Institut Superior de l'Aeronatique et de l'Espace

Master of Aerospace Engineering Research Project

S3 Project Report

Autonomous Robotic Aerial Vehicle: ARAV Control in Complex Environments

Author:
Akash SHARMA

Tutors:
Raghuvamsi Deepthimahanthi
Yves Briere

Due Date of report: March 29, 2023
Actual Submission Date: April 24, 2023

Starting Date of Project: January 29, 2022

Contents

List of Figures

List of Tables

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.

Date: April 24, 2023

Signature:
Akash SHARMA

Abstract

Unmanned Aerial Vehicles (UAVs) and Ground Robots are based on two pieces of technology that both have their own range of applications in several industries. These applications include surveillance, topography mapping, terrain exploration, videography and photography to name a few. This project which was started five years ago has the goal to merge these two technologies to create a hybrid vehicle capable of both terrestrial and aerial operations. One of the main objectives of this vehicle is to be able to navigate across unknown and dynamic environments completely autonomously, efficiently as well as safely by avoiding obstacles to find the most logical path to the destination. This path may or may not involve the use of both modes of transport available to the vehicle which is a decision that it should make on its own.

Considerable progress has already been made in this project. Several algorithms that implement different aspects of the functionality of the vehicle have been developed and have reached varying degrees of maturity. These algorithms include obstacle avoidance, path planning, hybrid module communication etc. An initial prototype has also been developed and tested with somewhat mixed results. Within the scope of this year's project, the goal is to focus on making the vehicle fully functional as well as optimization of the hybrid module.

This report presents the results as well as the methodology used to optimize and test the algorithms and the performance of the vehicle.

Keywords: Hybrid Vehicle, ROS, Path-Planning, Gazebo, Hybrid Module, Path Control

1 Introduction

Drones have brought about a technology revolution because of their vast range of applications. The design and kind of technology used to build them depends on these applications such as surveillance, exploration, delivery, topography mapping etc. Ground robots find many uses in space missions for collecting soil samples and analysing the terrain. This is because they are designed to be functionally robust in order to be able to traverse rough and unforgiving terrains. Unmanned Aerial Vehicles are another class of drones that can function without an onboard pilot. They are also used to map topography, collect land survey data for infrastructure projects, fertilizer distribution in agriculture, videography and photography projects etc. Finally, smart robots or autonomous robots that are capable of carrying out complex tasks without the need for human guidance have also emerged as a dominant piece of technology.

This project aims to design, program and test an unconventional design of an Autonomous Robotic Aerial Vehicle that combines the locomotive modes of ground as well as air, hence, making the vehicle more robust in handling mission trajectories in complex environments on its own. In essence, the vehicle demonstrates the technologies implemented in ground robots, UAVs as well as autonomous robots. The vehicle is composed of ground and air modules that activate depending on the decision taken by the vehicle on which path to take. The vehicle uses sensors and a set of algorithms to carry out different challenges during the mission such as obstacle avoidance, safe landing, communication between the air and ground modules etc.

Within the scope of this research, the objectives for the new semester are the following:

- This project has been worked on by two groups of students last year who developed different methodologies for the functionality of the vehicle. Therefore, this year's work has been focused on further developing and merging the two solutions to create a more mature software architecture.
- The path planning and path control algorithms have been further investigated for development to expand for different test cases.

2 Project Definition

2.1 Context and Key Issues

This project has come a long way from just an idea of the merger of different technologies to being in a stage where we have a partially functional prototype. In order to see this research to its fruition, several key challenges have been identified that are the main roadblocks to be overcome to deliver the final functional vehicle.

- **Master Algorithm Development**: The main challenge in this project is to bring together all the individual algorithms that function as the intelligence of the vehicle. This decision-making algorithm will allow the vehicle in real time to optimise its functional capability of obstacle avoidance, path-planning etc and be able to complete missions more efficiently. Figure 1 shows the desired behavior of the vehicle.

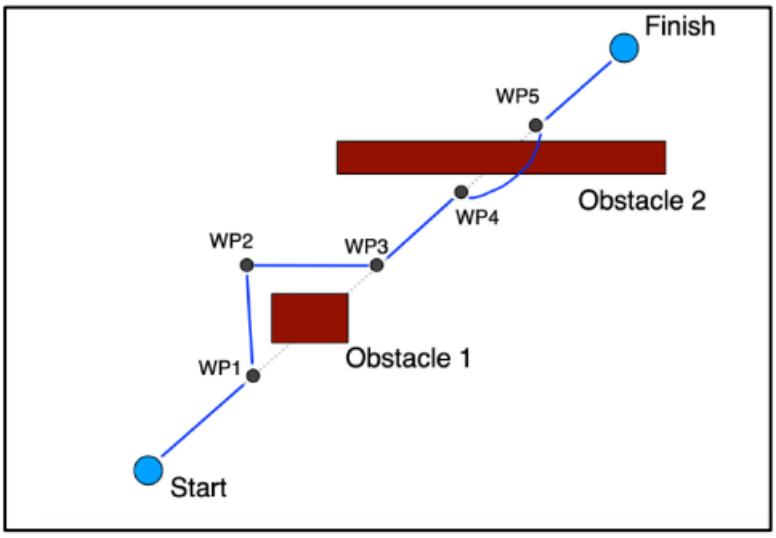


Figure 1: Desired Behavior of the Vehicle. Source [1]

