe a significantly smoother oscillation in reaching the target, both in an empty environment and in the Martian environment. Focusing more on the position x and y, in the mission using only UWB, the oscillations reach even more than a meter in amplitude, compared to almost zero amplitude oscillation in the adaptive controller.



(a) *rqt* with AprilTag implementation



(b) *rqt* with AprilTag on Martian Ground

Figure 5.16: *rqt* of "KF_pos_estimator" topic during the whole mission

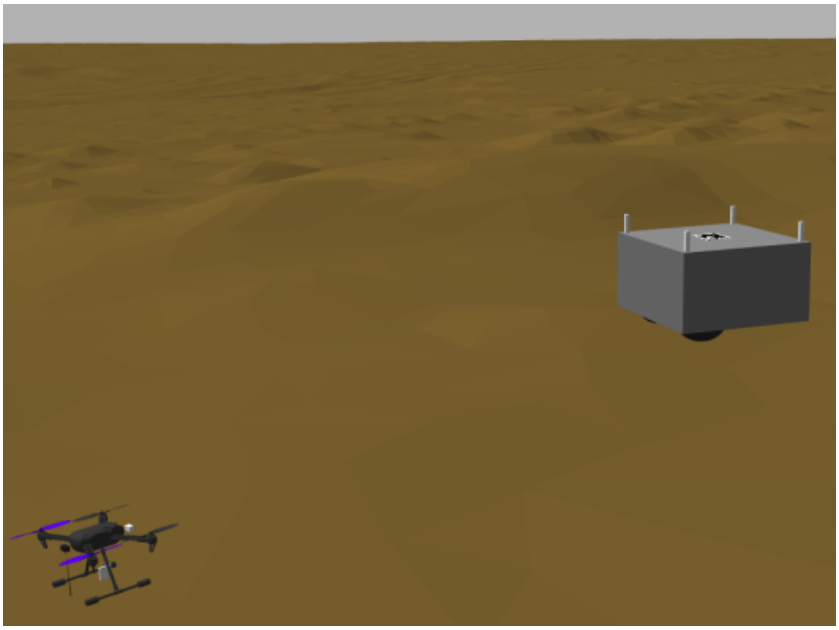
5.5.1 Pseudo-code for adaptive PID controller

Algorithm 2 Adaptive PID controller algorithm

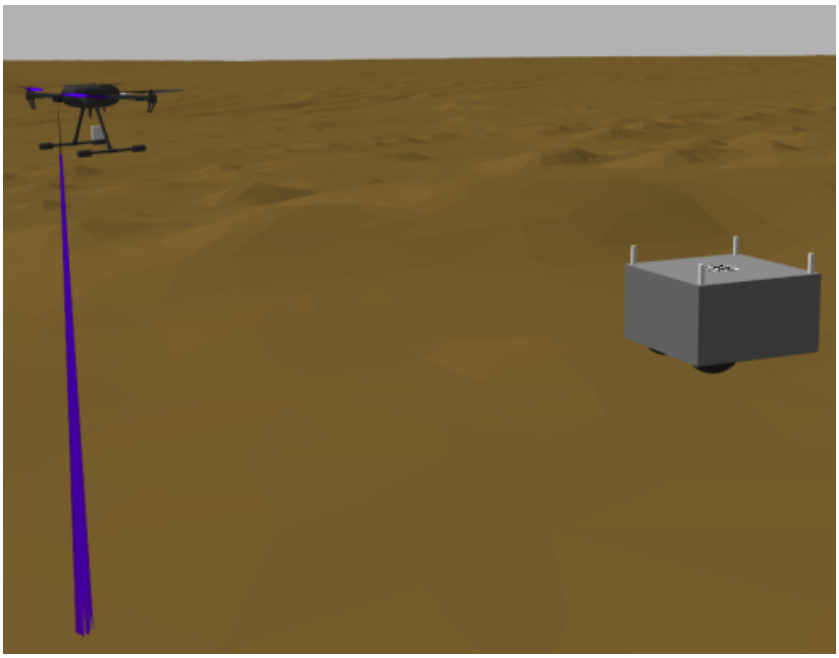
Data: The relative position between UAV and rover = X
Result: UAV will use a PID controller with different values depending on X

```
if X is more than 3 m then
    Use P controller with K_p = 0.1
else
    if X is more than 2.5m and X is less than 3m then
        Use P controller with K_p = 0.2
    else
        if X is more than 2m and X is less than 2.5m then
            Use P controller with K_p = 0.3
        else
            if X is more than 1.5m and X is less than 2m then
                Use P controller with K_p = 0.4
            else
                if X is more than 1m and X is less than 1.5m then
                    Use P controller with K_p = 0.5
                else
                    if The AprilTag can be seen by the camera then
                        position data with respect to relative x and y provided by the
                         Marker detection are employed
                    end
                    PID controller application taking as input the Kalman Filter
                     Estimation or AprilTag position data
                end
            end
        end
    end
end
```

