Institut Supérieur de l'Aéronautique et de l'Espace
SUPAERO

Master of Aerospace Engineering Research Project

Autonomous Robotic Aerial Vehicle

S3 Project Report

Author: Ricardo DE OLIVEIRA VALENTE MORENO RODRIGUES

Due Date of Report: 28/03/2021
Actual Submission Date: 28/03/2021

Starting Date of Project: 12/11/2019

Tutors: RaghuVamsi DEEPTHIMAHANTHI, Ahlem MIFDAOUI

Declaration of Authenticity

This assignment is entirely my own work. Quotations from literature are properly indicated with appropriated references in the text. All literature used in this piece of work is indicated in the bibliography placed at the end. I confirm that no sources have been used other than those stated.

I understand that plagiarism (copy without mentioning the reference) is a serious examinations offence that may result in disciplinary action being taken.

Abstract

Unmanned Aerial Vehicles (UAV) possess a vast potential for various autonomous applications such as surveying or delivery, while rovers are often used to collect information about the terrain and take crust samples. However, the combination of both is not a very active field of research. In this report, the problem of creating a fully autonomous hybrid vehicle composed by a flying and a ground module is explored.

Furthermore, when considering autonomous flight or ground operations, one of the most essential requirements certainly is a reliable obstacle avoidance mechanism. An hybrid model will be simulated using the Gazebo simulator and different algorithms will be quantitatively and visually analyzed.

Finally, this report will show the perspectives for the future and how everything that is tested in simulation can be translated to the real vehicle.

Keywords: **Algorithm, Rover, Drone, Simulation, Obstacle Avoidance, Path following**

Contents

1 Introduction

An autonomous vehicle is a machine that uses a fully automated driving system so it can move and guide itself without any human input. Recently, there has been a very strong tendency to develop this type of machines due to its characteristics and wide range of applications.

This project's main objective is to program an Autonomous Robotic Aerial Vehicle, which combines the capabilities of a UAV and a rover, so it can become completely autonomous and independent from any human command, therefore being able to take the required decisions in order to move from an initial to a final location.

Rovers are commonly used in space missions due to their capability of moving in rough terrain. Additionally, these types of robots can also perform tasks in areas with hard accessibility. On the other hand, UAVs have been very popular over the last years due to multiple factors, namely the fact that they do not require a pilot and also because they can be controlled from the ground or even autonomously follow a pre-planned path. This hybrid machine is ideal since it combines the high efficiency of both modules.

To achieve the desired goal, sensors will be used in order to avoid possible obstacles on the way. In case an obstruction is detected, the robot should be able to choose the most appropriate path to surpass the obstacle: either go around it on the ground or cross it in the air by using the flying module. This choice will depend on three parameters: power, time and energy.

In case the rover has not enough power to surpass the obstruction, the flying module takes over. Moreover, the ground module is preferred when it comes to saving energy, as long as the avoidance can be done on the ground. Finally, the flying module will always perform faster than the rover, in case there is a time constraint.

This report describes the main ambitions and tasks of the project as well as all the work that has been performed throughout the second and third semesters.

2 Project definition

2.1 Context and key issues

This work has been previously conducted by Anton Sambalov, Sakshi Chaudhary and Nandhini Raghunathan, who designed and assembled a vehicle composed by a ground and a flying module. Furthermore, Anton Sambalov also worked on the set up of different types of equipment as well as on the communication between them. Posteriorly, some simulations of required algorithms were performed [1]. The main objective was then to give continuity to what had been done before in order to achieve the desired goal.

One of the main issues of this project is the communication part between both the ground and air modules, since there must be an algorithm allowing to choose whether to utilise one or the other depending on the waypoints the vehicle has to follow.

The second key issue is composed by the path planning and obstacle avoidance algorithms, which will play a major role as well, since these will be responsible for the robot's autonomy. The path planning algorithm will be in charge of autonomously creating a path between an initial and a final location. However, that is not enough, since this algorithm does not give the robot the ability to avoid obstacles. Therefore, an obstacle avoidance method will also have to be implemented. An example of the desired behaviour of the vehicle is shown in figure 1.

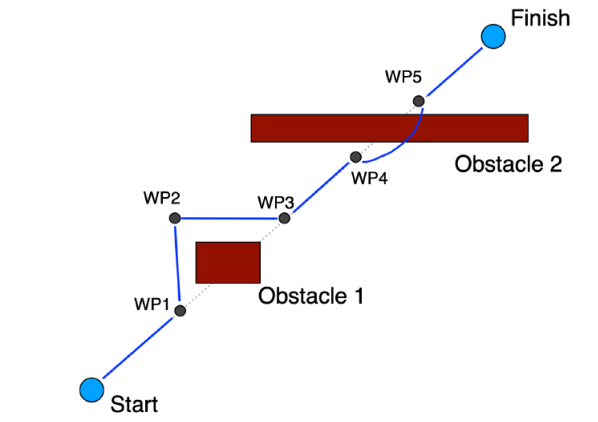


Figure 1: Desired behaviour of the vehicle

Furthermore, improving the already existing vehicle design may also be important since the correct allocation of sensors will be crucial for the obstacle avoidance part.

Last but not least, setting up path following and obstacle avoidance simulation environments is of paramount importance in order to analyze the algorithms performance before testing them on the robot.

2.2 State of the art & degree of novelty

Rovers and drones alone cannot be considered recent technologies, since there is already a lot of bibliographical references that explore them.

On the one hand, rovers are one of the commonly used robotic vehicles used to perform tasks in areas with hard accessibility [1]. Since these are efficient and able to move in rough terrain, they are also frequently utilized in the space sector to collect information about the terrain or even for taking crust samples. The most famous rovers are probably Curiosity and Opportunity, which were used by NASA to explore Mars, and the chinese Yutu and Yutu-2, which have been responsible for exploring the Moon in the past few years [2].

On the other hand, Unmanned Aerial Vehicles, better known as UAV, have also been widely used in the past years for a lot of different applications. This year, NASA sent its first flying vehicle to Mars, called Mars Helicopter Ingenuity [3].

However, there has not yet been any vehicle designed in order to be capable of moving both on the surface and in the air, although some small companies have already started attempting something similar recently. The Transition is one of the prototypes, created by Terrafugia, as well as The Urban Aeronautics X-Hawk [1].

3 Investigation Methods

3.1 Robotic Aerial Vehicle

As stated before, the vehicle in use is composed of two different modules, the drone and the rover. This robot, which can be observed in figure 2, has been previously assembled by former ISAE-SUPAERO students Anton Sambalov, Sakshi Chaudhary and Nandhini Raghunathan.

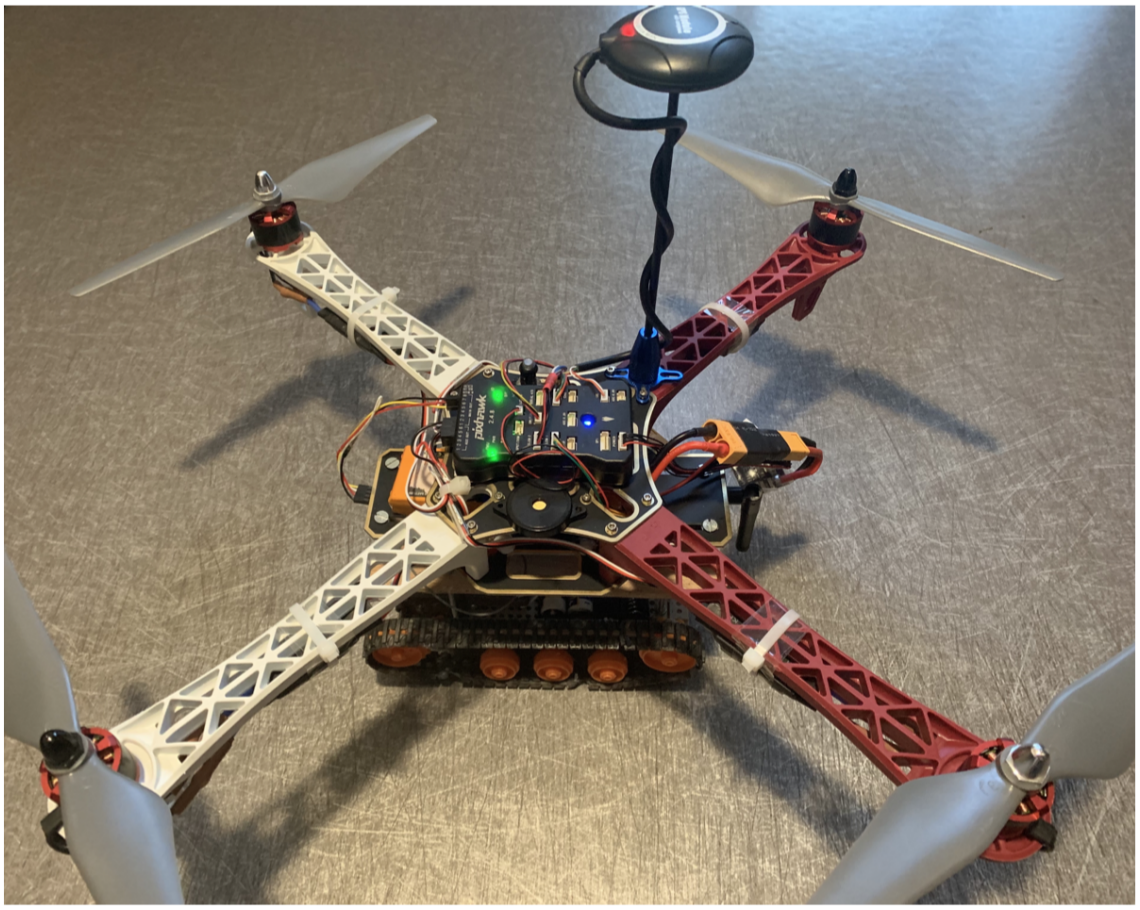


Figure 2: Fully Assembled Vehicle

3.1.1 Air Module

The air module is responsible for the flight part of the mission. It is able to lift its own weight as well as take-off with the ground module as a payload. Its components can be seen below:

- Frame: F450 Drone frame kit, which is an X-type frame with 450mm wheelbase.
- Power Distribution Board (PDB): Included in F450 Drone frame kit.
- Motors: Vehicle has four of the 2212 920kV brushless DC motors, meaning they have a stator width of 22mm and height of 12mm.
- Propellers: Propellers of type 9450 are used (9.4 inch diameter, 5.0 inch pitch).
- Electronic Speed Controllers (ESC): Four of the 30A ESCs are used in this project.
- Battery: Floureon 3S, 11.1V, 3300mAh, 40C LiPo battery is used to power up the vehicle.
- Flight Controller: The vehicle is equipped with a Pixhawk, which serves as a flight controller.

3.1.2 Ground Module

The Rover has a base similar to that of a tank, in view of the fact that a continuous track is used in order to achieve maximum control on rough terrain and also prevent the vehicle from getting stuck. The components of the ground module are listed below:

- Frame: Tamiya universal plate set is used as a frame.
- Tracks and Wheels: Ground module is based on Tamiya 70100 track and wheel set.
- Gearbox: Tamiya 70097 twin-motor gearbox consists of the gearboxes and two independent brushed DC motors, which are used to drive two shafts. Motors run on 3-6V.
- Power System: Battery box allows to install up to 4 batteries in series. At the moment two AA batteries are used to supply power to the motors.
- Motor Driver: The TB6612FNG motor driver is able to control speed and direction of the two motors. Having a supply range of 2.5V to 13.5V, it suits our motors.
- Computer: In order to let the motor driver know when and which way it should run the motors, as well as to communicate with the air module, there is a need for the on-board computer. Raspberry Pi is used for those purposes.

3.1.3 Raspberry Pi

The Raspberry Pi (RPi) is a credit card size single-board computer, which can be used for many different applications. A Raspberry Pi1 model A+ V1.1 was being used by Anton Sambalov [1] in order to control the rover movement and to implement the obstacle avoidance algorithms. This model was making use of the Raspbian operating system. However, installing Ubuntu Server on the RPi seemed to be by far the best option, due to reasons which will be explained later in this report. Because the current model was incompatible with this operating system, a new Raspberry Pi 3 Model B+ was acquired.

The desired Ubuntu image was then flashed onto a microSD card and then loaded into the Pi. Furthermore, the latter was connected to the PC using an SSH connection, which allows to log into a command prompt from a laptop and execute commands in the Pi. It should also be noted that this connection is only possible if the laptop and the RPi are connected to the same wifi. The result is displayed in figure 3.

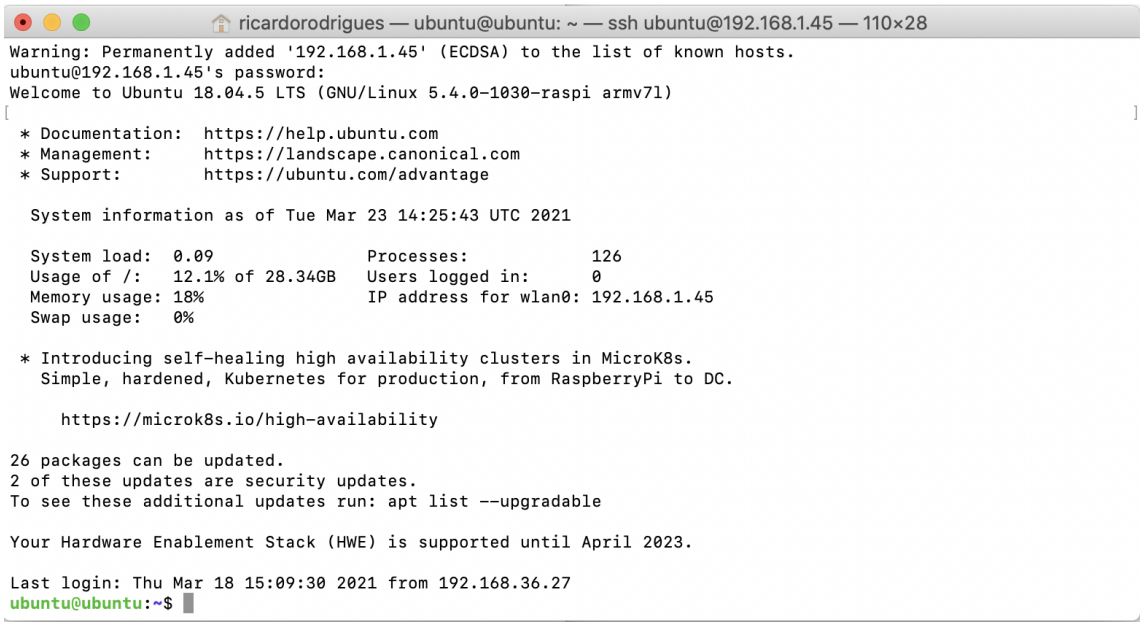


Figure 3: Raspberry Pi connected to computer using SSH

3.2 Software components

This section explores all the software components that were utilized in the experimental setup.