- **Hybrid Module Optimisation**: The choices between the two modules should be made by the vehicle after taking into consideration the optimisation parameters: energy and time. The aerial module would take much less time, but it consumes much more energy, so a trade-off must be made. As of now, the two air and ground modules function properly independently. However, development is needed to be able to switch effectively, seemlessly and safely between them while reducing switching times so the vehicle gets more time to actually navigate through the environment.
- **Weight and PID Tuning**: Another issue to be addressed will be the weight of the vehicle in the context of its aerial performance. The air module is controlled through a PID controller. Currently, the air module has not been tested due to the heavy weight of the vehicle making the PID inadequate in its operation. A thorough analysis must be conducted to find why the vehicle stopped functioning adequately.
- **Real Test Validation**: Due to the heavy weight of the vehicle, none of the aerial tests were performed with the real vehicle. Hence, the previously completed algorithms must be tested in the drone lab along with the new cases to gain confidence of the vehicle's robustness in every scenario.

2.2 State of the art

One of the main objectives of this literature review was to identify decision making algorithms used to optimize the functional capabilities of an autonomous vehicle. These algorithms could be used in the scope of applications other than path selection as well since the idea was to establish their utility.

Po-Lung Yu, in his book, Multiple-Criteria Decision Making [2], talks about non-trivial decision making when several criteria are in play. The best decision is made based on a set of alternatives or choices, the outcomes of each of those choices, the set of criteria that are imposed on the decision to be made and finally, the preferences of some outcomes over others. A decision-making algorithm based on Double Deep Q-Learning with Prioritized Experience Replay has been introduced by Kun Zhang et. al. [3] which is used for autonomous UAV maneuvering and route guidance. There has also been extensive work done by the Multi-Robot-Systems group at the Czech Technical University in Prague who published a paper presenting a multirotor UAV control

and estimation system which has been tested through simulations as well as experiments [4]. The algorithms developed by them are open source and have been investigated for possible usage for this project [5].

Rovers and drones are used extensively to scout and explore terrain in remote locations or places with hard accessibility. The most popular examples of exploratory drones and rovers are NASA's Mars Helicopter, Ingenuity [6] and the rover, Perseverance [7], as well as the Chinese rovers, Yutu and Yutu-2 [8] which have been exploring the Moon in recent years. Although scarce, some research of unconventional designs of drones similar to our prototype has also been carried out by the Aerospace Robotics and Control Lab at Caltech consisting of a bipedal walking and flying robot [9].

2.2.1 ROS Architecture

The technology revolution and push towards innovation in the field of automation and electric vehicles has created a huge variety of tools, particularly open-source software that have seen a rapid growth in development and usage. This is because these tools offer simple license management and the source code is well designed by communities of developers who also offer abundant support to the industry as well as individual users. The Robot Operating System (ROS) is the most widely used open-source development platform for development of robots and autonomous vehicles [10].

ROS is an open source development kit composed of different libraries and applications for robotics applications. It provides a software platform that can be used for research and prototyping as well as deployment and production. It supports usage of multiple languages and has been designed to be easily integratable with any project.

The underlying principle of ROS is the implementation of **ROS Nodes** which are essentially files of code that are executed when the node is run. These nodes can be written in different languages and still be run simultaneously. Additionally, instead of calling different nodes in order to share data with each other, these nodes communicate via **ROS Topics** that act as channels storing **ROS Messages** that are sent and received by these nodes. In order for a node to participate in communication, it has to either **publish** on a topic or **subscribe** to a topic. This kind of point-to-point architecture builds a network of nodes that works together to carry out different aspects of the robot's functionality. Figure 2 illustrates these concepts through a schematic.