5.6 Pictures of the full simulation



(a) beginning of the simulation



(b) takeoff phase

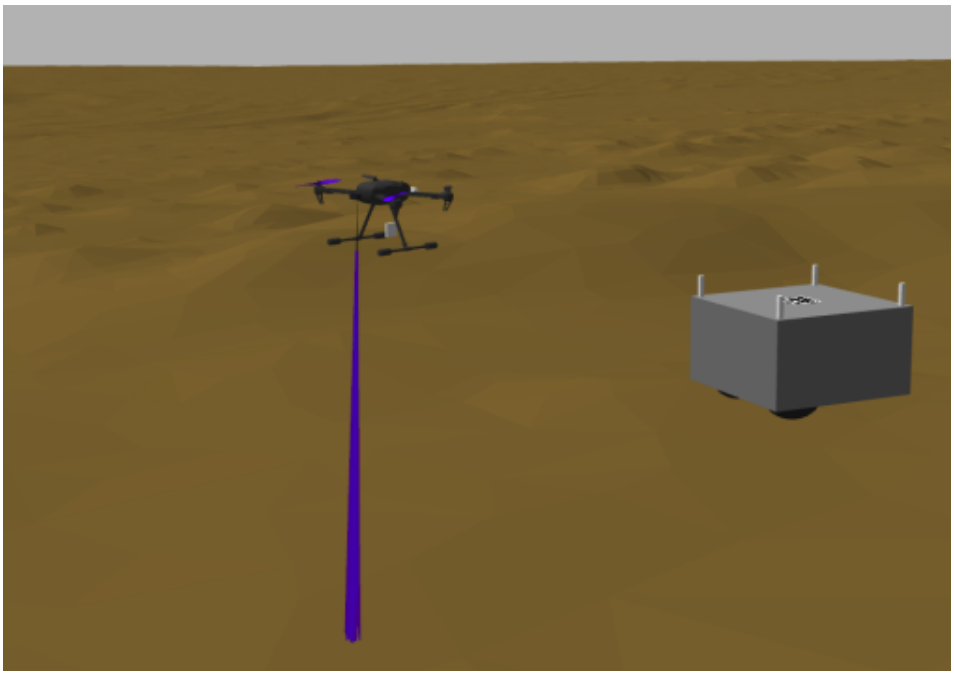


Figure 5.17: Landing phase start with $K_p = 0.1$

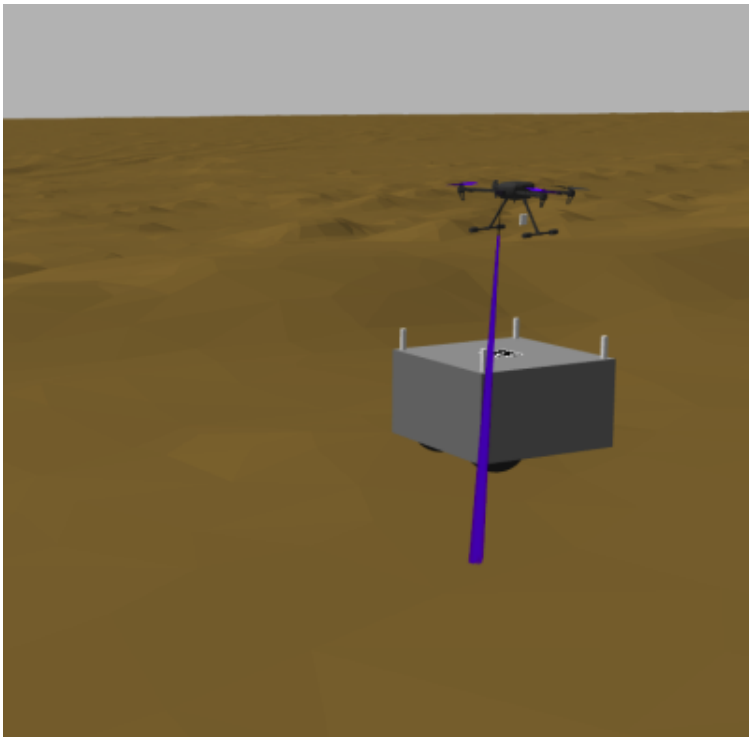


Figure 5.18: Landing phase with $K_p = 0.4$

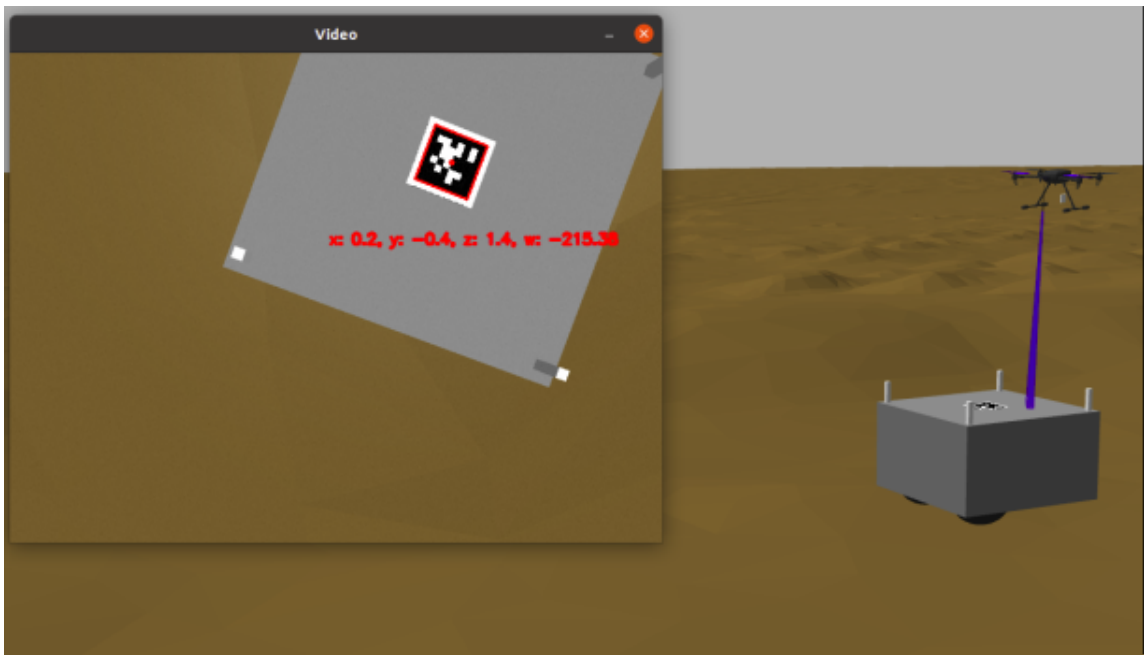


Figure 5.19: Landing phase with $K_p = 0.6$

Figure 5.20: UAV close enough to use PID Controller

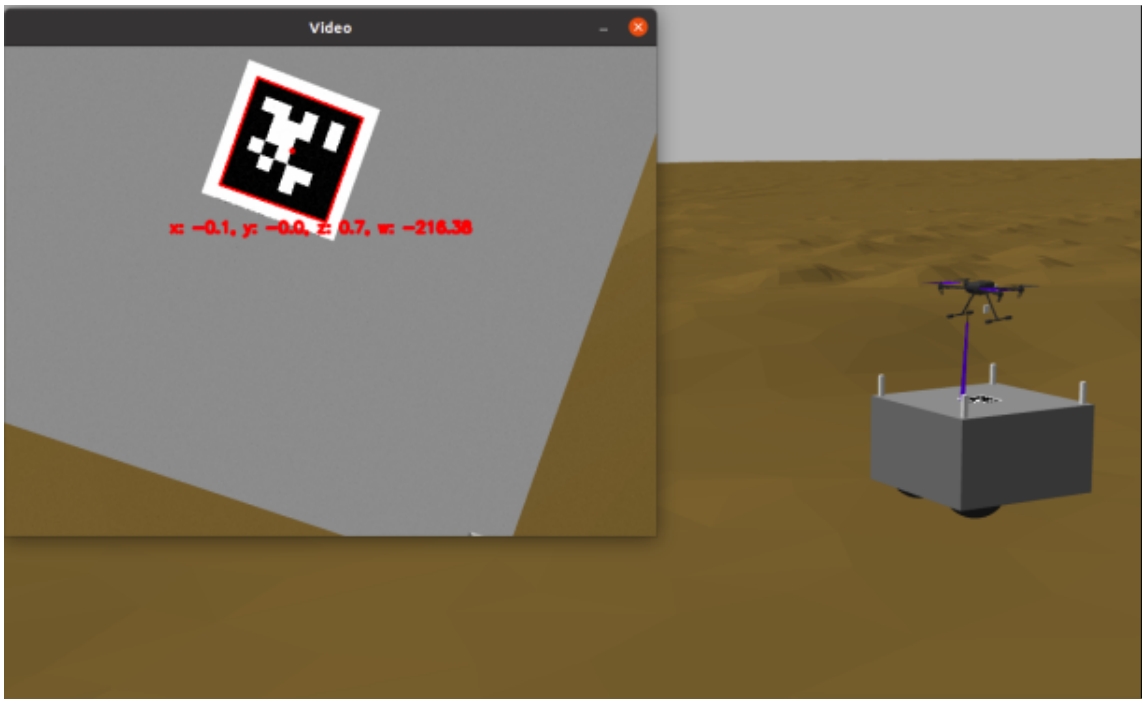


Figure 5.21: Touch phase using the AprilTag detection data

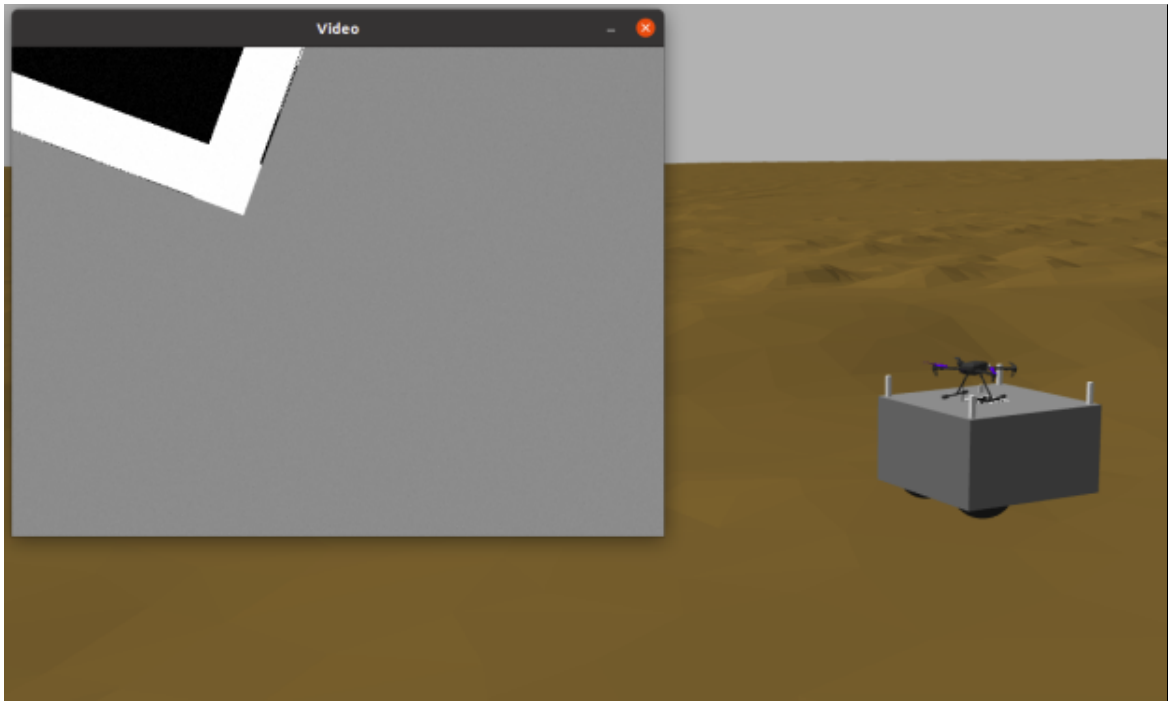


Figure 5.22: End of the landing phase with shutdown of the motors

Chapter 6

Hardware architecture

The drone represented in simulation is slightly different from the real drone represented in this chapter. This is because in simulation it is not as important to faithfully reproduce the entire structure of the drone as it is to correctly implement all the various useful sensors in order to correctly simulate the behavior according to a software point of view.

6.1 Existing HW description of the drone

The drone, which was also used for testing in the chapter 7, uses an X500 frame kit from Holybro, which already includes several components as a base, which can be seen in the figure 6.1.
It includes:

- Pixhawk 4 autopilot (where PX4 software is implemented)
- All the frame kit: main board, backing board, carbon fiber tubes, landing gear
- Brushless motors 2216 (22mm stator diameter and 16mm stator thickness) with 880KV (RPM as a function of the supply volts)
- Propellers 1045 (characteristics given by the type of engine to maximize efficiency)
- Eletronic Speed Controller which connect the flight controller with the motors by allowing the speed regulation of the motors.
- PM 07, a power distribution board that delivers information to the autopilot regarding the battery's voltage and current delivered to the flight controller and the motors in addition to giving Pixhawk 4 and the ESCs controlled power.

Figure 6.1: X500 Kit Holybro

- Battery strap (in order to keep the battery fixed)
- Power and Radio Cables in order to connect the remote control
- Telemetry radio for Ground Control Station connection
- Pixhawk 4 GPS (not used in this thesis work)

This kit as presented thus allows for control of the drone only in manual mode. In order, therefore, to be able to use the algorithms presented in the previous chapters to ensure autonomous flight, it was necessary to introduce another key electronic component: the 8 GB Raspberry pi 4, shown in the figure 6.2.