For this project, the Robot Operating System (ROS) [4] was used as a framework. The latter provides libraries and tools which allow to design and coordinate robot software. One should note that a ROS-based process is called a node. Different nodes are then able to communicate between each other by sending messages through ROS. The concept of node will be very important throughout the whole content of this report.

The next step was then to find a suitable simulation environment for testing different algorithms on this type of vehicle. This part is very important since it enables experimentation on a valid digital model of the system. Furthermore, the possibility of observing the performance of different algorithms in 3D is a major tool for understanding how the robot is going to behave in the real world.

The chosen simulation environment was the open-source 3D simulator Gazebo [5]. It provides a very robust physics engine which is able to detect collisions between the robot and the environment around, providing the user with a very accurate approximation of what would happen in reality. It also allows the usage of sensor models, which give the necessary sensor data to the robot model being simulated. Additionally, Gazebo is compatible with ROS. This means that, for example, the sensor data coming from the Gazebo sensor models can be published on a ROS topic. Topics are the way ROS uses in order for nodes to exchange messages between each other. Therefore, a publisher node can publish data on a topic and other subscriber nodes can then subscribe from that same topic in order to gather information.

Moreover, as stated before, a Pixhawk flight controller is used for this project. In Gazebo, one can simulate drones which are controlled by the PX4 flight control software. What happens is that the autopilot communicates with the rest of the system through a ROS node called MAVROS [6], which uses the MAVLink communication protocol. For example, an obstacle avoidance node collects information from the PX4 autopilot regarding the state of the vehicle, using the MAVLink protocol. Afterwards, it feeds the flight controller with the computed setpoints, which the vehicle will then follow. All of this happens by using the MAVROS node.

At this point, the whole configuration of PX4 and ROS has been explained. The relationship between each part is shown in figure 4.

As can be seen, PX4 Software In The Loop [7] communicates with the simulator, which is Gazebo, in order to receive sensor data from the simulated world and then send motor and actuator commands. It can also communicate with a Ground Control Station. In this project, QGroundControl will be used for that purpose. It should be noted again that the Offboard API represented in figure 4 is ROS.

One can now better explain why the new RPi model was acquired. A Virtual Machine running Ubuntu will be used for simulations due to the fact that ROS is only supported on Linux. This means that if Ubuntu is installed on the Pi, all the nodes being simulated can directly be implemented on the real vehicle. The only differences are that the sensor data will be coming from real sensors instead of Gazebo sensor models and the actuator & motor commands will be computed by the real flight controller, which will act on the robot so that it follows the desired behaviour.

3.3 Ground module

This part of the report concerns the ground module. In order to drive a robot from a point A to a point B, information about the goal point, the robot itself and the surroundings will need to be fed into a designed controller which objective is to follow a path.

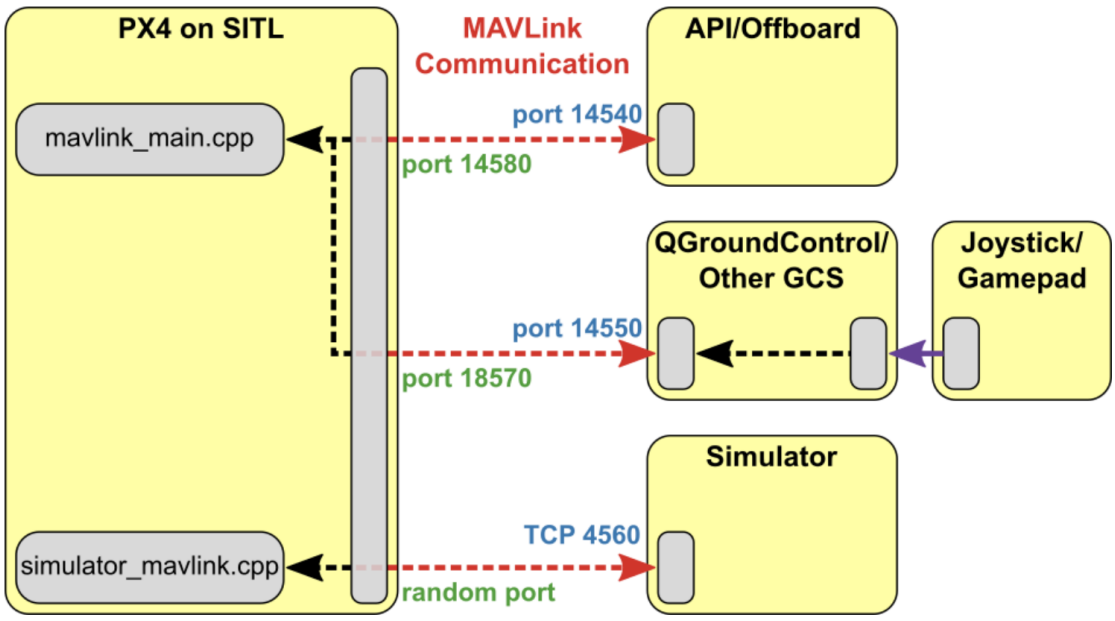


Figure 4: Complete configuration of all the software components [7]

3.3.1 Building a model for the rover

In order to design behaviors or controllers for a robot, a model of its dynamics is inevitably needed. Even though the ground part of the vehicle makes use of tracks and five wheels on each side [1], its kinematics are similar to the ones of a differential-drive mobile robot with a single fixed axle and two wheels separated by a specified track width. This is due to the fact that the gearbox and motor driver are actually only inputing torque to one wheel on each side, while the others rotate due to the Tamiya 70100 track.

This type of vehicle has two wheels, which can turn at different rates. For instance, if they are turning at the same rate, the robot is moving straight ahead. If one wheel is turning slower than the other, then the robot is going to be turning towards the direction in which the slower wheel is. The first thing that needs to be known is what the dimensions of the rover are. These dimensions are displayed in figure 5.

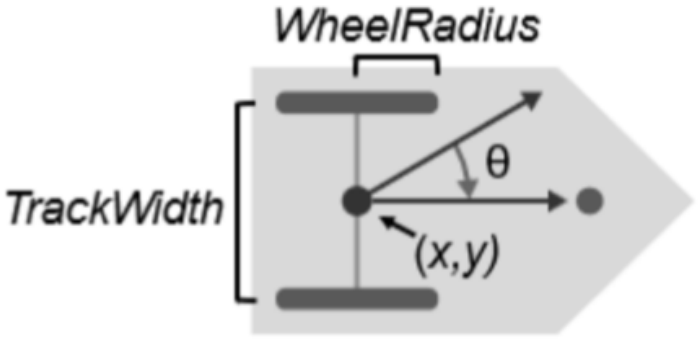


Figure 5: Dimensions of the robot [8]

The goal is to control how the robot is moving. The control signals that are at one's disposal

are given by v_r, which is the rate at which the right wheel is turning, and v_l, which is the rate at which the left wheel is turning. These two represent the two inputs to the system. Therefore, now that the inputs are known, the question is: what is the state of the vehicle? Well, typically, for a mobile robot, it is x and y, which define the robot's position, in meters, and its heading θ, defined in radians. These variables are also observable in figure 5. Furthermore, the robot model needs to connect the inputs v_r and v_l to the state, somehow. That is the function of the differential-drive robot model dynamics, which can be seen below.

$$\begin{cases} \dot{x} = \frac{R}{2}\left(\mathrm{v}_r + v_l\right)\cos\theta \\ \dot{y} = \frac{R}{2}\left(\mathrm{v}_r + v_l\right)\sin\theta \\ \dot{\theta} = \frac{R}{L}\left(\mathrm{v}_r - v_l\right) \end{cases} \tag{1}$$

These dynamics, where R represents the radius of the wheels and L the track width, provide everything needed in terms of mapping control inputs onto states. However, it is quite cumbersome and unnatural to think in terms of rates of various wheels. That is the reason why this model is not that commonly used when designing controllers. Instead, something called the unicycle model can be used. This one overcomes the issue of dealing with unnatural or unintuitive terms, like wheel velocities, caring only about the speed of the vehicle and its angular velocity. The unicycle dynamics look as follows:

$$\begin{cases} \dot{x} = v\,cos\theta \\ \dot{y} = v\,sin\theta \\ \dot{\theta} = \omega \end{cases} \tag{2}$$

This is the model for which a controller is going to be designed. Nevertheless, it should not be forgotten that this is not the differential-drive model. As a result, these new dynamics will have to be implemented on the latter.

After some mathematical transformations, it was possible to conclude that the linear velocity v and the angular velocity ω are given by:

$$\begin{cases} v = \frac{R}{2}\left(\mathrm{v}_r + v_l\right) \\ \omega = \frac{R}{L}\left(\mathrm{v}_r - v_l\right) \end{cases} \tag{3}$$

The linear equations above can be explicitly solved for v_r and v_l. After doing that, the result was:

$$\begin{cases} v_r = \frac{2v+\omega L}{2R} \\ v_l = \frac{2v-\omega L}{2R} \end{cases} \tag{4}$$

It should be noted that v and ω are designed parameters. Moreover, L and R are known measured parameters for the robot. With all these parameters, it is possible to map the designed

inputs v and ω onto the actual inputs that are indeed running on the robot, which are v_r and v_l.

A Gazebo model had now to be designed for the ground module. This one is essentially a collection of links, joints, collision objects, visuals, and plugins. The final result is displayed in figure 6.

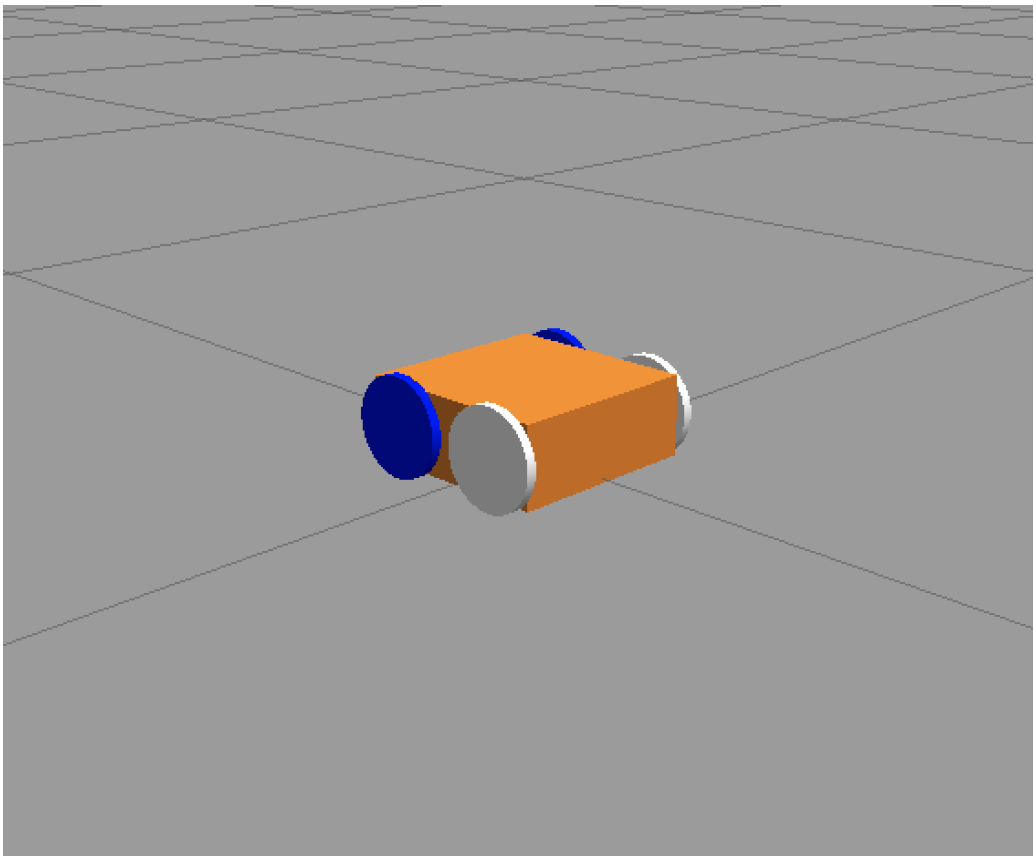


Figure 6: Rover Gazebo model

The orange box represents the frame, the gearbox, the motor driver and all the other components of the ground module, expect for the tracks and wheels. Its length and width are equal to 0.2 m, while the height is given by 0.08 m. The wheels have a radius of 0.05 m and a length of 0.01 m. The complete model weighs 1.5 kg.