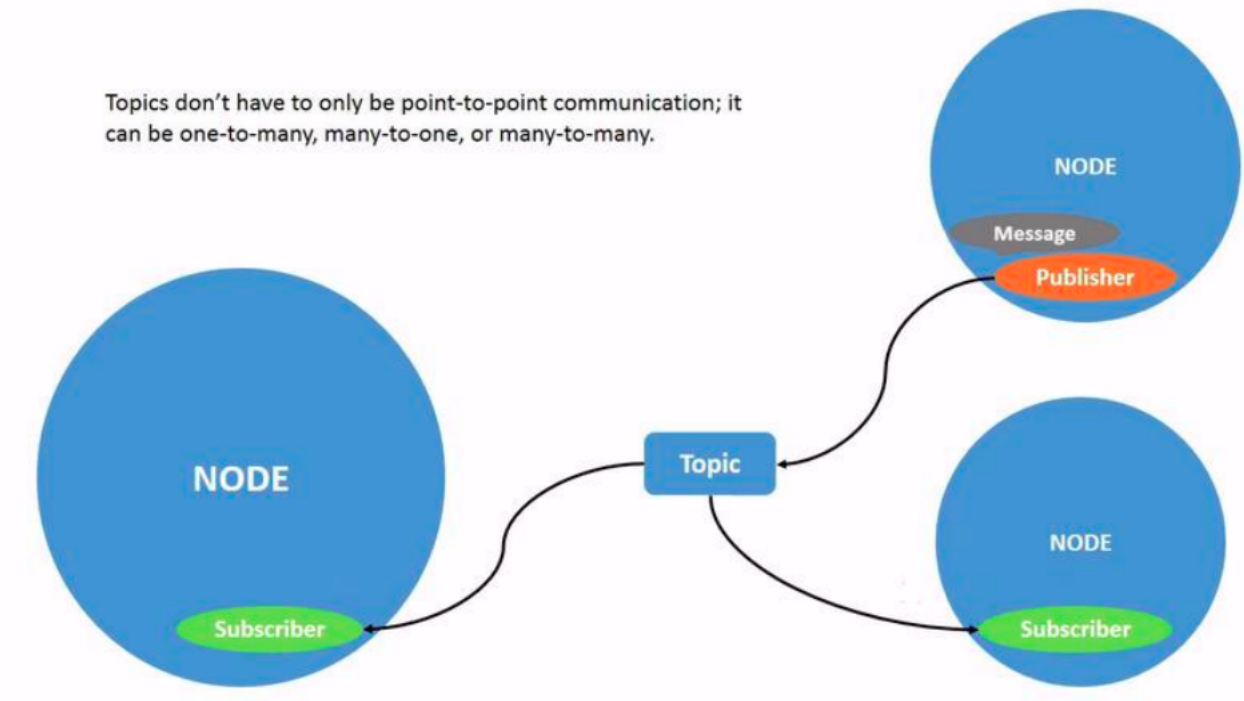


Figure 2: ROS architecture. Source [11]

2.2.2 Initial State of the Project

This project was handed over to this year after making significant progress. The prototype required for real world testing has been fully assembled comprising of the aerial and ground modules attached together so that

during the flight mode, the ground module basically acts as a payload. This prototype was too heavy to be flown with the current configuration of the Pixhawk PID flight controller. Figure 3 shows the real prototype developed by the previous students [12][13] as well as how the two modules are attached together.

(a) Final Prototype

(b) Air and Ground Module Attachment

Figure 3: Fully Assembled Real Vehicle. Source: Taken by the author

Preliminary research had been conducted on a set of algorithms referred to as modules, which have been simulated and later fully developed by the previous students [13][14][15][1]. However, most of them have only been tested on the ground module. Table 1 gives an idea about the progress made on building atleast a basic structure of the software architecture of the vehicle.

Algorithm	Ground Module		Aerial Module	
	Simulation	Testing	Simulation	Testing
Path Planning	Done	Done	Done	Not Done
Environment Perception	Done	Done	Done	Not Done
Path Control	Preliminary Stage	Not Done	Preliminary Stage	Not Done

Table 1: Current Status of the Algorithms. Source [1][15]

2.3 Potential Degree of Novelty

One of the main factors that has made the technology of drones and rovers so interesting and useful is that they are mechanically simple. These two kinds of robots have been extensively researched and optimized for a variety of applications. Here on Earth, rovers are used for accessing hard to reach or even dangerous places, for example, for diffusing bombs and going into underground mines. At other planets like Mars, they are used to map the terrain or carry out soil sample analysis. Flying drones offer many capabilities of their own such as aerial surveillance and mapping, cargo delivery etc.

This project has the goal to combine these two applications of different kinds of technology that can open the doors to a whole new range of applications. With two different means of transport at its disposal, the vehicle can drastically increase its range of area that it can cover. Moreover, the vehicle is also being designed to be fully autonomous which would be particularly useful when carrying out missions that involve uncertainty since the vehicle is equipped to handle such situations. Exploration of other planets with this kind of vehicles will also remove the problem of communication and command signal latency.

2.4 Objectives

The final objective of this project is to produce a hybrid autonomous vehicle that has the capabilities of a UAV as well as a ground rover and which can navigate through complex environments by assessing its surroundings and making intelligent decisions. It must be able to decide which path and mode of travel is best suited taking into consideration the power consumption, mission time, nature of the environment and the destination coordinates.

Two separate routes of research were taken by two pairs of students last year to develop the software architecture of this vehicle. Since the two sets of algorithms were developed in different ROS versions, it would be tedious to

merge the two algorithms together to test them in parallel. Therefore, each semester was dedicated to exploring and optimizing one software architecture developed by [1] and [15].

A set of objectives have been defined for the second semester when the software architecture developed by [15] is studied and further developed. These are as follows.