On it was installed the ubuntu server operating system on which ROS was deployed. Within the raspberry, the entire workspace used in simulation was then downloaded and implemented except for Gazebo, which will obviously no longer be needed. In addition, in the specific case of this thesis work, it was necessary to introduce a UWB module configured as a tag (same type depicted in the figure 5.4) and a Raspberry Pi Camera.

Figure 6.2: Raspberry pi 4 with 8 GB of RAM

6.2 Camera implementation

The mono-camera is essential to be able to correctly execute the algorithms related to AprilTag introduction. In fact, it is responsible for the AprilTag detection and without that, it would be impossible to use the system presented in the chapter 5.2. However, research had to be done on the type of camera that best matched the existing hardware.

6.2.1 Camera confrontation

Camera	Infrared	FOV	Focal Length (MM)	Suit for
RPi V2	No	62.2	3.04	Raspberry Pi, Jetson
RPi NoIR V2	Yes	62.2	3.04	Raspberry Pi, Jetson
IMX219-77	No	77	2.96	Jetson Nano
IMX219-77IR	Yes	77	2.96	Jetson Nano
IMX219-120	No	120	1.88	Jetson Nano
IMX219-160	No	160	3.15	Jetson Nano
IMX219-160IR	Yes	160	3.15	Jetson Nano
IMX219-170	No	170	0.87	Jetson Nano
IMX219-200	No	200	3.15	Jetson Nano
IMX219-D160	No	160	3.15	Raspberry Pi, Jetson

Table 6.1: Camera comparison [32]

The table 6.1 shows an overview of raspberry Pi cams, all of which are 8 Mega

Pixels and mount an IMX219 sensor. The main difference is that not all of them are suitable for the Raspberry Pi 8 Gb. In fact, most are only compatible with the Jeston (Nano or later models) which is another electronic board that is often used in order to implement the above autonomous driving algorithms. The choice, therefore, fell on the Raspberry Pi Cam V2 with a 62.2 FOV, this is because too high a value (such as 160) would lead to too high a distortion, thus making the implementation more complicated. Of the two left, the infrared version may also be very useful in a hypothetical Mars mission, however for the purpose of testing on Earth the two are similar.

6.2.2 3D Camera support

In order to implement the camera on the drone, it was necessary to build a 3D support that could interface with the drone. To create it, it was necessary to reproduce the CAD 6.3 of the entire frame so that the overall dimensions and joints could be well evaluated. The main CAD components were online available and, after making the assembly, it was added a battery (schematized with a rectangle) and the support for the mono-camera implementation (the one in green color).