In order to simulate the dynamics of a differential-drive mobile robot as the one that has been described before, *ros_control packages* were added to the blue wheels. These are more specifically called as *joint_velocity_controllers* and they are responsible for receiving velocity commands, sending them to a PID controller and only then output the required torques to the wheels. These PID controllers were chosen to have the proportional gain P equal to 1 and D = I = 0. The objective of these controllers is therefore to control the output which is sent to the actuators. The white wheels will not receive any command and they were built just to simulate the symmetry properties of our vehicle. It should also be noted that the masses and inertias of all the links (orange box and wheels) that have been included in this model were adjusted in order to better approximate the characteristics of the real robot.

3.3.2 Path following and obstacle avoidance algorithms

The world is fundamentally dynamic and unknown to the robot. Therefore, it does not make sense to overplan and think very hardly about how to act optimally given these assumptions about what the world looks like. That may make sense when designing controllers for an industrial robot at a manufacturing plant where the robot is going to repeat the same motion over and over again, but not in this case.

For that reason, it is also a bad idea to try to produce, in advance, a very complicated monolithic controller for doing everything. Instead, one should be able to divide the control task into chunks and then design controllers for those individual chunks. For instance, in case the robot wants to get to a goal point, it may have some kind of controller that is taking it there. On the other hand, when something shows up in the environment, it should switch to another controller that allows the machine to avoid that specific obstacle in the environment. Therefore, the key idea is to simply develop a library of useful controllers which do different things.

The next step was then to start idealizing a go-to-goal controller when considering a constant linear velocity v. Let's say the rover is heading in direction θ and, for some reason, it wants to go in direction $\theta_{desired}$. In order to drive the robot in this direction and since it is known that $\dot{\theta} = \omega$, the solution is to figure out what ω should be equal to. Well, in this situation, there is a reference $r = \theta_{desired}$, a model, a control input and a tracking error given by $e = \theta_{desired} - \theta$, which means that everything is ready for a controller to be designed. In this case, it should be a PID controller, since it combines the advantage of proportional, derivative and integral control action. Consequently, $\omega = K_p e + K_i \int e d\tau + K_d \dot{e}$.

K_p, K_i and K_d are the proportional, integral and derivative gains, which need to be tuned like in any other PID controller. It is also easily predictable that making K_p large makes the system respond quicker but may induce oscillations. Additionally, the derivative gain makes the system very responsive but it can make it become a little bit oversensitive to noise. Finally, the integral part of the controller is integrating all the tiny tracking errors that may exist and, after a while, it becomes large enough to push the system up to no tracking errors at all. However, a big K_i may also induce oscillations.

From the action of different types of sensors, the pose of the vehicle, which is given by x, y and the heading θ, will be estimated. The question now is related to the desired heading of the robot. In case it wants to go to a certain location which coordinates are given by (x_g, y_g), $\theta_{desired}$ will be equal to:

$$\theta_{desired} = \arctan(\frac{y_g - y}{x_g - x}) \tag{5}$$

A model describing the kinematics of the ground module, together with the principles of a go-to-goal process, have now been described. This algorithm computes the necessary angular velocity to move the robot from its position to a final destination while maintaining the vehicle's linear velocity constant. The question that arises now is: what if the robot needs to do something slightly more elaborate than just get to a goal point? For instance, the robot must not hit obstacles on the way to its final destination. The idea is to create an obstacle avoidance controller that takes over whenever the sensors find an obstruction. However, it should be noted that there are multiple ways of avoiding obstacles (the robot can take several different headings), such as going in the opposite direction, what may seem a little bit too overcautious. This would be called a pure avoidance algorithm, which means the robot would

just prevent collisions and nothing else.

Another idea is to move away from the obstacle while somewhat getting towards the goal. Well, the punchline here is that choosing a new heading after finding an obstacle is not a pure strategy since knowledge about the goal location is required. Therefore, what has to exist is the coexistence between an obstacle avoidance and a go-to-goal algorithm, in the sense that the ground module has to take the direction to the goal into account when figuring out in which direction it should be going after finding an obstacle. There are fundamentally two different possible mechanisms. The first one is a hard switch, which means that the vehicle is either using the go-to-goal or the obstacle avoidance approach and never both at the same time. The second one is a blended mechanism which means the vehicle is somehow combining the angle to the goal and the angle to the obstacle in order to produce a new desired angle. For example, when finding an obstacle, the robot would move perpendicularly but in such a way that it would be getting closer to the goal. Performance wise, blending or smoothing the two behaviors makes a lot of sense.

After some research, a specific type of obstacle avoidance algorithm attracted some attention. This one is an obstacle following algorithm [9] and it makes the robot follow the boundary of an obstacle at a certain safety distance by means of a laser sensor. We will now consider an example of how this algorithm behaves in figure 7, where p is the position of the robot, p_g is the goal location and Δ is a safety distance between the vehicle and the obstacle.

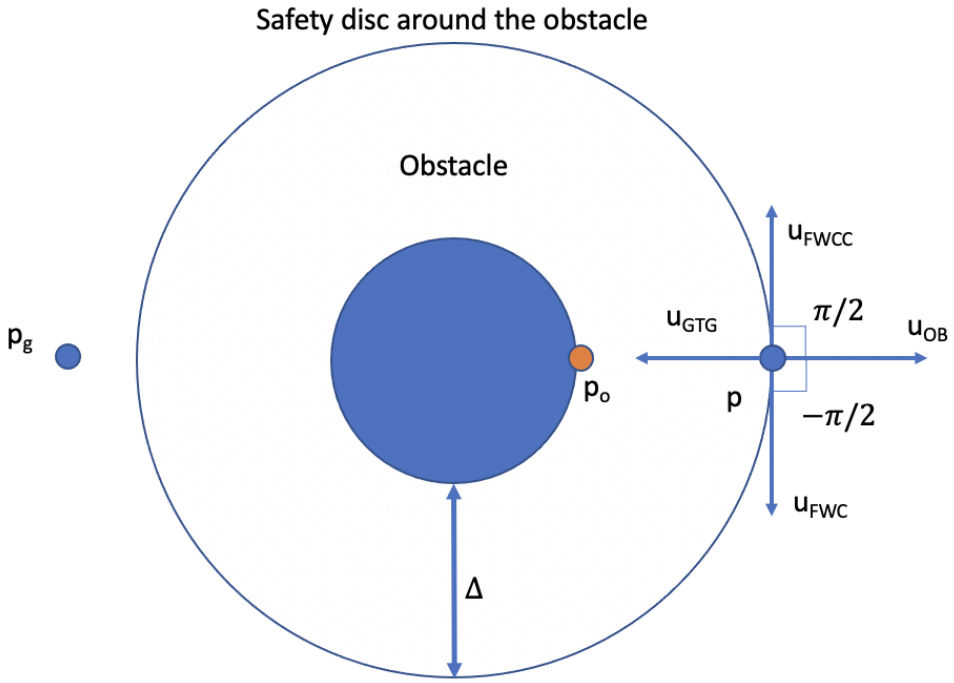


Figure 7: Behaviour of the obstacle following algorithm

The first question is: how does the algorithm know if it should be using the go-to-goal, the pure obstacle avoidance or the obstacle following mechanism? For that, we introduce a new function g, which is given by:

$$g(p) = \left(||p - p_0||^2 - \Delta^2\right) \tag{6}$$

If g is bigger than 0, the robot uses the go-to-goal approach. If g is 0, which is the case of

figure 7, it follows the obstacle. Finally, if g is smaller than 0, it uses a pure obstacle avoidance method, which results in the direction $u_{\vec{OB}}$. It should also be noted the p_0 represents the obstacle closest point which is being detected by the laser sensor.

It is now important to explore the case when g is equal to 0. First, the algorithm computes $u_{F\vec{W}CC}$ and $u_{F\vec{W}C}$ by rotating $u_{\vec{OB}}$ 90 degrees clockwise and counterclockwise. The question that arises is: how should the robot decide in which direction to go? Well, in this case, the algorithm lets the go-to-goal mechanism decide. It takes the inner product between $u_{\vec{GTG}}$ & $u_{F\vec{W}CC}$ and then between $u_{\vec{GTG}}$ & $u_{F\vec{W}C}$ and checks which of the products is greater than 0. In the case of figure 7, the direction the robot follows is irrelevant, since both paths will take the same amount of time.

The last question then is: when should the robot stop following the obstacle and just go straight to the goal? Well, that should happen when enough progress has been made and there is a clear empty path between the robot and the desired location. The vehicle then switches to the go-to-goal algorithm again and moves towards the goal point. That happens when $||p - p_g|| < ||p(\tau) - p_g||$ and the inner product between $u_{\vec{GTG}}$ and $u_{\vec{OB}}$ is greater than 0, where τ represents the time when the last switch between the go-to-goal and the obstacle following mechanisms occurred.

We will now take the obstacle displayed at figure 8, which was created in Gazebo, as an example.

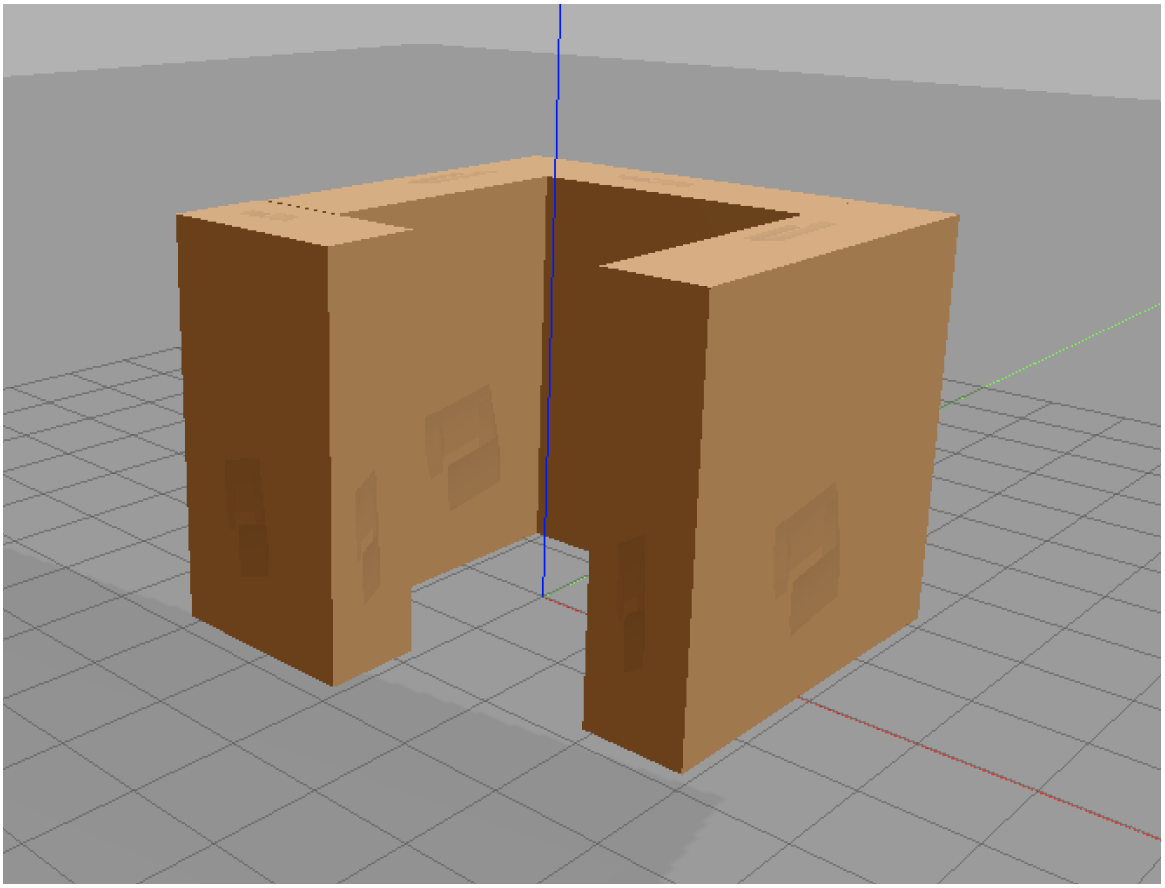


Figure 8: Gazebo wall model

In the Gazebo simulation environment, the red axis is the x axis while the green one represents the y axis. Finally, the blue line takes the place of the z axis. It should also be noted that the total ground is a mesh with a spacing of 1 meter.

If a robot is placed at the origin and wants to move to a goal point given by $(x, y, z) = (3, 10, 0)$, in meters, it will eventually detect the wall and it will have to follow it until there are no more obstacles between the vehicle and the desired location. The robot can do it clockwise, which in this case will be the long way around, or counter-clockwise, which will represent the shortest path. In this case, because the x coordinate of the goal location is positive, the robot would move counter-clockwise. In reality, this procedure will not always result in the most optimal path, although it is a very good way of choosing a direction without having a full understanding of the environment around.

As referred before, the world is fundamentally dynamic and unknown to the robot. Therefore, it is almost impossible to always plan a free of obstacles path before the robot even starts moving. This controller allows to avoid obstacles on the spot thus deserving attention. Instead of using a laser sensor, another possibility is also to use ultrasonic sensors in order to detect obstructions.

3.4 Air module

Now that we had explored the ground module, path-following and obstacle avoidance methods were also needed for the drone. In this case, there was no need to worry about creating controllers or path following algorithms for the air module. This is due to the fact that the PX4 autopilot already contains embedded PID controllers which are responsible for tracking the desired setpoints. However, these may need some tuning, something which will be explored later in this report.

3.4.1 Multicopter Control Architecture

There are two types of controllers, namely lower and higher-level. The higher-level controllers always pass its results to the lower-level controllers, as in layers, where the highest-level controller is the one that controls the position of the drone. Then, there is the attitude controller. Finally, the rate controller is the lowest-level one. PID tuning needs to be done in a growing level order thus starting with the rate controller, since it affects all the others. This Multicopter Control Architecture [10] is displayed in figure 9.