- **Path-Planning Optimization**: The path-planner needs to be further developed to be able to plan increasing complex paths for more complex environments such as planning a mixture of ground and aerial paths.
- **Latency Reduction**: The algorithms need to run with minimal latency so as to be deployed for real time operations.
- **Path Control Optimization**: The path control module will be expanded to include an aerial controller and a more robust and stabilizing ground controller.
- **Test Case Expansion**: The system needs to be simulated in the presence of an environment of increasing levels of complexity to analyze whether it continues to be able to adhere to the constraints.

3 Hardware and Software Description

3.1 Vehicle Specification and Description

The robotic aerial vehicle designed for this project has been upgraded throughout its 4 years of life. This vehicle is a hybrid robot, capable to moving on the ground as well as flying. In order to achieve this, it comprises of 2 different but interconnected modules – the air and the ground module that are managed by a single embedded computer within the vehicle. This section details all the important parts of the vehicle along with their descriptions and functions.

There are several components which are responsible for different functions pertaining to both the modules in the vehicle. These are as follows.

- **Embedded Computer**: The embedded flight computer is responsible for establishing a connection with the external server as well as with the flight controller (Pixhawk [16]) and the ground motor driver. The embedded vehicle computer model is a Raspberry Pi 3 Model B+ [17], which is a single-board computer with a 1.4GHz 64-bit quad-core processor using the arm64 architecture. A 3D printed protective casing has been designed to house the Raspberry Pi on the vehicle.
- **Raspberry Pi Power System**: It is necessary to constantly power the RPi through an external battery with an appropriate capacity and power. A 10000 mAh capacity battery, with a USB-A exit with a 5V-2A performance has been chosen.
- **Environment perception sensors**: For the critical task of understanding the vehicle's environment, it is equipped with different sensors. Two USB stereo cameras based on the OV2659 chipset have been acquired for real time depth perception. Three HC-SR04 ultrasonic sensors are responsible for calculating distances from certain objects that may be obstacles. These sensors have been mounted onto the vehicle frame using 3D printed holders by the previous students [15].
- **Voltage Potentiometer and Adaptor**: Since the voltage of the aerial battery is too high for the ground motors (11.1V compared to the specified 6V) a voltage potentiometer must be used. For this prototype, a DC-DC converter (LM2596 Model) is used which can be regulated allowing the fine-tuning of the voltage amount received by the ground motors. An adaptor component has been designed to install the potentiometer within the vehicle frame.
- **Spherical Markers**: The positioning system present in the ISAE Volière laboratory has been used for the testing of the vehicle, which consists of an OptiTrack [18] motion capture system that uses several cameras around the laboratory environment which detect several spherical markers using a range of cameras located around the environment. These spherical markers on the vehicle provide high accuracy position and orientation of the vehicle.

3.1.1 Aerial Module

The air module takes the form of a quadcopter configuration. The components involved in this module are mentioned below.

- **Drone Frame**: A F450 Drone frame kit with is an X-type frame with a 450mm wheelbase is used for this project.
- **Power Distribution Board (PDB)**: This is included in the F450 Drone frame kit.
- **Motors**: The vehicle has four 2212 920kV brushless DC motors which means that they have a stator width of 22mm and a height of 12mm.
- **Propellers**: Propellers of type 9450 are used (9.4-inch diameter, 5.0-inch pitch) with the motors.
- **Electronic Speed Controller (ESCs)**: 4 of the 30A ESCs are used in this project.
- **Aerial Battery**: A Floureon 3S, 11.1V, 3000mAh, 30C LiPo battery is used to power up the vehicle.
- **Flight Controller**: The vehicle is equipped with Pixhawk which serves as the flight controller.

3.1.2 Ground Module

The ground module design is adopted from that of a tank which enables control on rough terrain and lessens the chances of it getting stuck. The components in this module are as follows.

- **Rover Frame**: The Tamiya universal plate set is used as a frame.
- **Tracks and Wheels**: The ground module is based on the Tamiya 70100 track and wheel set.
- **Gearbox**: The Tamiya 70097 twin-motor gearbox consists of the gearboxes and two independent brushed DC motors, which are used to drive two shafts. The motors run on 3-6V.
- **Power System**: The Floureon 3S, 11.1V, 3000mAh, 30C LiPo battery is used (The same battery powers the drone propellers as well as the rover motors).
- **Motor Driver**: The ground module connections enable the communication between the Raspberry Pi and the TB6612FNG motor driver, as well as the connection of this driver with the ground motors and battery. The TB6612FNG motor driver controls the speed and direction of the two rear motors with a supply range of 2.5V to 13.5V.

3.2 Simulation Platforms and Software

The final objective of the vehicle is to be completely autonomous. It should be able to navigate through complex environments and avoid obstacles by perceiving the environment around it and making decisions without any human input. In order to make this happen, several algorithms, referred to as modules, have been designed which are responsible for specific aspects of the functionality of the vehicle. Additionally, a strong development platform is needed to build the underlying software for such a hybrid vehicle as well as carry out simulations to test the algorithms. The development tools used for this project are detailed below.