Figure 6.3: CAD of the drone in Fusion 360

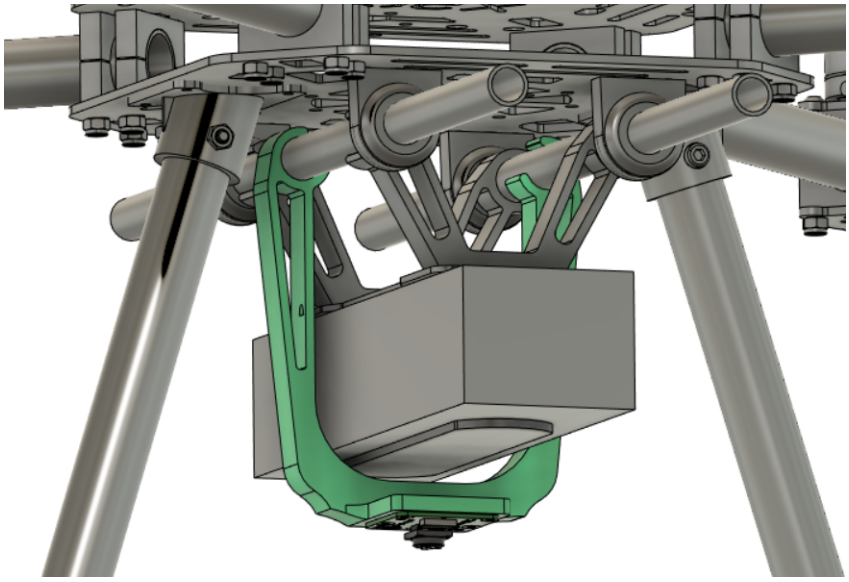


Figure 6.4: Implementation of the support of the camera on the UAV

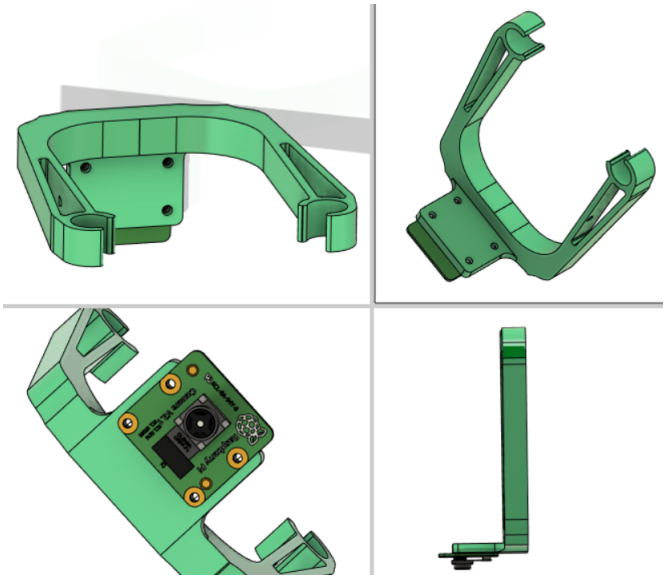


Figure 6.5: RasPi Camera 3D support

There are several reasons for the choice of such a support design including:

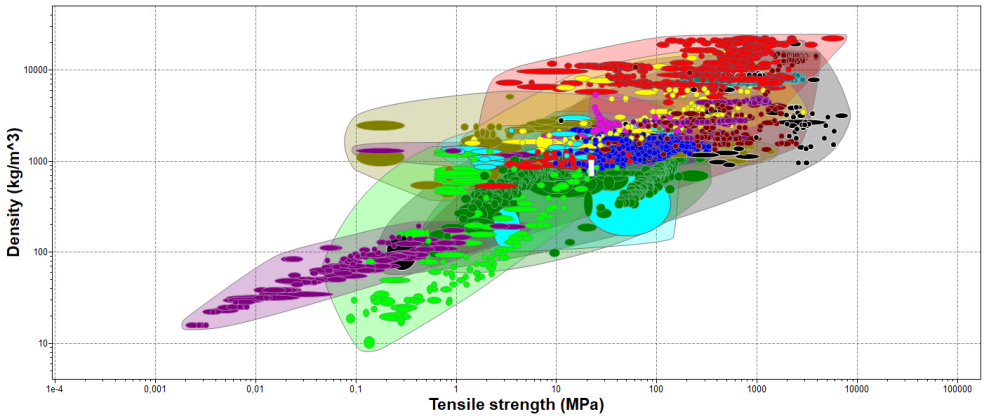
- ease of printing without waste of material (no inclinations below 45°)
- low weight (creation of internal buttonholes to lighten the weight)
- high compatibility with the existing X500 kit through a simple joint

- modularity and maintainability
- small space requirement
- ease of operation on the flat cable connecting the camera and the Raspberry

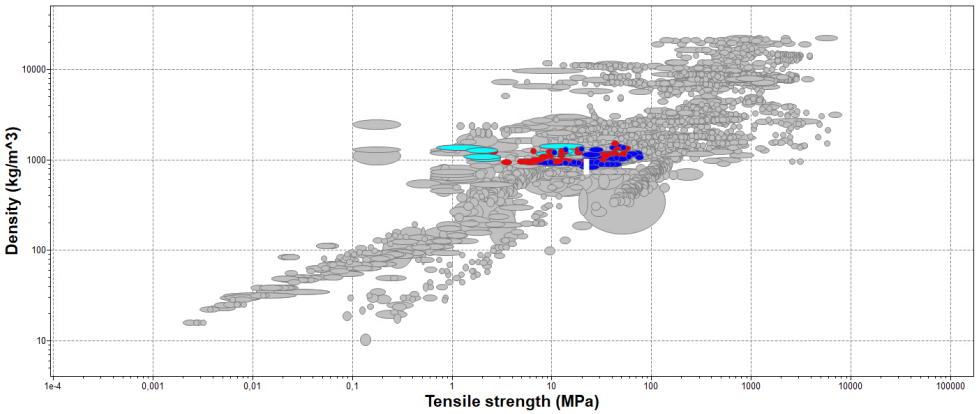
6.2.3 Choice of material for 3D printing

It is very important that the component be functional, but also that it be robust. To figure out which material would best suit this application, the GRANTA EDU Pack software was used. Thanks to this there is a very large library available, and by going and entering appropriate limits, it is possible to choose the one that best fits that application. For these applications, fast prototyping is always necessary, which is well associated with the use of 3D printing by FDM (Fused Deposition Modeling) technique. Once then the library was reduced to only those plastics that can be used by this technology, a graph was plotted to compare a mechanical characteristic, tensile strength (MPa), positioned on the x-axis and, a physical characteristic, density kg/m^3. The reason for this choice is that thus it can immediately perceived which material can resist the most while maintaining a low weight. The lower is the value of the latter, the more several advantages will be gained including increased flight range and a smaller overall space requirement. Lastly, there is a constraint on the minimum temperature of 80° that they can resist, since these are to be items mounted on a drone that will have to fly outdoors. Proceeding by steps, the following graphs were then generated 6.6, where the colors in the first graph refer to:

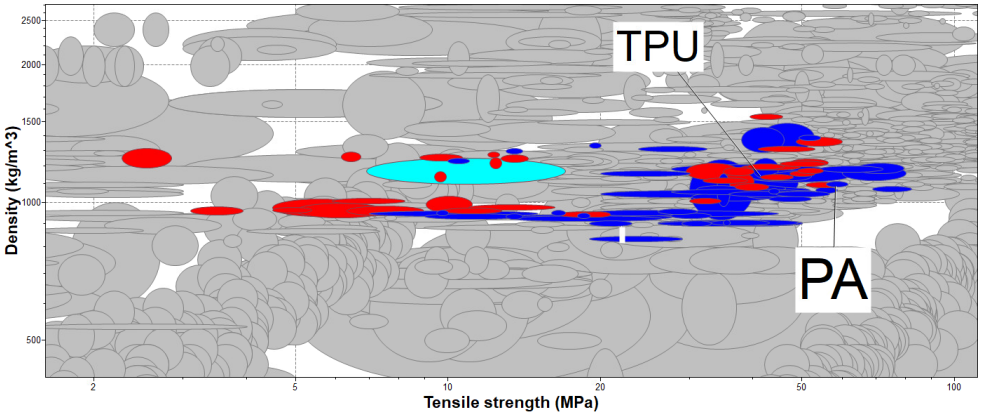
- Yellow: Ceramics and glasses
- Black: Fibers and particulates
- Green: Hybrids (composites, honeycombs, etc.)
- White: Liquids and gases
- Grey: Magnetic Materials
- Red: Metals and alloys
- Purple: Polymers (Elastomers and plastics)
  - Red: Thermoplastic elastomers (TPE)
  - light blue: Thermoset elastomers (rubber)
  - blue: Thermoplastics



(a) Material Universe



(b) Polymer FDM extrusion selection



(c) Best Materials for camera holder

Figure 6.6: Granta EDUPack Database for the choice of material

As it is possible to see in the last graph in reference to 6.6, the most suitable materials are TPU (Thermoplastic polyurethane) and PA (Polyamide Nylon). The former is less strong and heavier (48 Mpa tensile strength and 1200 kg/m^3 density), but it can dampen vibrations much better. The latter, on the other hand, is much stronger and less heavy (52 MPA tensile strength and 1000 kg/m^3 density). Also considering an economic factor, PA is definitely more expensive and is more difficult to print since it is necessary to use a steel nozzle (and replace the classic brass ones) while maintaining an extrusion temperature of about 280°, a support plate heated to about 100°, and using an enclosure box). TPU is, therefore, better in such an application, since the component is not of extreme structural importance and the aspect of vibration resistance plays a more important role especially when signals have to be sent and received from a small camera. The characteristics mentioned above may then vary slightly depending on the brand of the material, however, to perform initial printing trials, it is convenient to use an additional type of material which is PLA (polylactic acid) and it is even easier to print than TPU and it is significantly cheaper (so perfect for prototyping).

6.2.4 Slicing software and printer settings

Once the 3D model has been created, a mesh of the model can be generated and exported in *.stl* format, still using the Fusion 360 software. It will be legible by the slicing software, which will generate a GCode that can be read by the 3D printer via a microSD/USB. In this case, "Prusaslicer" software was used and the component was printed with a Prusa mk3s+ printer, using a 0.6mm nozzle and a layers height of 0.4mm. This choice is due to the fact that this component has a mechanical and not an aesthetic functionality, consequently decreasing the total number of layers that compose the part will increase the strength and decrease the printing time, at the expense of less definition. From the figure 6.7 it is possible to view some printing characteristics, such as total printing time and grams used. The latter two characteristics range from 30 minutes (for PLA and PA) to 2 hours (for TPU) of printing time and a weight ranging from 14 to 17 grams and these properties are highly dependent on the internal infill, which was set at a value of 30 percent and with a honeycomb pattern (as visible in the figure 6.8). Finally, positioning the component in this printing direction has several advantages as there are no cantilevered components that require support with additional material and the fibers are arranged in the same direction as the load, thus increasing their resistance. The only disadvantage of this positioning is the fact that the holes for fixing the camera will be slightly oval, but in this case it does not represent a problem as the holes have been specially designed larger in order to easily insert the screw that can be locked with a self-locking nut.