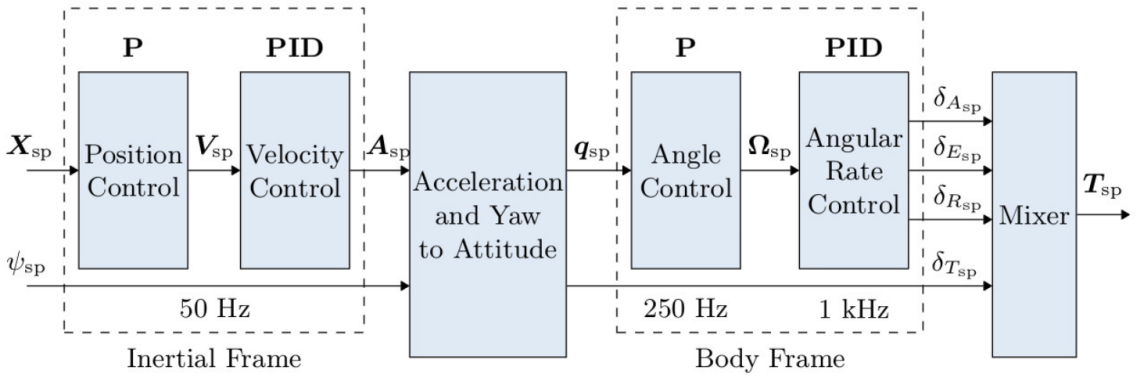


Figure 9: Multicopter Control Architecture

3.4.2 Offboard mode and setpoints creation

The first step was then to be able to feed setpoints to the flight controller, which the vehicle would autonomously follow. For that, a node consisting on a simple trajectory generation was created. This node implements a circular trajectory with a radius of 1 m, at an altitude of 2 m and with an angular rate of 0.5 rad/s.

In this node, a publisher is instantiated in order to publish the desired position setpoints, as well as clients which request arming of the vehicle and mode change to Offboard mode. The biggest difference between clients and publishers is that clients send some data but also expect a response message. These are usually used for quick actions. It should also be noted that the Offboard mode serves for the case when setpoints are being provided by MAVROS running on a companion computer (RPi), which is our case.

PX4 has a timeout of 500ms between two Offboard commands. In case this timeout is exceeded, the autopilot will enter Failsafe mode, which brings the vehicle to an altitude of 10 m and lands it afterwards in the current x and y position. This is why the publishing rate needs to be faster than 2 Hz, which already accounts for possible latencies. Before publishing anything, the node waits for the connection to be established between MAVROS and the autopilot. Then, a stream of setpoint commands must be received by the vehicle prior to engaging the Offboard mode. In the rest of the code, there is a switch to Offboard mode, after which the vehicle is armed and ready to fly. In the same loop, the requested setpoints are then sent at an appropriate rate.

Sending setpoints to the vehicle can also be done with QGroundControl. The difference is that the vehicle will be in Mission mode instead of Offboard mode. The QGC Plan View is used to plan autonomous missions and upload them to the vehicle. Afterwards, one can then fly the mission.

3.4.3 Obstacle avoidance algorithm

After knowing how to feed the flight controller with position setpoints, it was time to find an obstacle avoidance algorithm for the drone. In the PX4 documentation, a very interesting one was found [11]. The latter builds a map of the environment and computes optimal paths towards the goal. For this to be possible, accurate global position and heading are required.

Path finding algorithms for drones always aim to find a trajectory in 3D space while minimizing the distance, time and energy that it takes to get to a goal point. This algorithm [12] contains a global planner which objective is to find a path to the goal, as well as a local planner which is responsible for converting that same path into velocity commands. The latter are then transmitted to the autopilot and translated into motor torques.

In order to track obstacles, it uses a grid-based solution named OctoMap [13], which is an efficient way to represent a 3D environment. This framework is based on the octree structure, which is a hierarchical structure containing multiple nodes, that can be seen as cubic volumes in space. The purpose of this method is to discretize the environment around the vehicle. This type of structure can be observable in figure 10.

Robot applications often use a probabilistic representation of the environment in view of the fact that input sensors are not 100% reliable and have some uncertainty on its measures. All the octomap leaf nodes are associated to a probabilistic number, what allows the octomap to have a probabilistic representation of the environment. The probabilistic number which is associated to each node will then be used to determine if it is empty or occupied.

In simulation, the OctoMap will be implemented as a ROS node named *octomap_server*. In figure 11, one can visualize a Gazebo model and its OctoMap representation. The latter

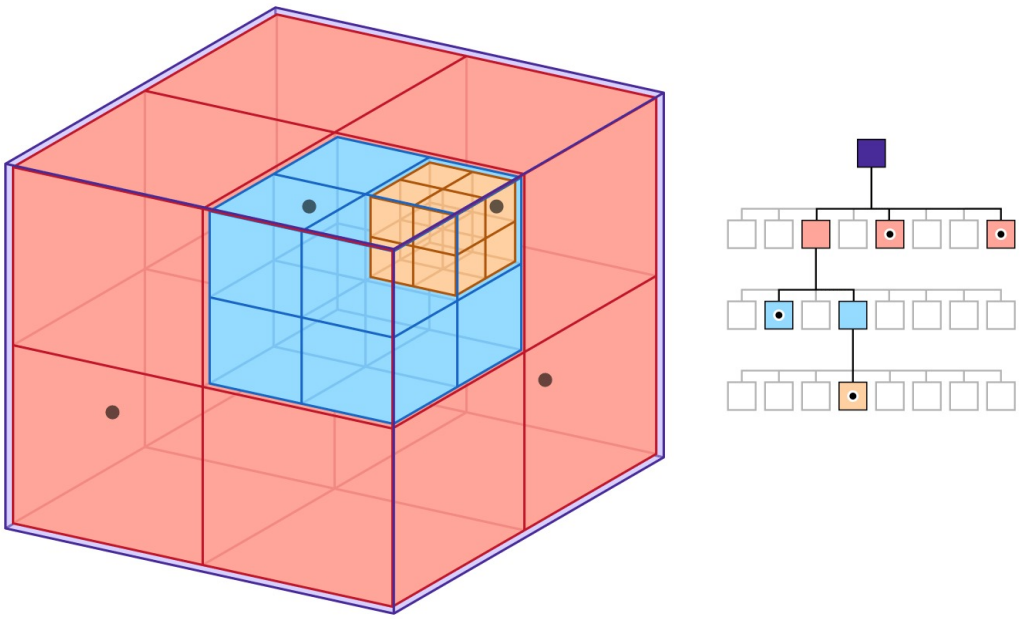
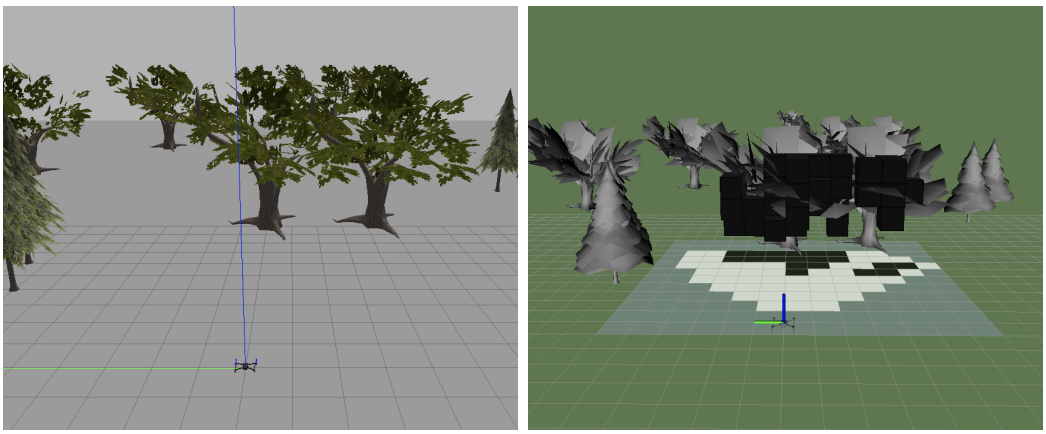


Figure 10: Octree structure [14]

was obtained using Rviz, which is a ROS graphical interface that allows the visualization of information like the one published by cameras.



(a) Gazebo tree (b) OctoMap tree

Figure 11: OctoMap representation of a Gazebo tree model

The way the algorithm works is the following: for every obstacle, the Octomap traces a ray between the camera and a certain point. Afterwards, it increases the probability of the node which cointains that point being occupied, while reducing the same probability for all the nodes in between the camera and the point that is being investigated.

Moreover, the *global_planner_node* handles all the communication for the global_planner, which means that it receives information about the current position of the vehicle, where obstacles are located and also the desired goal location. Then, it plans a free of obstructions path and publishes it in the form of waypoints to the *path_handler_node*. The latter makes the bridge between the *global_planner_node* and MAVROS, by getting the new waypoints and streaming them to MAVROS. The communication between all the nodes of the system can finally be observable in figure 12.

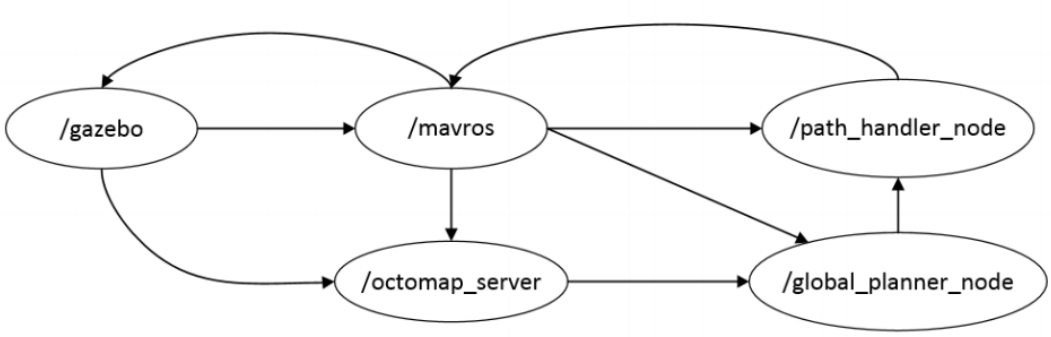


Figure 12: Complete communication of the all the system nodes

3.5 Gazebo hybrid model for simulation

Now that all the desired path following and obstacle avoidance algorithms had been found, only one thing was missing, which was creating an hybrid model in Gazebo for simulation. This model had to be as accurate as possible when compared to the real vehicle, so that realistic results could be obtained.

PX4 Software In The Loop already had some supported vehicles in Gazebo, such as the 3D Iris Quadcopter model, which can be seen in figure 13. The latter approximately matches the characteristics of the air module of our vehicle.

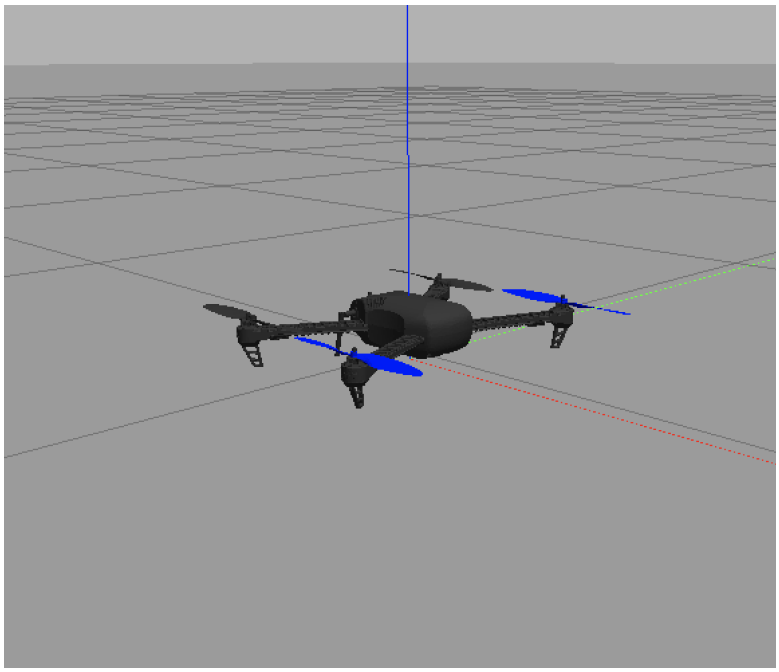


Figure 13: 3D Iris Quadcopter model

However, because there was not any model matching the hybrid features of our robot, a vehicle comprised of the Iris and the rover model was created as shown in figure 6. Moreover, two different sensors were added. The first one is a laser sensor, which is required for the obstacle avoidance algorithm concerning the rover. The second one is a stereo camera, which is demanded by the air module global planner. The final model for the vehicle is displayed in 14.