- **ROS – Robot Operating System**: ROS [10] has been used for developing the different modules that manage various functionalities of the vehicle, from path-planning and obstacle avoidance to high level ground and air control. As mentioned in Section 2.2.1, ROS is an open source development environment comprised of libraries, applications and different tools that can be utilized to develop and customize any kind of robot. ROS operates on a point-to-point architecture where several nodes communicate messages with each other through communication channels called topics. This entire network of interconnected nodes provides the base of the vehicle software.
- **Gazebo**: Gazebo [19] is the chosen 3D simulator for testing the vehicle in simulation before the real test. It is another open source real world simulator which can realistically simulate scenarios in indoor as well as outdoor environments. It offers a robust physics engine which makes simulating collisions easier. It can also simulate sensor feedback with noise through its various sensor models. Gazebo is also compatible with ROS which makes its utility much more convenient.
- **Pixhawk (PX4)**: Pixhawk [16] is the flight controller installed on the vehicle. The controller receives sensor data and adequately sends motor and actuator values. A mission using Pixhawk is carried out through a state machine. Each phase of the flight is activated only when the vehicle has successfully gone into that state. For example, the vehicle needs to go into the **Hold Mode** in order to be able to hover at any point in space or go into **Land Mode** to land the vehicle. Since the vehicle is planned to be commanded by an offboard API which is ROS, the vehicle needs to go into **Offboard Mode** before ROS can run its nodes which supply Pixhawk with the destination coordinates.
- **QGroundControl (QGC)**: The QGC station [20] is useful for real time monitoring and mission planning. This tool gives a comprehensive picture about the vehicle state, power and battery consumption which is useful to study the decision-making capabilities of the vehicles when it tries to conserve battery while navigating an optimum path.

It is necessary to simulate the performance of PX4 in the Gazebo simulations as well before carrying out real tests. For this purpose, the **MAVROS** package [21] available in ROS is used to establish communication between ROS and Pixhawk. This package enables the **MAVLink** communication protocol [22] which is the supporting bridge between ROS and PX4. It is a binary telemetry protocol that must be enabled in all drones using Pixhawk. PX4 communicates with Gazebo to receive sensor data from the simulated world and send motor and actuator values. It also communicates with QGC or an Offboard API like ROS to send telemetry from the simulated environment and receive commands. Figure 4 provides a summary of the different elements of the software architecture and how they interact with one another. The ports mentioned in the diagram refer to the connections to be made to integrate Pixhawk to any real vehicle.

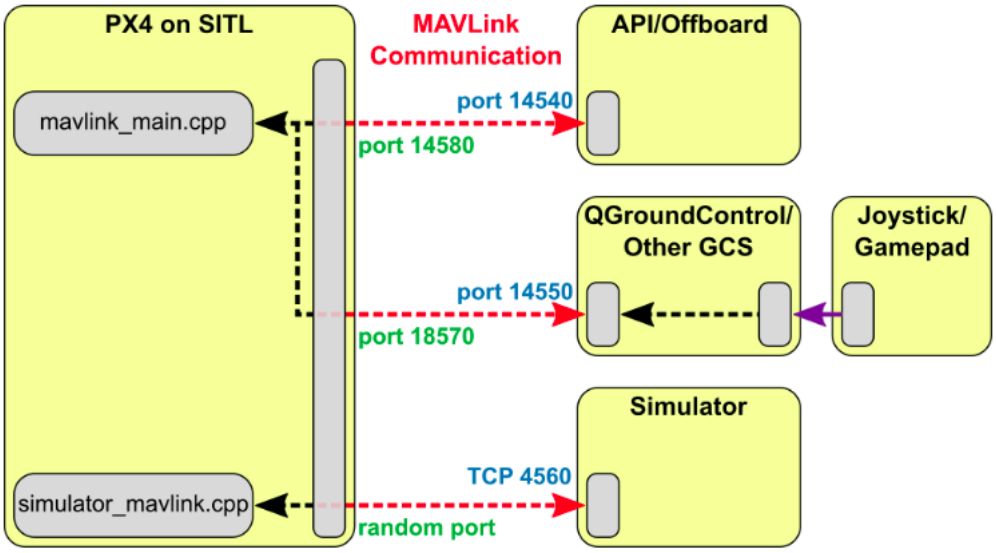


Figure 4: Software Architecture of the vehicle (GCS - Ground Control Station) Source: Pixhawk User Guide [16]

4 Progress - Semester 2

The work of semester 2 was dedicated to improving the algorithms developed in ROS Melodic by the first pair of students [1]. In addition to this, the issues that persisted from last year in the actual prototype were also investigated.