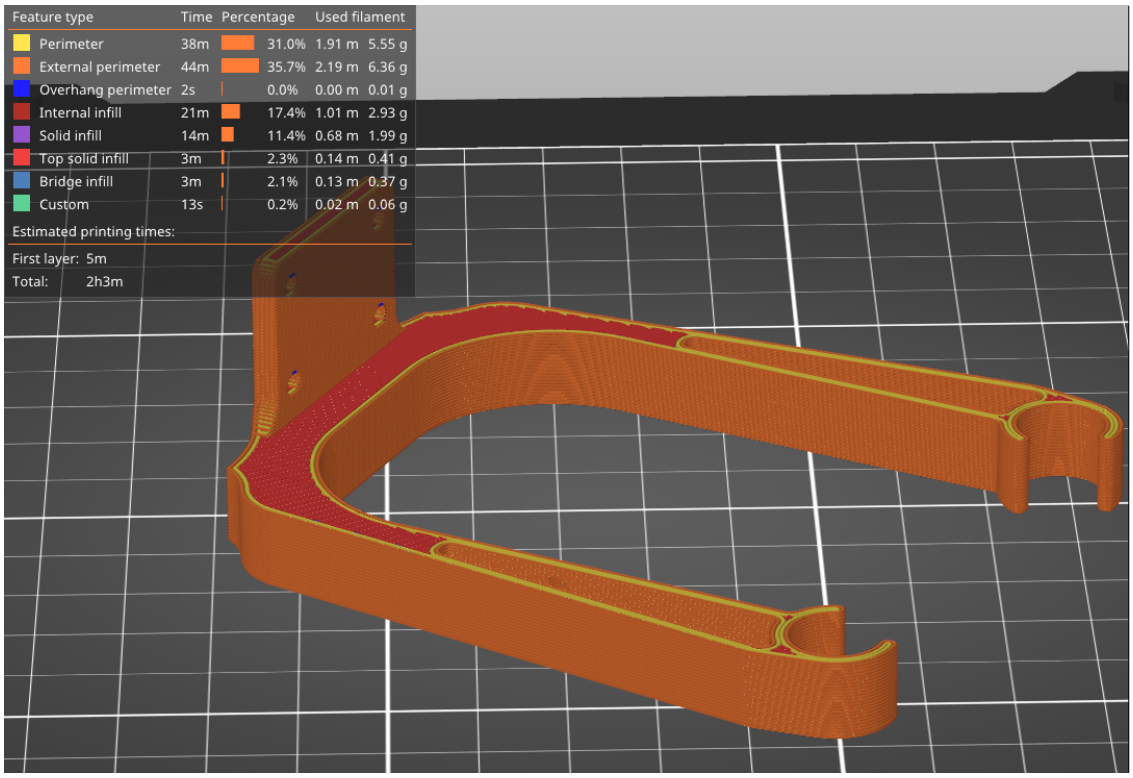


Figure 6.7: Slicing of the 3D model

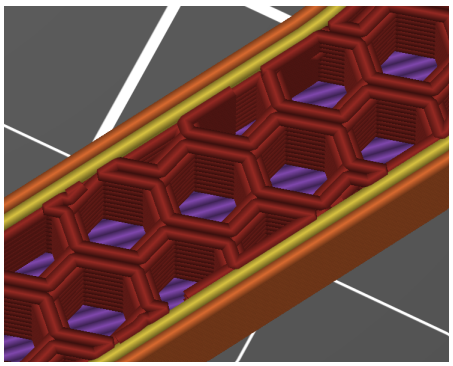


Figure 6.8: Honeycomb internal fill

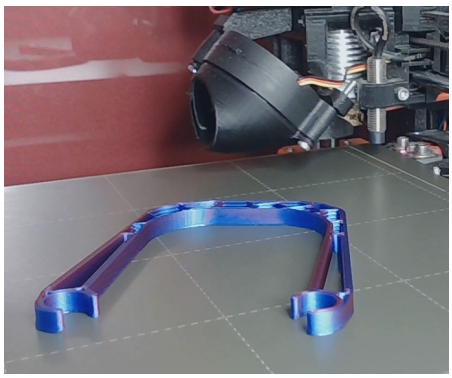


Figure 6.9: 3D printer during the extrusion of material

Chapter 7

Test

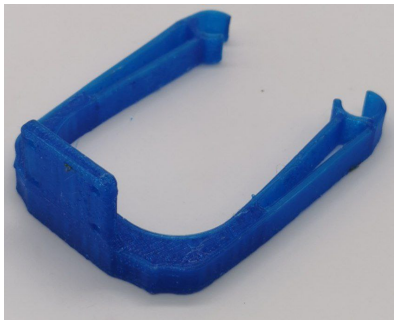
7.1 Implementation of 3D printed support on the UAV

Before proceeding to tests where it is possible to replicate exactly what was seen in the previous chapters in simulation, the hardware environment must be configured correctly. In fact, now it will no longer be necessary to use Gazebo and therefore all the plugins that were simulated are now real. In the context of this thesis work, the substantial change was related to the introduction of the camera. Based on the considerations made in Chapter 6 the support for the camera was printed in PLA as the first iteration and visible in the figure 7.1 and it was integrated on the real drone 7.3. After testing the right tolerances and space requirements, a TPU version, which is the best for this type of application, was upgraded 7.2.

(a) **(b)**

Figure 7.1: 3D printed camera support in PLA

(a) black clamp to show the flexibility



(b)

Figure 7.2: 3D printed camera support in TPU

Figure 7.3: Camera implementation on the real UAV

7.2 Camera implementation test for AprilTag detection

The next step was to connect the Raspberry pi 4 8 GB (with ROS inside) to the personal computer via SSH protocol and verify the operation of the camera 7.4. It was possible, thanks to an open-source package, to implement a node that would be responsible for publishing the camera data to a topic called *image_raw*. It was then necessary to modify the python script *apriltag_estimator.py* in a way that this node could subscribe to this new topic. Before launching the node related to detection, it was also necessary to download the Open CV libraries (so that the on-screen of detection could be opened just as in simulation) and to consider that the camera's raw image does not represent the true view from the ideal camera so it needed to be changed according to two main parameters:

- **Intrinsic parameters**: the one represented in table 6.1 which remain constant for a given camera.
- **Distortion parameters**: these depend on the fact that light rays create image distortion especially at the corners of a curved lens. These therefore need to be corrected by a calibration action for each camera that is to be implemented.

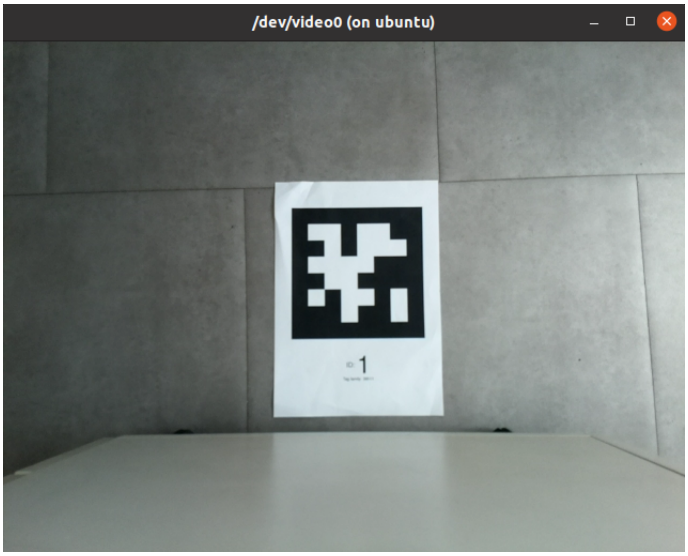


Figure 7.4: Testing general operation of the camera

In order to perform the camera calibration, it was necessary to simply install a tool directly on the Raspberry and print in 2D a 6x8 checkerboard (only the inner vertices need to be counted) with the squares 0.035 meters wide.

```
ubuntu@ubuntu:~$ ros2 run camera_calibration cameracalibrator --size 6x8 --square 0.035
-ros-args -r image:=/image_raw -p camera:=/my_camera
Waiting for service /my_camera/set_camera_info ...
OK
Waiting for service left_camera/set_camera_info ...
OK
Waiting for service right_camera/set_camera_info ...
OK
*** Added sample 1, p_x = 0.388, p_y = 0.851, p_size = 0.247, skew = 0.261
*** Added sample 2, p_x = 0.395, p_y = 0.731, p_size = 0.212, skew = 0.144
*** Added sample 3, p_x = 0.446, p_y = 0.695, p_size = 0.207, skew = 0.026
```

Figure 7.5: Running the ROS command to open calculate distortion parameters

Once the dedicated command 7.5 is given, the GUI opens automatically and will begin to do a detection of the squares using the colors of the rainbow as can be seen in figure 7.6.