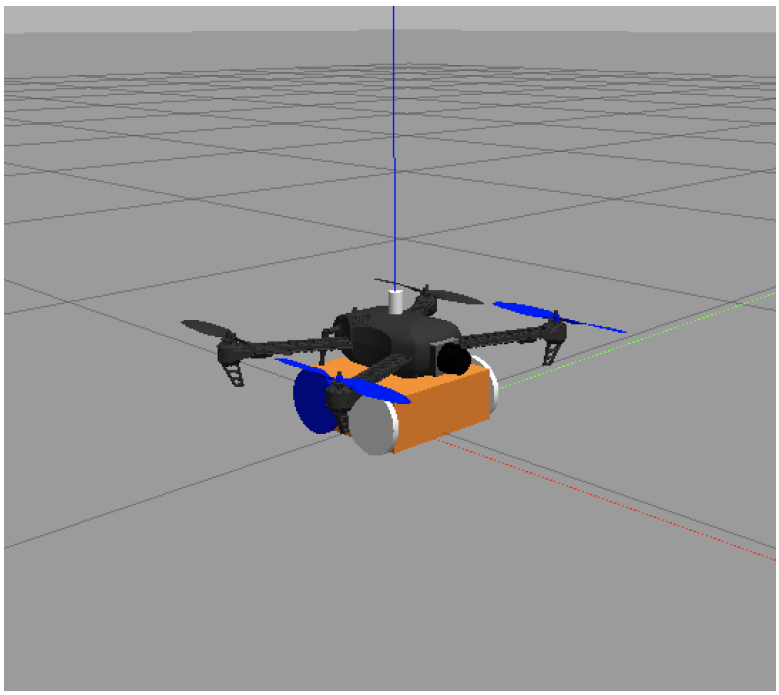


Figure 14: Hybrid Gazebo model

The stereo camera was placed in front of the Iris, whereas the laser sensor, which is represented by a white cylindrical shape, was set on top of the whole vehicle. This avoids the possibility of the laser detecting the vehicle itself instead of obstacles in the surroundings. In figure 15, one can observe the detection range of this sensor in Gazebo, which has a radius of 1 meter and can detect obstructions in all directions at the same time. Gaussian noise was also added in order to make the simulation more realistic. It should also be noted that the complete hybrid model has a mass of 2.2 kg.

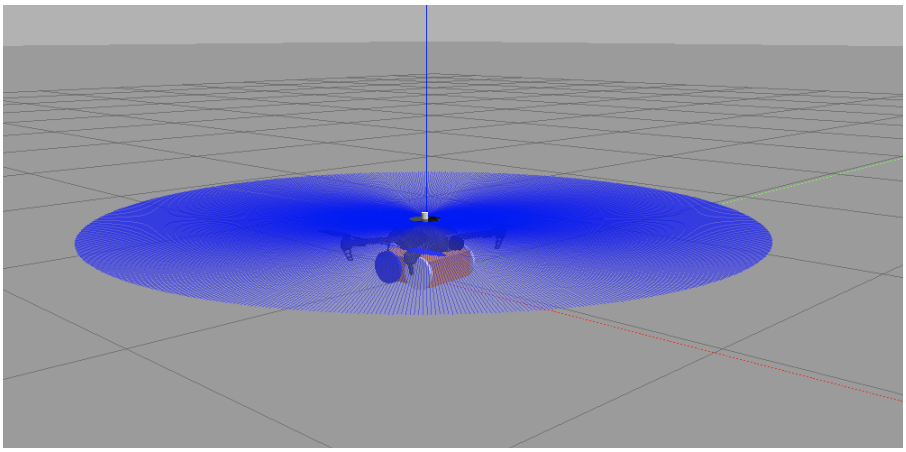


Figure 15: Range of the laser sensor in Gazebo

3.6 Connection between RPi and Pixhawk

Connecting a Raspberry Pi 3 Model B+ to a Pixhawk-family board works in the following way: They are interfaced using a serial port to TELEM 2, which is the port intended for the

communication between an onboard computer and the flight controller. The message format on this link is MAVLink.

First, MAVLink was enabled on TELEM 2 for the Pixhawk. In order to set up the default companion computer message stream on this port, the following parameters had to be set on QGroundControl and then uploaded onto the vehicle:

- MAV_1_CONFIG = TELEM 2
- MAV_1_MODE = Onboard
- SER_TEL2_BAUD = 921600 (recommended baud rate for this type of application)

Afterwards, the companion computer also needed to be set up, since it needs to run some kind of software talking to the serial port and allowing the MAVLink communication. As stated before, the Ubuntu operating system was installed on the RPi, together with ROS and the MAVROS node.

Following next, the TELEM2 port had to be physically connected to the pins of the Raspberry Pi. The TX and RX pins were connected to the TELEM2 port of the Pixhawk, as well as the GND and 5V pins, which are used to provide the RPi with power. However, the Raspberry Pi kept turning off as the 5V bus on the TELEM2 port was not providing enough current to the onboard computer. Consequently, other connections had to be made, so that power coming from the 5V bus on the Pixhawk servo rail was also being provided. The complete configuration is displayed in figure 16.

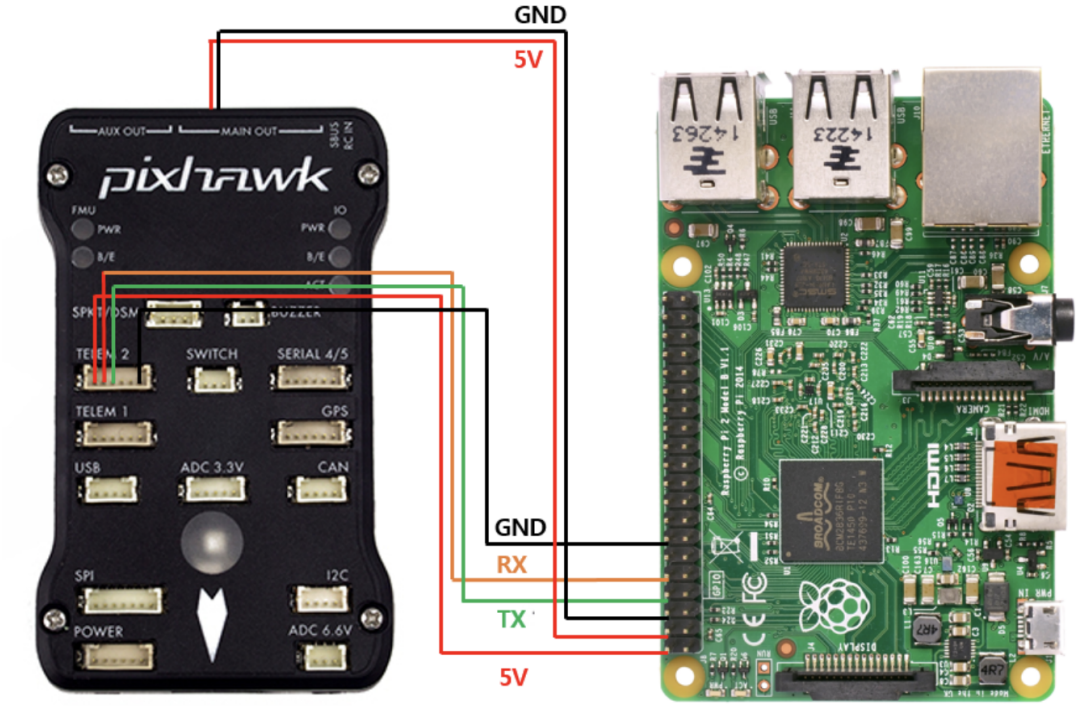


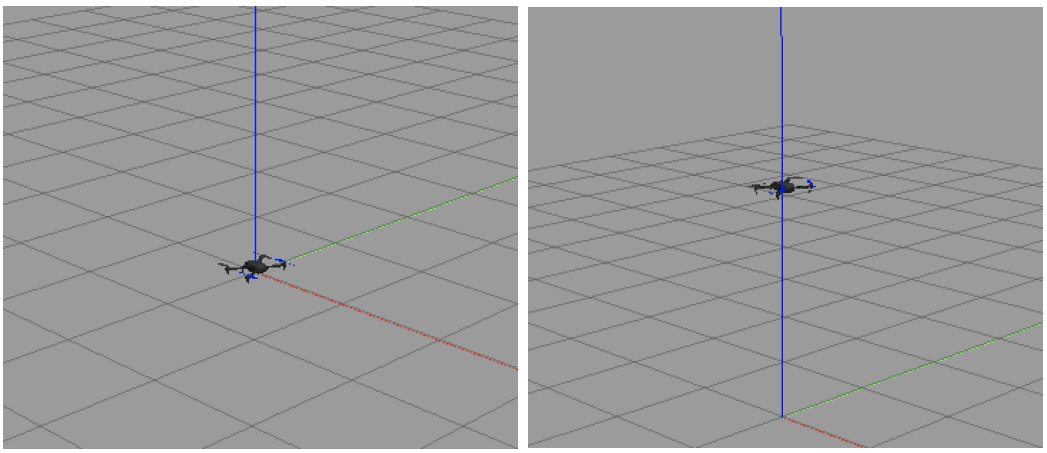
Figure 16: Complete configuration between RPi and Pixhawk [15]

4 Results and Analysis

It should be noted that, for obtaining results, ROS launch files were used. These launch files usually bring up a set of nodes while providing some aggregate functionality between them. This is how Gazebo, MAVROS, the PX4 SITL (Software In The Loop) and all the required nodes were launched in the same simulation.

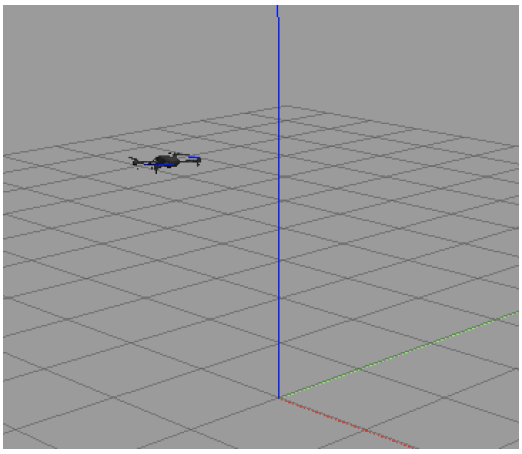
4.1 Circular trajectory node

The circular trajectory node was tested in this section, using the Iris model which was presented in figure 13. In this case, the objective was to investigate if the vehicle responded correctly to the given commands. While the simulation was running, some pictures of the vehicle state were taken. In figure 17, one can observe the vehicle at rest, taking off and then following a circular trajectory, as intended.



(a) Vehicle at rest

(b) Take off



(c) Circular trajectory

Figure 17: Different moments of the simulation showing the performance of the circular trajectory node

In addition, figure 18 displays the evolution of the state of the vehicle, more specifically of x, y and z. It is clearly observable that the robot follows a circular trajectory of a 1 m radius at an altitude of 2 meters. The tracking is very good, for the simple reason that the PX4 embedded PID controllers are already tuned for this type of quadcopter. In the next section, we will see that the hybrid model which was created will need some tuning.

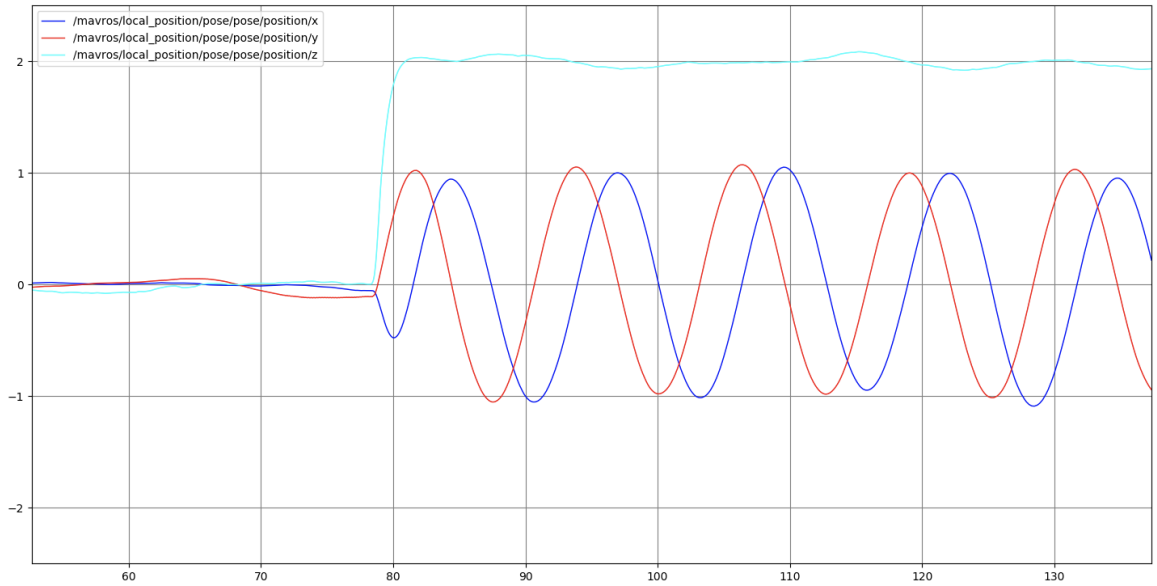


Figure 18: Time evolution of x, y and z for the circular trajectory node

4.2 PID tuning

The hybrid model displayed in figure 14 will be used in this section. Because the rover was added to the Iris model, its center of mass changed, together with the moment of inertia of the whole vehicle. Therefore, and because the objective was to test the algorithms on the hybrid model, the PX4 embedded PID controllers had to be tuned.

Here, a new node was used. The latter arms the vehicle, puts it into Offboard mode, and consecutively inputs the following waypoints into the system: (0,0,2), (2,2,2), (2,2,4) and (0,0,4). The time evolution of the vehicle state is displayed in figure 19, where the default PID gains for a quadcopter were applied.