4.1 Investigation of Persistent Hardware Issues

The first task which was addressed was the problem of the aerial module of the vehicle which was not able to fly. This was, apparently, due to the high weight, thrust saturation and vertical oscillations that were observed in the tests. In order to solve this, a complete test analysis was conducted of all the past tests carried out by the previous students. These tests revealed that the vehicle was not tuned at the right weight which had led to the oscillations. As for the increased thrust capacity, the current propulsion system of the the battery, motors and propellers was investigated and two solutions were proposed. The 2 ways to solve this problem are either to reduce the weight of the vehicle or increase the thrust capacity of the current propulsion system. The weight can be reduced by removing one of the batteries powering the Rasperry Pi and having it receive power from the other battery directly. This reduced the thrust capacity to 74% from the previous 80% with a battery life of approximately 11 minutes. The second solution deals with the type of propellers used. Larger propellers are proposed which can generate the same thrust at a lower RPM. For the weight of the vehicle, APC 1045 propellers were proposed which bring the thrust capacity down to about 67%. Figure 5 provides the thrust performance data for the 2212 920kV DC motor used in this project.

ML2212 MOTOR

Item NO.	Volts (V)	Prop	Throttle	Amps (A)	Watts (W)	Thrust (g)	Efficiency (g/W)	Operating temperature(℃)
ML2212 920KV	11.1V	DJI9.4*4.3	50%	1.8	20.0	230	11.5	37℃
			65%	2.8	31.1	310	10.0	
			75%	3.9	43.3	410	9.5	
			85%	5.5	61.1	480	7.9	
			100%	7.6	84.4	610	7.2	
		APC10*4.5	50%	2.6	28.9	290	10.0	55℃
			65%	5.1	56.6	460	8.1	
			75%	7.4	82.1	590	7.2	
			85%	10.1	112.1	730	6.5	
			100%	13.4	148.7	860	5.8	
	14.8V	DJI9.4*4.3	50%	2.7	40.0	350	8.8	52℃
			65%	4.4	65.1	490	7.5	
			75%	6.3	93.2	640	6.9	
			85%	8.3	122.8	790	6.4	
			100%	11.5	170.2	990	5.8	

Notes: The test condition oftemperature is motor surface temperature in 100% throttle while the motor run 10 min.environment temperature 24℃

Figure 5: Thrust Performance Data of the 2212 920kV motor

4.2 Hybrid Module Optimization

The development of the hybrid module was carried out with the objective to minimize energy consumption while having the vehicle adhere to a strict timeline in which it had to reach the final coordinates. With this in mind, the algorithm was optimized in different levels where each level improved upon a certain aspect of the performance of the vehicle.

4.2.1 Adding the Flight Domain

In the first level for this hybrid drone, minimizing the energy translates to prioritizing the ground module over the aerial module. The aerial module should be used if the vehicle needs to overcome many complex obstacles while still reaching the target on time. Therefore, to impose an urgency over the vehicle to reach the target, the concept of a flight domain was introduced. Put simply, this flight domain is a sphere with a continually increasing radius centered at the final coordinates. The vehicle must fly while it is inside this domain and

drive if it finds itself outside it. The exception to this rule would be if it encounters an obstacle outside the flight domain that is impossible to overcome without flying. The rate of increase of this radius is inversely proportional to the time specified by the user for the vehicle to complete the mission. This places a condition of optimality for switching on the vehicle. Figure 6 shows a schematic of the concept of the flight domain.

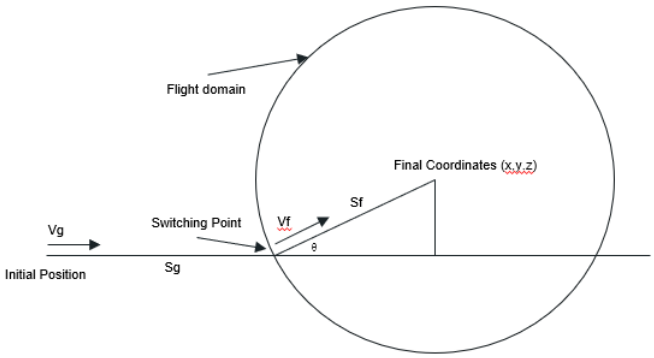
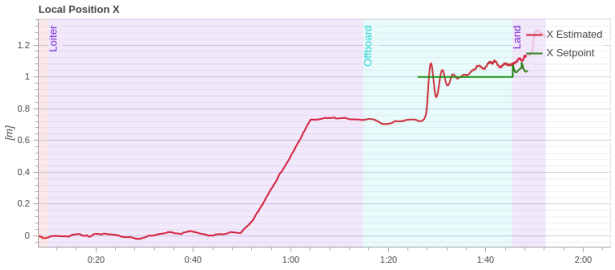
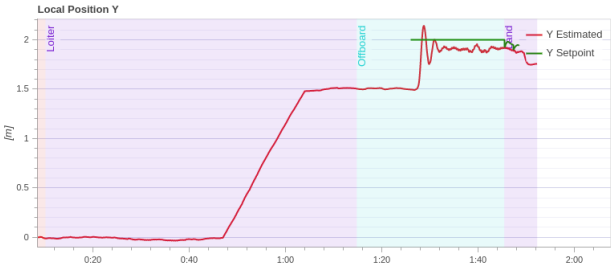