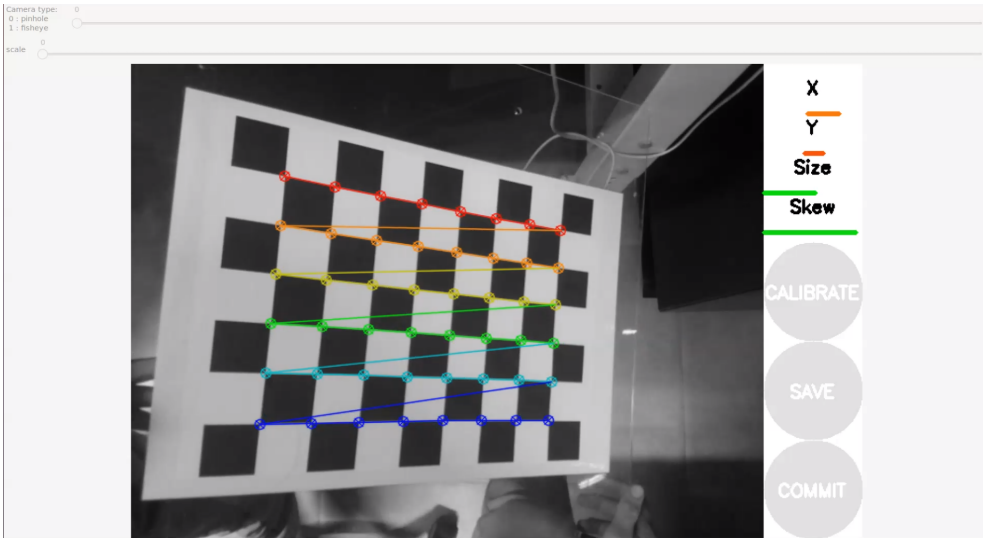


Figure 7.6: Graphical tool interface to calibrate the camera

The camera will have to be kept in a fixed position, while the checkerboard will be placed in more than twenty different positions so that all the various parameters can be calculated correctly. On the right side a feedback on the progress of this operation can be displayed, and only when the values of "X", "Y", "$Size$"and"$Skew$" are all in green color can the calibration button be clicked 7.7 and after some minutes, it is possible to click to the "save" button to be able to store all the parameters.
In the specific case of the camera that was used (Raspberry pi cam V2), the following distortion parameters were obtained:

$$camera\_params = [504.579158, 506.433694, 328.458887, 241.522065] \quad (7.1)$$

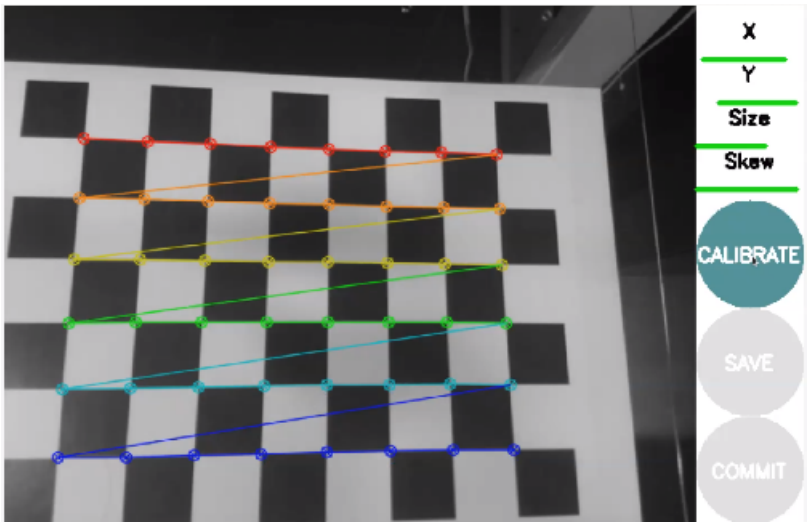


Figure 7.7: Calibration phase completion

These were included in the script *apriltag_estimator.py* and finally was able to proceed with the execution of the relevant node, giving in output what is visible in the figure 7.8.

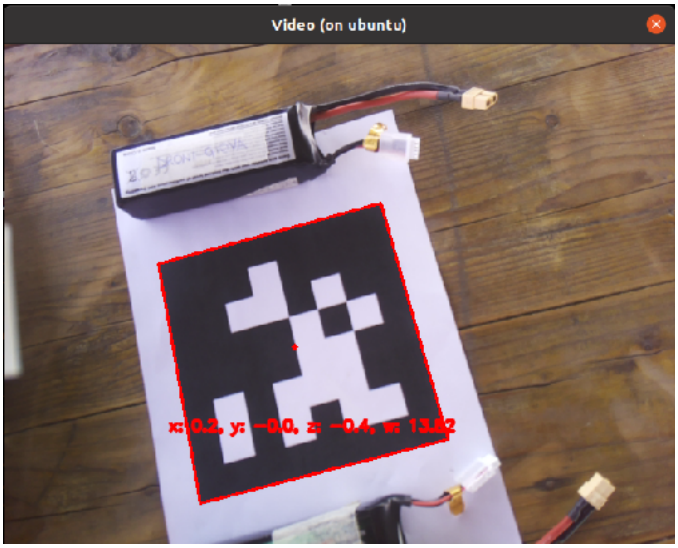


Figure 7.8: AprilTag detection on a real UAV

With the algorithm's operation established, in order to test the entire landing system, it was necessary to re-launch the same commands as in the simulation, but applying the same corrections on the plugins that were made on the camera. The landing platform was implemented on a Husky UGV (Unmanned Ground Vehicle)

and four UWB modules set as "anchors" were inserted at the corners. On the drone side, the same module was implemented, but set as a "tag". In the figure 7.9 it is possible to see a preparatory phase to test all the various algorithms that allow the completion of the whole mission. However, since the real system is very complex, several more variables have to be considered compared to the simulated version that require longer time in order to be implemented.

Figure 7.9: Preparatory phase for testing the precision landing algorithm at the flying field

Chapter 8

Conclusions and future works

The work in this thesis has demonstrated the effectiveness of a precision landing algorithm by exploiting a technology based on UWB and AprilTag by implementing an adaptive PID controller that allows achieving an accuracy of about 5cm (in SITL simulation) and a very smooth and precise landing trajectory using low values of K_p, K_i, K_d that in real world could lead to anomalous behaviors. This solution can be useful in many different environments, especially where there is a lack of GPS signal. The latter feature penalizes this system greatly from a range point of view, but it gains in accuracy and flexibility. However, in the near future, in case a network of satellites can be established on Mars as well, it might also be possible to consider exploiting the data from them for navigation. In addition, an EKF or UKF could be developed to have an even more precise estimation, thus also taking into account nonlinear phenomena.

This thesis thus presents another starting point for developing even more robust and effective systems that could be even better expendable in an eventual Mars-like environment. The beginning of testing has brought the onset of numerous problems that were not apparent in simulation. Once these algorithms are well tested in the terrestrial environment, it can be considered modifying the drone's characteristics to make them more similar to a flight in the Martian atmosphere, as it has been done, for example, for the Mars Helicopter Scout (best known as Ingenuity). The latter is in fact extremely light and has much wider rotors than the usual UAVs since, given the more rarefied atmosphere, they have to rotate much faster to get the same thrust (even though Mars' gravity is smaller than Earth's).

As for The Controller, the adaptive PID presented in this thesis proved to be definitely efficient, however, it could be thought of using a polynomial function to follow the trajectory and make tracking or precision landing even more effective, avoiding the algorithm presented in 2 that might be less effective in real flight. In addition, the implementation of an artificial neural network in simulation could be useful so that the drone could learn on its own, after performing a reinforcement learning phase, which type of flight controller best optimize that particular mission phase.