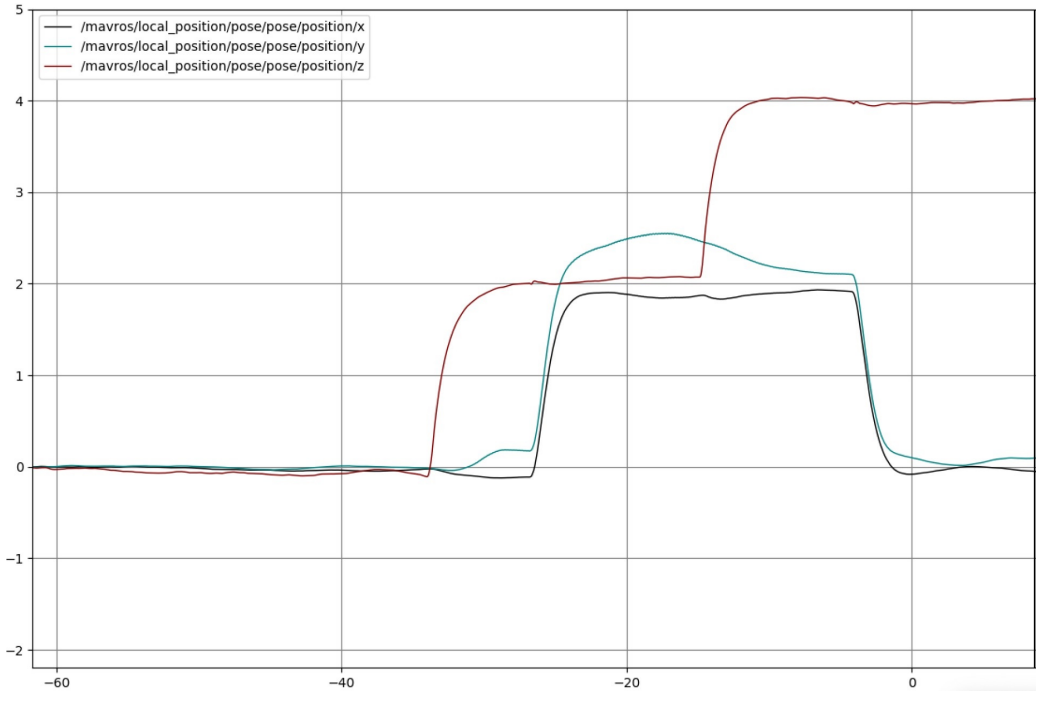


Figure 19: Time evolution of x, y and z before PID tuning

The tracking for the z axis showed up to be quite good. However, for the x and the y axis, the tracking errors were not very satisfactory. On the one hand, the overshoot and steady-state error for the y axis response were quite large. On the other hand, the x axis response showed a slight steady-state error, in view of the fact that it never achieved the value of x = 2 meters, as required.

PX4 PID gains can be tuned by changing its equivalent parameters on QGroundControl. Furthermore, QGroundControl has a PID Tuning setup [16] which provides real-time plots of the vehicle setpoints and response curves for the yaw, pitch and roll rates. The goal is then to set the PID values such that the response curve matches the setpoint curve as accurately as possible, while attempting to achieve a fast response with small overshoots. This PID Tuning setup can be seen in figure 20.

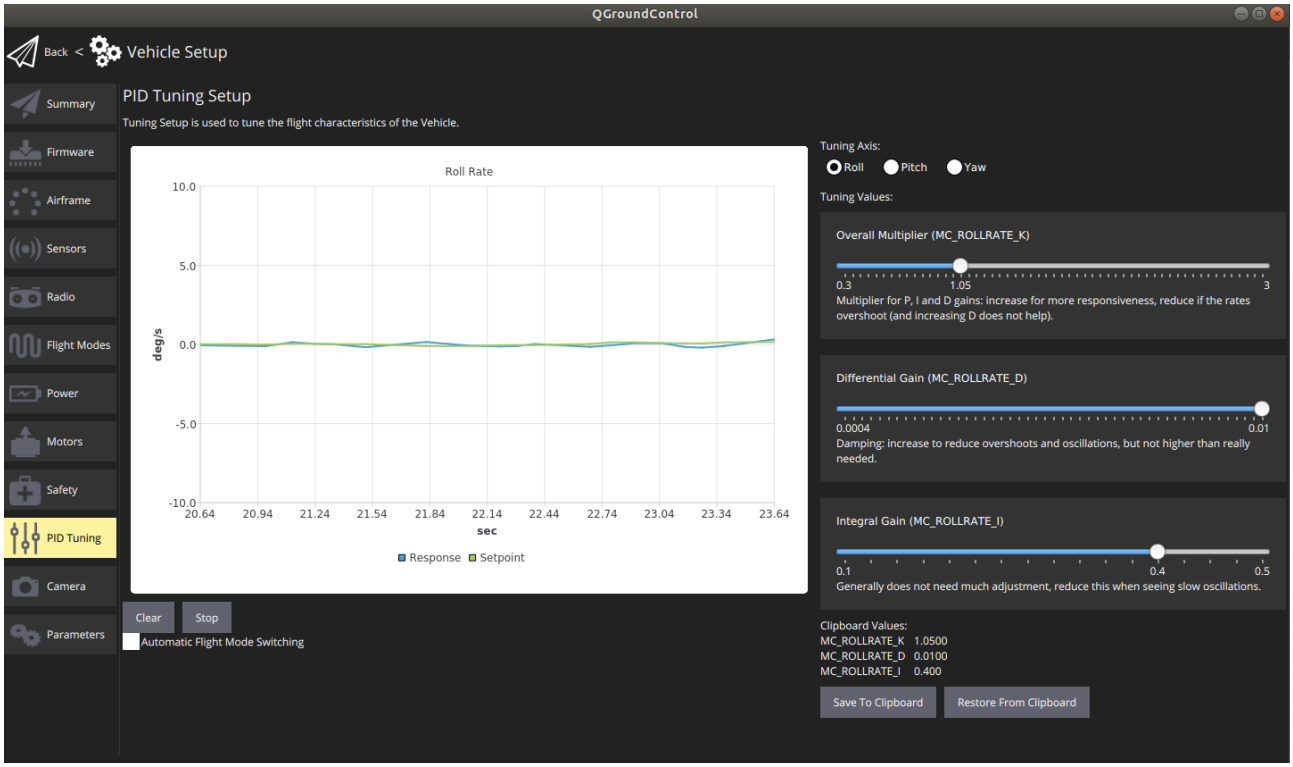


Figure 20: QGroundControl PID Tuning setup

More attention was given to the derivative and integral gains because increasing both leads to damping overshoots and reduced steady-state error. After adjusting all the gains for the different rates, the best compromise was obtained with the following parameters:

- MC_ROLL_P = 7
- MC_ROLLRATE_P = 0.3
- MC_ROLLRATE_I = 0.4
- MC_ROLLRATE_D = 0.01
- MC_PITCH_P = 7
- MC_PITCHRATE_P = 0.3

- MC_PITCHRATE_I = 0.5
- MC_PITCHRATE_D = 0.01
- MC_YAW_P = 2.5
- MC_YAWRATE_P = 0.5
- MC_YAWRATE_I = 0.12
- MC_YAWRATE_D = 0

These were then uploaded on the simulation environment and the same node was ran on the hybrid vehicle model. The results are displayed in figure 21.

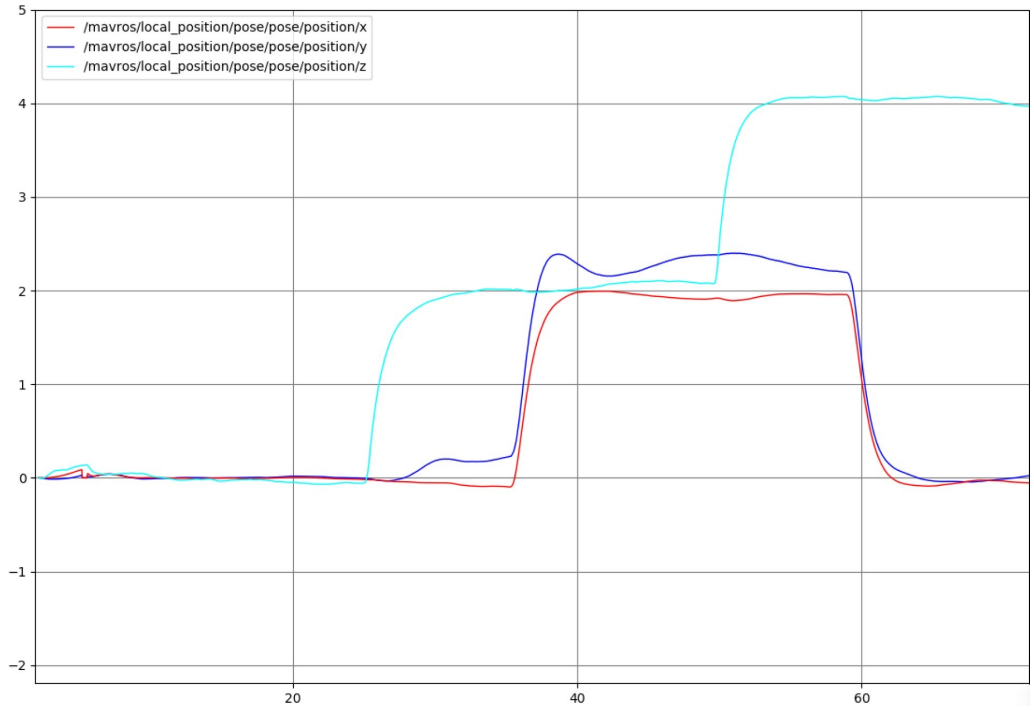


Figure 21: Time evolution of x, y and z after PID tuning

First, one should note that the tracking regarding the z axis position remained very good. Concerning the x axis, the steady-state error is now also very small, something which was not happening before when the desired setpoint was equal to x = 2 m. Finally, for the y axis, the behaviour improved, even though there was still some kind of overshoot and poor tracking. It should be noted that it was attempted to increase the integral gains even more in order to reduce the error. The problem is that it was creating some undesired oscillations. An effort was also made to enlarge the derivative gains and reduce the proportional ones to try to reduce the oscillations, but it did not work. Consequently, this was the best compromise which was found between all the gains, and these values are thus going to be utilized in the next simulations.

4.3 Ground module obstacle avoidance

In order to simulate both the go-to-goal and the obstacle following mechanisms [17], the wall obstacle which was shown in figure 8 was used. Some parameters had to be tuned and introduced

to the system, namely the distance between the center of mass of the two wheels, which is equal to 0.2 m, and its radius of the wheels, which is given by 0.05 m. In addition, this algorithm also needs an obstacle tolerance parameter, which was set to 0.5 meters. This distance is measured from the laser sensor to the obstacles, and that is the reason why this value was chosen to be bigger than the length of the robot. Moreover, the velocity was kept constant at 0.1 m/s. This algorithm also needs a distance threshold for the desired final location, in order to make the robot stop when reaching the goal, which was set as 0.1 meters.

Logically, the robot is placed at the origin in the beginning of the simulation. The desired goal point was chosen to be equal to $(x, y, z) = (3, 10, 0)$. In figure 25, one can observe different moments of the simulation. First, the vehicle uses the go-to-goal algorithm until it finds the wall. Because the x coordinate of the goal location is positive, the robot then decides to follow the wall in the counter-clockwise direction. As stated before, this is due to the blended mechanism of this algorithm, which chooses the direction to follow based on the go-to-goal. It can also be observable that, when the vehicle has made enough progress and there is an empty path between the robot and the desired location, it switches to the go-to-goal mechanism again and reaches the goal.

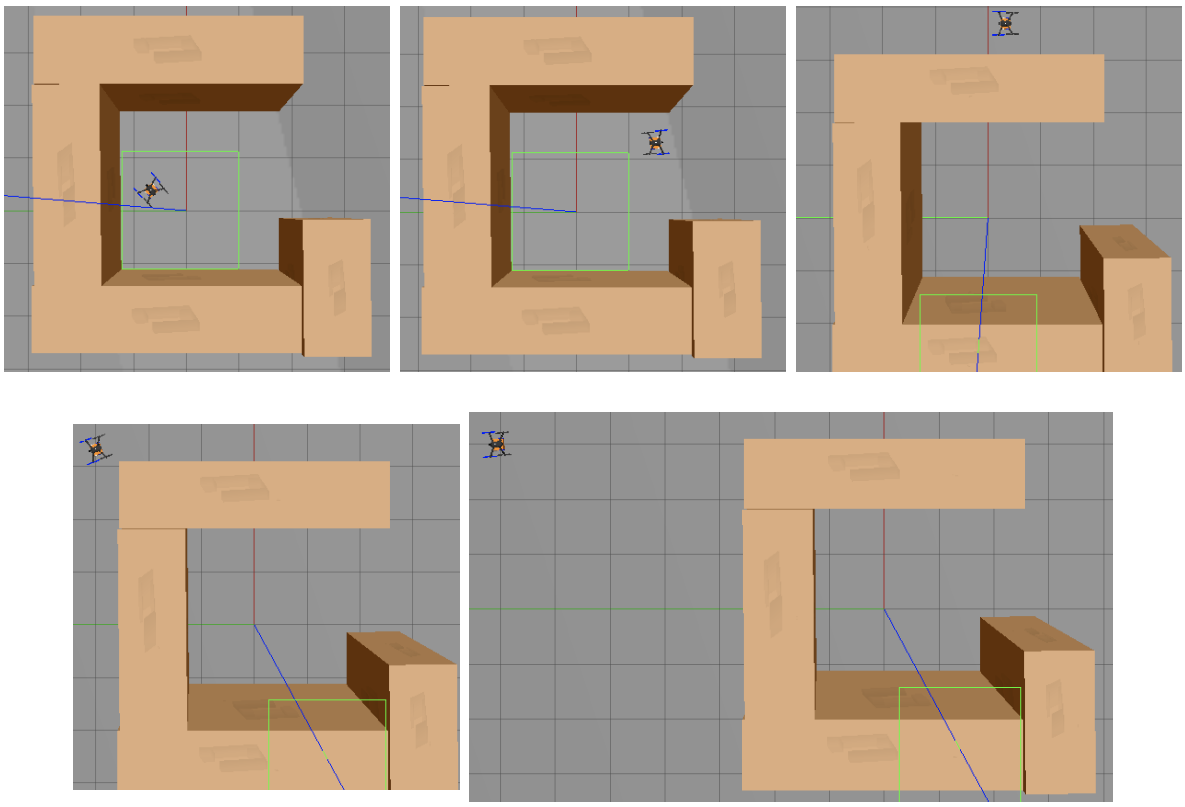


Figure 22: Different moments of the simulation showing the performance of the ground module obstacle avoidance algorithm

In order to better illustrate the path of the vehicle and because we were using the hybrid model to simulate this algorithm the MAVROS position topics were used for plotting the state of the robot with respect to time, as can be seen in figure 23.