Figure 6: Switching in the Flight Domain. Source: Prepared by the author

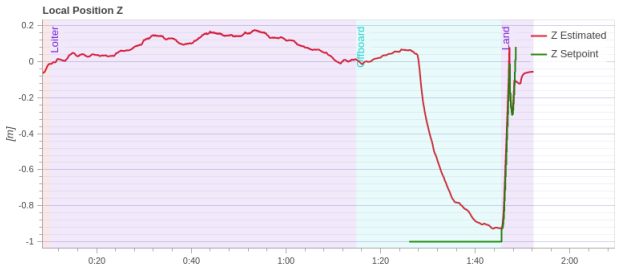
The performance of the vehicle was simulated with the addition of this flight domain and the results were found to be positive. One of the simulations is shown in Figure 7 where the final coordinates entered are (1,2,1) with a timeframe of 60 seconds. The loiter region of the graph represents the part where the vehicle uses the ground module while the offboard region represents the aerial module.



(a) X position vs Time



(b) Y position vs Time



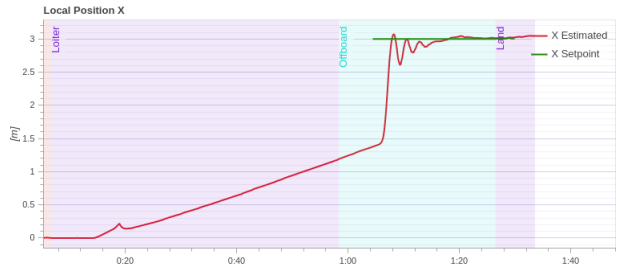
(c) Z position vs Time

Figure 7: Hybrid Module Optimized Trajectory with flight domain. Source: Prepared by the author

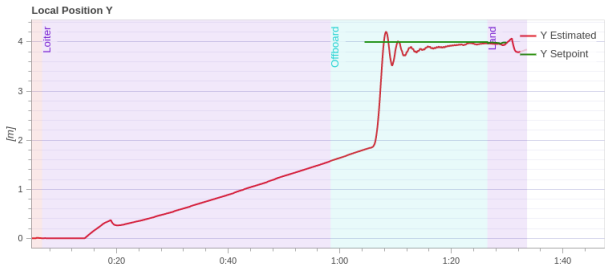
The plots show that even though the final destination is in the air, the ground module is used first so as to prioritize energy conservation until it is time to switch to the air module.

4.2.2 Switching Time

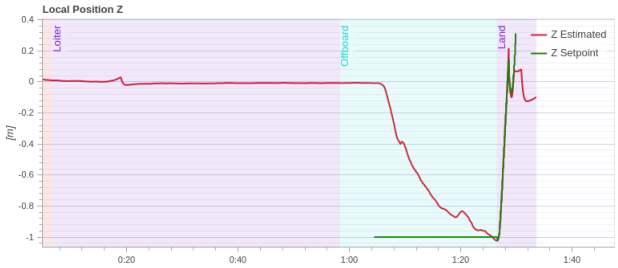
The second level of optimization focused on switching realistically from the ground to the serial module. Switching to the offboard mode of Pixhawk requires several pre-flight and arming checks to be performed during which time the vehicle does not move at all. This period can be observed in Figure 7 where the X and Y values remain constant. Therefore, the the switch was triggered intelligently so that these checks would be performed while the vehicle is still using the ground module and starts flying just as the ground module is deactivated. The results of simulations run after this change are shown in Figure 8. One can observe that the constant X and Y value trend has disappeared and the vehicle switches smoothly.



(a) X position vs Time (smooth switch)



(b) Y position vs Time (smooth switch)



(c) Z position vs Time (smooth switch)

Figure 8: Hybrid Module Optimized Trajectory with a smooth switch. Source: Prepared by the author

4.2.3 Safety Takeoff and Landing Protocol

Finally as part of the third level of improvement, a safety protocol had been added to the optimization algorithm to ensure that the vehicle is safely away from the ground and from other potential obstacles before it starts chasing the target. This protocol states that the vehicle must reach an altitude of 1m before continuing its journey. The vehicle will reach the final coordinates directly after achieving this altitude if they are in the air or fly to the point directly above the final coordinates if they are on the ground before landing.

5 Progress - Semester 3

The main focus of this semester is to further develop the software architecture proposed by Alberto Ceballos and Joan Bessa [15]. Figure 9 shows a flowchart which describes the different modules that have been worked on by the seniors.