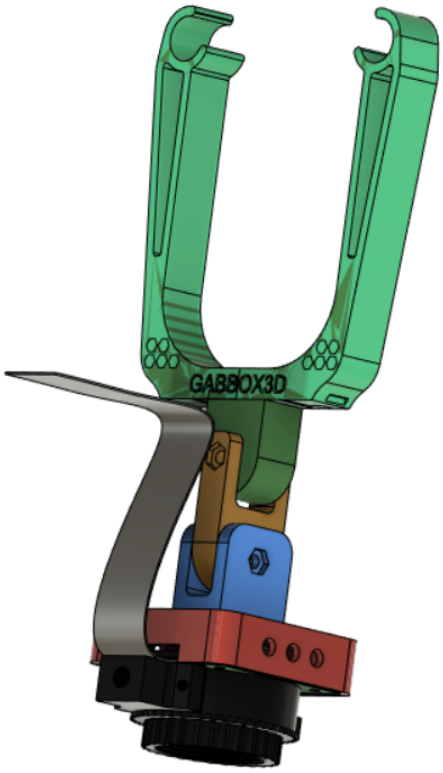


Figure 8.1: CAD of Raspberry HQ Camera

On the hardware side, it could be considered either to implement a more powerful electronic board, such as a Jetson Xavier, or to implement a different camera through more expensive solutions (such as the Intel® RealSense™ Tracking Camera T265). The latter camera could be used not only for Marker detection, but the sensors implemented on it could also provide data that might be used as input into a more sophisticated filtering system to provide even more accurate output than the ones outlined in this thesis. A more imminent solution may instead be to use the new Raspberry Pi Cam HQ, which has a higher definition and thus could

further improve the AprilTag detection phase. In addition, a support to be 3D printed that allows two-axis motion has been made in Fusion 360 and depicted in figure 8.1. It is already prepared to interface with the X500 drone presented in the chapter 6 again through a simple interlocking and thus without having to assemble or disassemble components from the existing drone. The future idea would be to connect two servomotors to the raspberry, which through algorithms can autonomously control the movement of the 3D printed plastic parts (and so the HQ camera) and be able to use it both to improve the detection phase of the AprilTag and for other purposes.

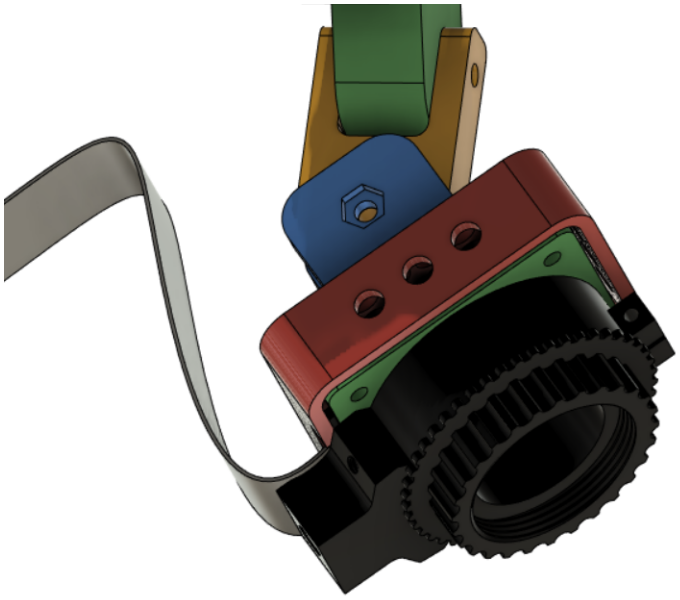


Figure 8.2: Movement of the two axis of the camera support

Bibliography

[1] Stefano Primatesta. «ROS Introduction and basic concepts». In: () (cit. on pp. 3–6).

[2] Lorenzo Croccolino. *La piattaforma ROS per lo sviluppo di applicazioni per la robotica: panoramica e caso di studio*. URL: https://amslaurea.unibo.it/19755/1/Tesi_ROS_CT.pdf (cit. on p. 4).

[3] URL: https://wiki.ros.org/Nodes (cit. on p. 4).

[4] URL: https://wiki.ros.org/Topics (cit. on p. 4).

[5] URL: https://wiki.ros.org/Messages (cit. on p. 4).

[6] URL: https://wiki.ros.org/Master (cit. on p. 4).

[7] URL: https://wiki.ros.org/Services (cit. on p. 4).

[8] URL: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html (cit. on p. 5).

[9] URL: https://wiki.ros.org/rqt_graph (cit. on p. 5).

[10] URL: https://docs.px4.io/main/en/simulation/ (cit. on p. 7).

[11] URL: https://roboticsknowledgebase.com/wiki/tools/gazebo-simulation/ (cit. on p. 8).

[12] URL: https://www.linuxadictos.com/it/robot-simulatore-di-gazebo.html (cit. on p. 9).

[13] Lorenzo Galtarossa. «Obstacle Avoidance Algorithms for Autonomous Navigation system in Unstructured Indoor areas». Master's degree thesis. Politecnico di Torino, 2018 (cit. on p. 9).

[14] URL: https://www.acsysteme.com/en/customised-service/model-based-design/l (cit. on p. 11).

[15] URL: https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html (cit. on pp. 11, 13).

[16] URL: https://docs.px4.io/main/en/config/airframe.html (cit. on p. 12).

[17] Simone Rapisarda. «ROS-Based Data Structure for Service Robotics Applications». Master's degree thesis. Politecnico di Torino, 2019 (cit. on p. 12).

[18] URL: https://ardupilot.org/dev/docs/using-gazebo-simulator-with-sitl.html#using-gazebo-simulator-with-sitl (cit. on p. 13).

[19] URL: https://docs.px4.io/main/en/simulation/ (cit. on pp. 15, 16).

[20] URL: https://docs.px4.io/main/en/ros/ros2_comm.html (cit. on pp. 15, 16).

[21] Gennaro Scarati. «UAV precise ATOL techniques using UWB technology». Master's degree thesis. Politecnico di Torino, 2021 (cit. on pp. 17, 22).

[22] URL: https://docs.px4.io/main/en/simulation/multi_vehicle_simulation_gazebo.html (cit. on p. 18).

[23] URL: https://en.wikipedia.org/wiki/Kalman_filter (cit. on p. 20).

[24] URL: https://tjosh.medium.com/kalman-filter-predict-measure-update-repeat-20a5e618be66 (cit. on p. 21).

[25] URL: https://it.mathworks.com/matlabcentral/fileexchange/105525-kalman-filter-virtual-lab (cit. on p. 21).

[26] URL: https://en.wikipedia.org/wiki/PID_controller (cit. on pp. 25, 26).

[27] URL: https://docs.px4.io/main/en/advanced_features/precland.html (cit. on p. 28).

[28] Ksenia Shabalina, Artur Sagitov, Leysan Sabirova, Hongbing Li, and Evgeni Magid. «ARTag, AprilTag and CALTag Fiducial Systems Comparison in a Presence of Partial Rotation: Manual and Automated Approaches». In: Jan. 2020, pp. 536–558. ISBN: 978-3-030-11291-2. DOI: 10.1007/978-3-030-11292-9_27 (cit. on pp. 28, 29).

[29] URL: https://www.droneblog.news/rtk-un-sistema-di-posizionamento-utile-per-i-droni/ (cit. on p. 30).

[30] URL: https://shop.holybro.com/h-rtk-f9p-gnss-series_p1226.html (cit. on p. 30).

[31] URL: https://en.wikipedia.org/wiki/Ultra-wideband (cit. on p. 30).

[32] URL: https://www.waveshare.com/wiki/IMX219-77_Camera (cit. on p. 52).