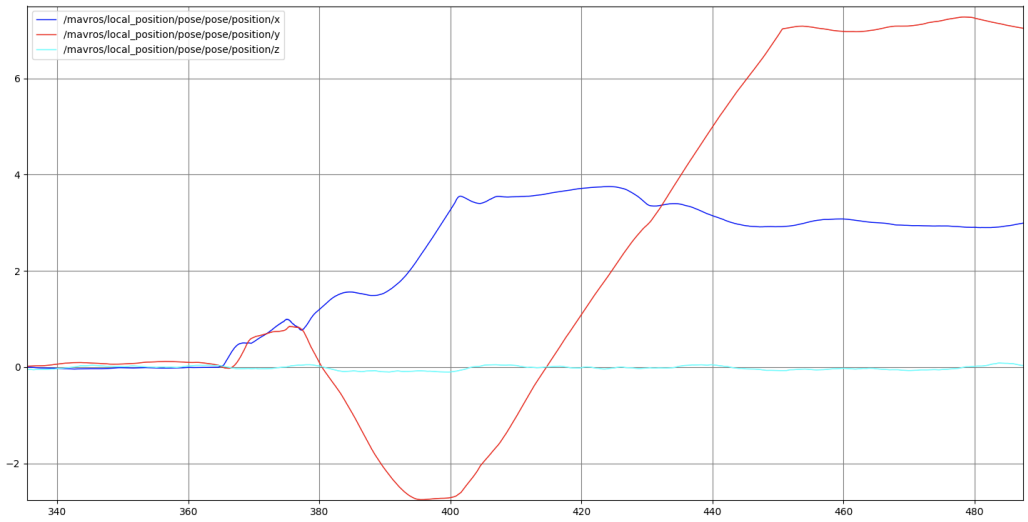


Figure 23: Time evolution of x, y and z for the rover obstacle avoidance algorithm

In the end, this algorithm calculates steering directions so that the robot does not collide with any obstruction on the way. First, it takes information from the laser scan data. Afterwards, based on some parameters and thresholds, it computes the directions in which the vehicle can freely move.

One should also note that the code which was simulated in this section can be directly implemented on the real vehicle. The only difference will be that the Raspberry Pi will be receive data from a real laser sensor, make the required computations and then send the commands to the motors. The laser sensor must be positioned on top of the vehicle for better detecting the obstructions around.

4.4 Air module obstacle avoidance

We have presented an algorithm which efficiently finds paths in 3D while minimizing distance & risk and maximizing the path's smoothness [18]. It was now time to simulate it using the hybrid model. For that purpose, the QGC Plan View [19] was used to plan a mission, as can be seen in figure 24. It commands the vehicle to take off at an altitude of 3 meters and then to go to a certain waypoint, which is also located at an altitude of 3 meters and 22 meters far from the initial position of the robot, in the positive x direction, which is represented by the red axis in Gazebo. The world in which the simulation was performed has several trees distributed along the environment, as the ones seen in figure 11a.

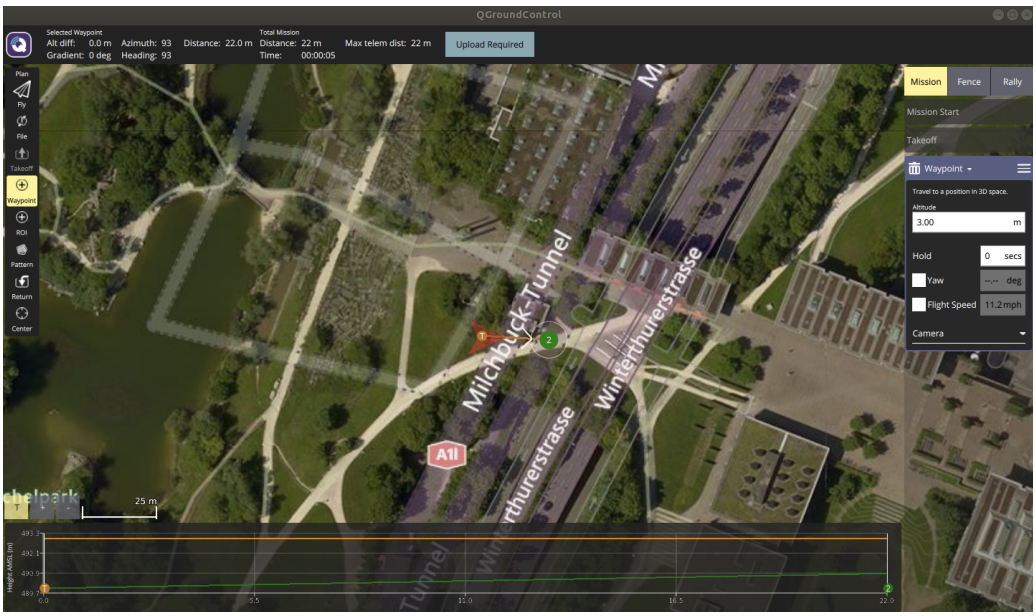
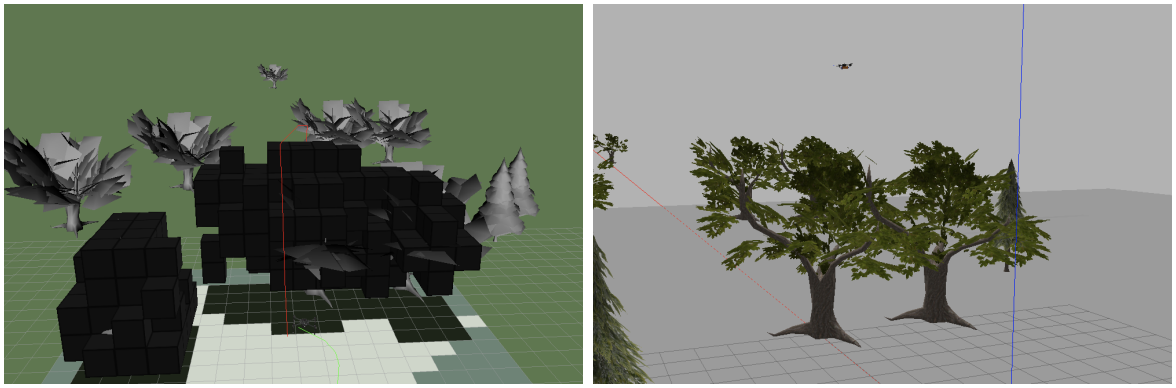


Figure 24: Planned mission in QGroundControl

After starting the planned mission, the vehicle starts moving and finds a tree on its way. It then tries to compute the most optimal path and decides to surpass the obstacle by flying over it. In figure 25a, one can observe the drone finding a new path to overcome the obstructions, where the red line represents the path the algorithm calculated in order to reach the goal point. Moreover, a picture of the drone flying above the tree in Gazebo is displayed in figure 25b.



(a) Vehicle computing and updating its path (b) Vehicle flying over the tree

Figure 25: Different moments of the simulation showing the performance of the algorithm

As a final illustration, the state of the vehicle was also plotted with respect to time. Except for some oscillations, it can be observable that the algorithm computed a smooth path for the robot to follow, while avoiding obstacles. This tree is located at x = 10 and y = 0 meters, being that the reason why the vehicle starts increasing its altitude when approaching these coordinates.

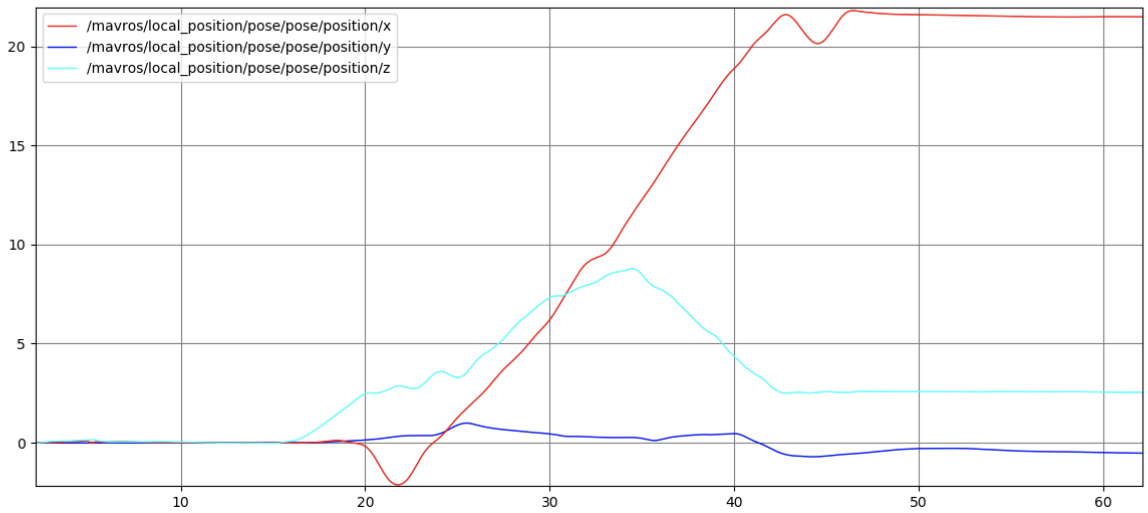


Figure 26: Time evolution of the state of the vehicle

As said before, the simulation tests are done using the open-source 3D simulation environment Gazebo, which allows the use of sensor models providing the necessary sensor data. Then, the artificial sensor data is published as a ROS topic. This means that from the point of view of this global_planner, whether the algorithm is used with simulated sensor data or data coming from a real camera, everything stays the same. Therefore, it can be directly implemented on the real hardware. The only constraint is that the camera which is going to be used on the real vehicle has to provide 3D point cloud data. Some examples are the Intel Realsense D415 and R200.

4.5 Low battery simulation

The simulated battery of the PX4 SITL models in Gazebo only deplete to 50% of its capacity, by default, thus being implemented to never run out of energy. However, as stated in the beginning of this report, one of the parameters which will dictate the motion of the vehicle is the battery percentage. Therefore, a publisher node was created in order to change the battery percentage to 40%. At the same time, another node was created. The latter is very similar to the ones that have been used before. It puts the vehicle in Offboard mode, arms it and then sends setpoints requesting the vehicle to go to an altitude of 3 meters while x = y = 0. Furthermore, it adds an extra condition, which says that, in case the percentage of the battery is lower than 45%, the node stops sending setpoints. Consequently, the vehicle goes into Failsafe mode, which is exactly what would happen in reality in case the robot was running out of energy. The results can be observable in figure 27.

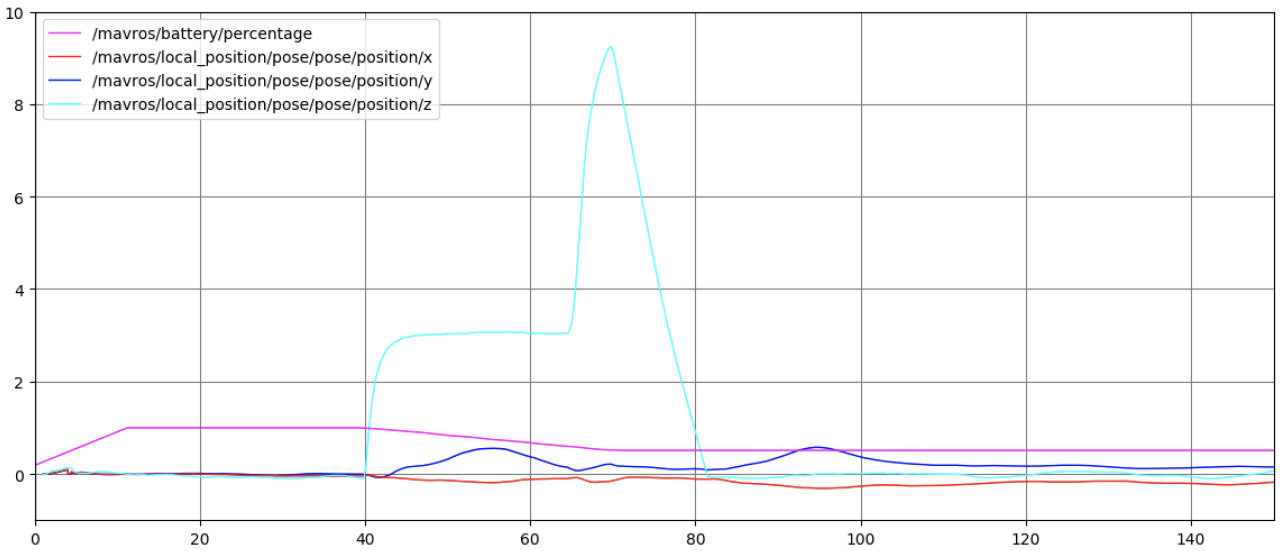


Figure 27: Time evolution of x, y and z for low battery simulation in middle of a flight

When the simulation starts, the battery percentage, which is obtained from the topic */mavros/battery/percentage*, starts increasing until it gets to 1, which means it is fully charged. Then, at around t = 40s, the node which is responsible for sending setpoints to the vehicle brings it to an altitude of 3 meters. Because the vehicle took off, the battery percentage slowly starts decreasing until it gets to 50%, which is the minimum it can get in simulation. At approximately t = 65s, the node which is responsible for publishing fake battery information was ran, which resulted in the battery percentage dropping to 0.4, which means 40%. At that moment, the other node stops sending setpoints and the vehicle goes into Failsafe mode, which results in it going to an altitude of 10 m and then landing.

4.6 Combination of all the algorithms

Now that all the different elements regarding path following and obstacle avoidance algorithms had been explored, it was time to investigate how everything could be combined. Figure 28 shows how the complete algorithm should perform. This one always gives priority to the air module, which will always perform the required path much faster than the rover. In addition, it behaves differently in case there is a need for being more energy efficient. For example, if we want the battery to last for 1 hour, that should become a constraint on how the vehicle decides to act. Finally, when the battery percentage is lower than a certain value, the vehicle should not fly, for safety reasons.

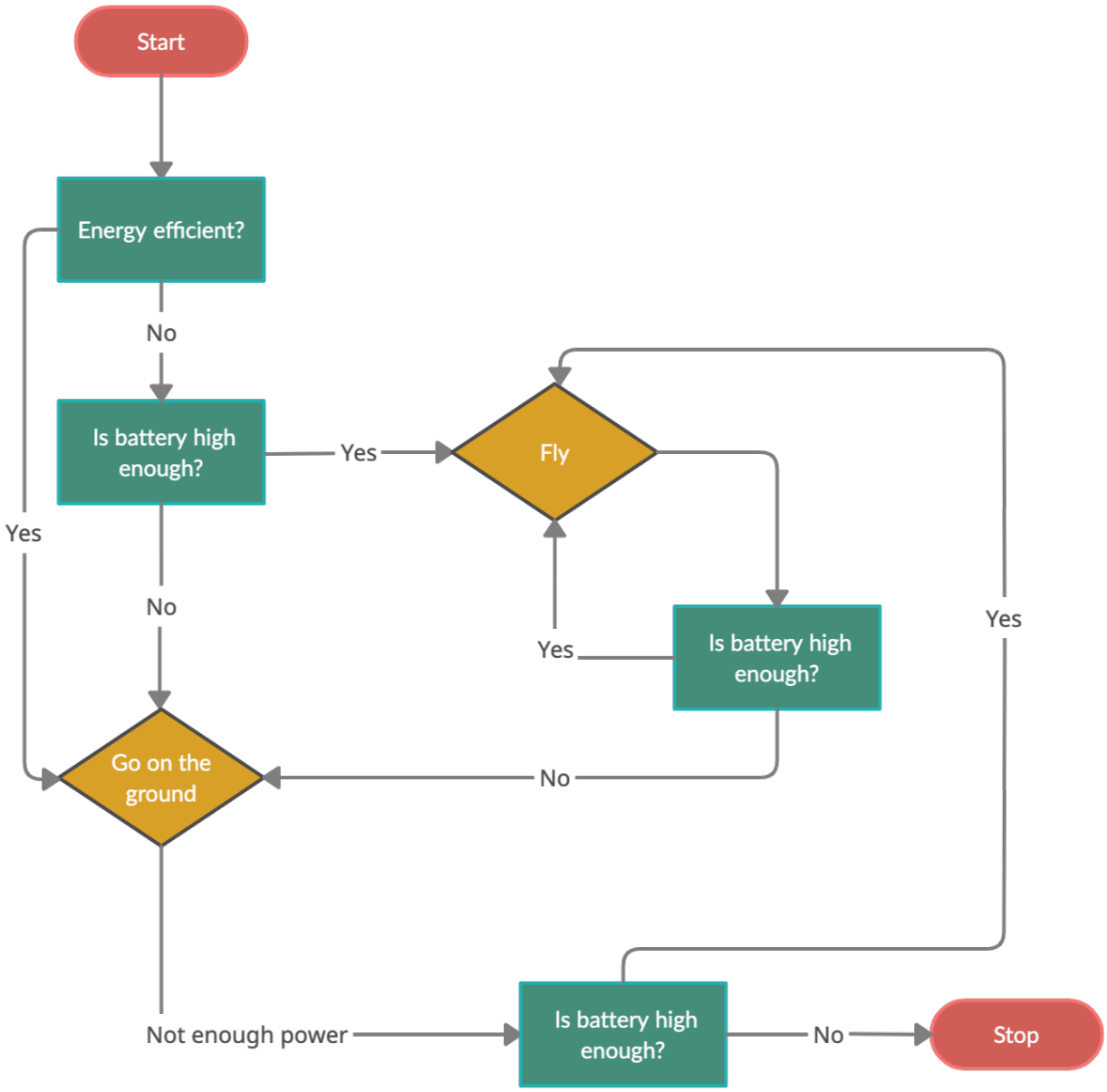


Figure 28: Flowchart of how time, energy and power dictate the motion of the vehicle

4.7 Communication test

All the connections that were shown in section 3.6 were performed and the RPi was connected to the laptop using an SSH connection, as can be seen in figure 29. Afterwards, the MAVROS node was initialized, as well as an offboard node arming the vehicle, putting it in offboard mode and then sending setpoint commands, more particularly asking the vehicle to go to an altitude of 2 meters. Because this meant to be a test for the communication module, the propellers were not added to the vehicle. The test showed very good results, since the motors responded to the commands and started spinning. However, even though power was being provided from the TELEM2 port and also the PX4 servo rail, the RPi did not seem to have enough power for running the required code. Therefore, an extra USB connection had to be made between the Pi and the laptop.

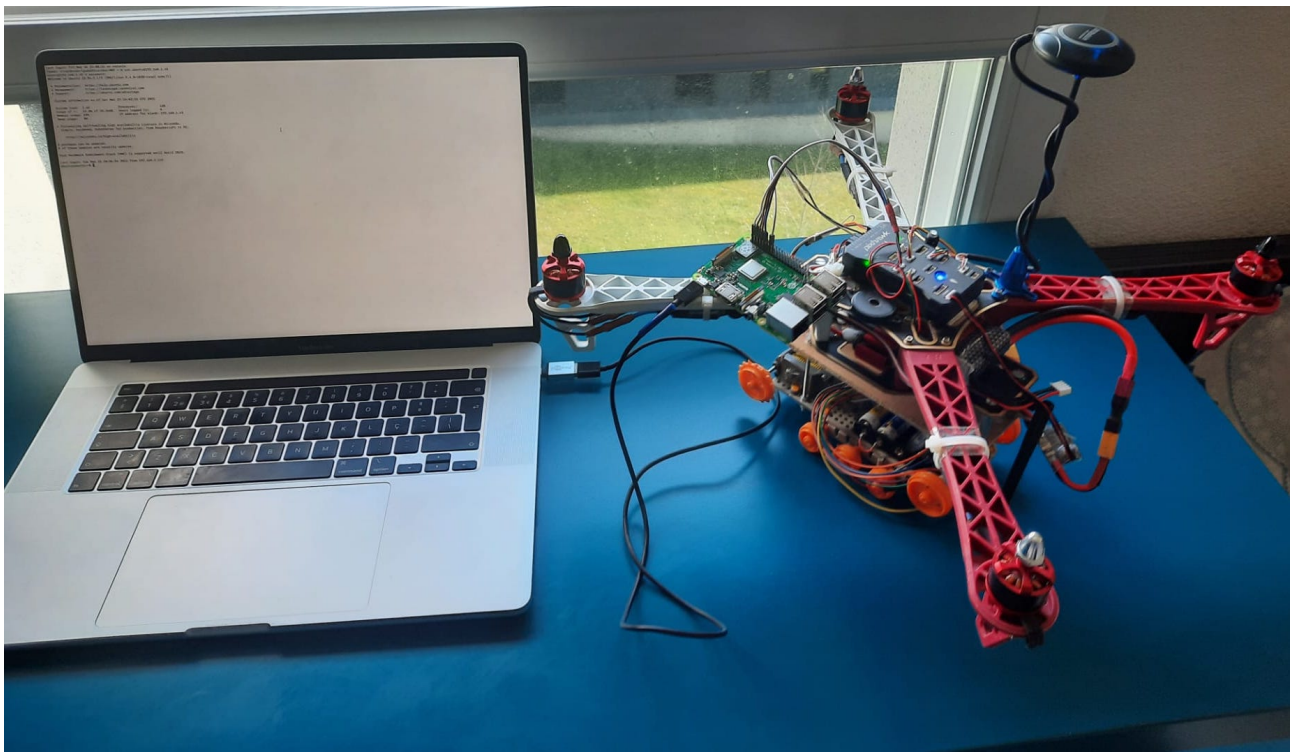


Figure 29: Set up of all the different connections between the RPi and the Pixhawk flight controller

5 Conclusion and perspectives

The project was first handed to me and my partner Brian with the vehicle created and with working computers to control both the aerial and ground modules. The research that was done was then focused on finding and implementing obstacle avoidance algorithms on the ground and aerial module. This involved deciding which type of planner would work best for our current vehicle and then simulating these algorithms on a PX4 autopilot controlled vehicle. After literature review, it was decided that a global planner was better suited for the drone due to its ability to guarantee an optimal path to the goal. In addition, an obstacle following mechanism was chosen for the rover.

With everything implemented, the next step was to create a realistic simulation environment, in view of the fact that it is necessary to test the vehicle in a safe environment in order to prevent damage to the physical vehicle. Gazebo was chosen due to its open source framework, its large support system and its vast range of robotics compatibility. In addition, ROS was chosen as an offboard API due to the fact that it is largely used in the robotics community thus having a vast amount of libraries which are necessary for implementing all the algorithms. This also gave us the ability to create nodes simulating the PX4 autopilot software and allowing to create real like missions of predefined waypoints in QGroundControl, which can then be given to the obstacle avoidance algorithm.

Another very important task was also to create a communication module for the vehicle, which allows to have a link between the PX4 flight controller and the onboard computer, the latter being responsible for acquiring sensor data and deciding how to act on the robot. This was achieved by means of MAVROS, which is a ROS package that enables MAVLink extendable communication between computers running ROS and an autopilot, ground station, or peripheral.

At the end of this research project, the vehicle has initial path planning and obstacle avoidance algorithms that are ready for more simulation tests and initial hardware tests. So as to get more realistic results, the hybrid Gazebo model which has been created should be modeled in more detail.

Additionally, installing sensors on the final vehicle will be of paramount importance since these will be crucial to perform obstacle avoidance. Furthermore, Hardware In the Loop tests must also be performed, which will give a better understanding on how to better integrate these algorithms on the physical hardware.

Moreover, a safe landing planner classifying the terrain underneath the vehicle should also be investigated, since this is also a very important algorithm for the vehicle.

Finally, now that the communication module between the PX4 and the RPi has been established, it is of paramount importance to create a final algorithm articulating all the other mechanisms that were explored. This one will allow the vehicle to switch to the module that better suits the robot for a certain task. The change of the operating module should also depend on time-efficiency and energy consumption, as stated before. After these works have been concluded, it will then be possible to have a working vehicle which is able to conduct an autonomous mission.

References

[1] A. Sambalov, “Autonomous robotic aerial vehicle: S3 project report”, March 2020.

[2] A. Jones, "Chang'e-4 begins lunar day 7 after Yutu-2 rover overcomes cosmic challenges - SpaceNews", SpaceNews. [Online]. Available: https://spacenews.com/change-4-begins-lunar-day-7-after-yutu-2-rover-overcomes-cosmic-challenges/.

[3] "Mars Helicopter to Fly on NASA's Next Red Planet Rover Mission", NASA. [Online]. Available: https://www.nasa.gov/press-release/mars-helicopter-to-fly-on-nasa-s-next-red-planet-rover-mission.

[4] "Documentation - ROS Wiki", Wiki.ros.org. [Online]. Available: http://wiki.ros.org/Documentation.

[5] "Gazebo", Gazebosim.org. [Online]. Available: http://gazebosim.org/.

[6] "mavlink/mavros", GitHub. [Online]. Available: https://github.com/mavlink/mavros.

[7] "Simulation | PX4 User Guide", Docs.px4.io. [Online]. Available: https://docs.px4.io/master/en/simulation/.

[8] D. Model, "Differential-drive vehicle model - MATLAB", Mathworks.com. [Online]. Available: https://www.mathworks.com/help/robotics/ref/differentialdrivekinematics.html.

[9] "Boundary following" [Online]. Available: https://www.youtube.com/playlist?list=PLp8ijpvp8iCvFDYdcXqqYU5Ibl_aOqwjr.

[10] "Controller Diagrams | PX4 User Guide", Docs.px4.io. [Online]. Available: https://docs.px4.io/master/en/flight_stack/controller_diagrams.html.

[11] "Obstacle Avoidance | PX4 User Guide", Docs.px4.io. [Online]. Available: https://docs.px4.io/master/en/computer_vision/obstacle_avoidance.html.

[12] Vilhjalmur Vilhjalmsson, “Risk-based Pathfinding for Drones”, May 2016.

[13] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. Autonomous Robots, 34(3):189–206, 2013.

[14] "Apple Developer Documentation", Developer.apple.com. [Online]. Available: https://developer.apple.com/documentation/gameplaykit/gkoctree.

[15] J. Lim, "Trajectory following with MAVROS OFFBOARD on Raspberry Pi", 404warehouse. [Online]. Available: https://404warehouse.net/2016/08/10/trajectory-following-with-mavros-on-raspberry-pi/.

[16] "Multicopter PID Tuning Guide | PX4 User Guide", Docs.px4.io. [Online]. Available: https://docs.px4.io/master/en/config_mc/pid_tuning_guide_multicopter_basic.html.

[17] "aniket0112/hyrbid_automata", GitHub. [Online]. Available: https://github.com/aniket0112/hyrbid_automata.

[18] "PX4/PX4-Avoidance", GitHub. [Online]. Available: https://github.com/PX4/PX4-Avoidanceobstacle-detection-and-avoidance.

[19] "Plan · QGroundControl User Guide", Docs.qgroundcontrol.com. [Online]. Available: https://docs.qgroundcontrol.com/master/en/PlanView/PlanView.html.