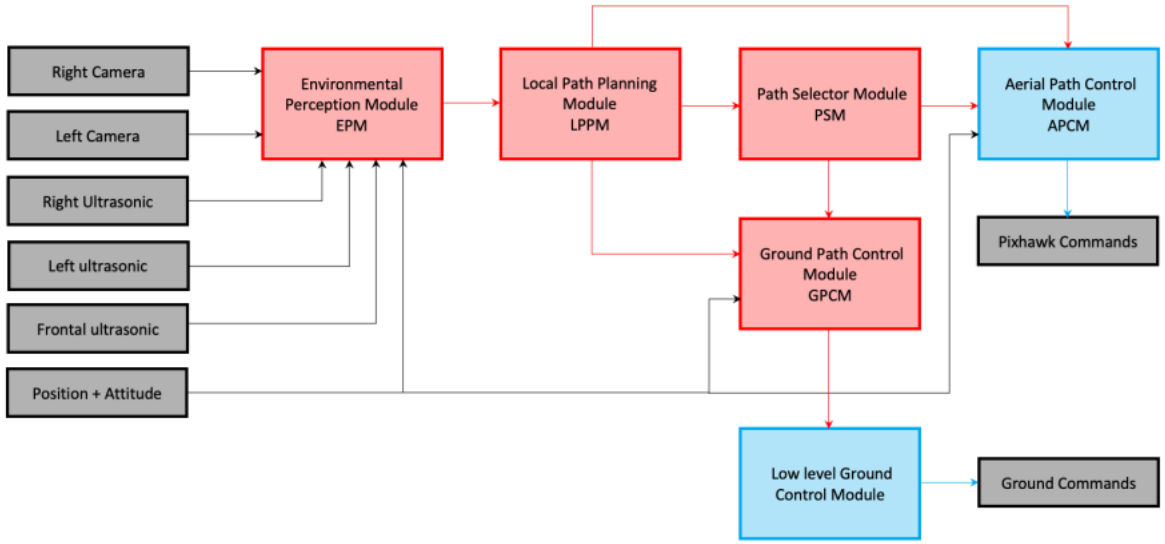


Figure 9: Software Architecture Developed by the authors. Source: S3 Project Report, March, 2022 [15]

The simulation starts by launching the environment with the defined characteristics and obstacles along with spawning the vehicle. As described in Section 3.1, the vehicle is equipped with camera and ultrasonic sensors used to detect various obstacles in the environment. Several plugins have been added to the robot description to simulate these sensors in Gazebo. The **Environment Perception Module (EPM)** receives this data to create a map of the environment using a point cloud data structure. The **Local Path Planning Module** receives information about this mapped environment in the form of an Octree structure and creates a list of waypoints that the vehicle should follow which make up a reasonable path to a pre-defined goal. Currently, this module is computing two sets of waypoints, one solely on the ground and one solely in the air. It sends the respective lists to two different modules called **Aerial Path Control Module** and **Ground Path Control Module**. The **Path Selector Module** is responsible for computing the more optimal path between these two and activating the corresponding control module. These two control modules have their separate low level controllers that control the movement of the vehicle to implement the calculated path.
The following section provides a brief description of the underlying algorithms that are involved in this architecture.

5.1 Functioning Algorithms

This vehicle has the first task to navigate through a complex environment. This means that it needs to understand its environment and identify the optimum path to reach its destination by avoiding different obstacles in the way. The second and equally important task is to do this safely. Several algorithms or modules have been developed that are dedicated to each of these tasks of environment perception, path-planning, path-control etc.

Point Cloud Map Creation

The onboard ultrasonic and stereo camera sensors capture data of the environment to generate a point cloud map of the environment. This is carried out through AI based object detection algorithms using the TensorFlow library. This map is then published to a ROS topic that is subscribed by the octomap_server node.

Octomap Grid Generation

The Octomap_Server package [23] is a ROS package containing the Octomap Server node. This node subscribes to a point cloud message and generates an Octomap [24]. A basic occupancy grid represents 3D space as an evenly spaced grid of cubes. Every cube in the space has a binary identifier, which represents whether the

space it occupies has an obstacle present or not. The package also has more sophisticated functionality wherein instead of just providing a binary identifier, it provides a probabilistic map of the environment where each cube has a probability of occupancy associated with it. The generated octomap is published to a rostopic that is subscribed by the path-planner.

Path-Planning

The path-planner implemented in this project is the Rapidly-exploring Random Trees – star (RRT*) planner. The RRT* algorithm uses Rapidly exploring Random Trees to map the environment and find a suitable path. Once the initial solution is found, it continues to identify more suitable paths, by rewiring the way-points using an optimization logic. In this way, RRT* creates an expanding tree, biased in reaching the solution.
The RRT* planner has been implemented using the **Open Motion Planning Library** (OMPL) [25] which houses many sampling-based path planning algorithms. In addition to this, the path planning module also utilizes the **Flexible Collision Library** (FCL) [26] which is used to detect collisions with obstacles and implement distance computation algorithms. This library is used to keep checking if the computed paths are viable or not after receiving the octomap data.
Once the path-planner has the robot position, the destination coordinates and the obstacle map from the sensors, it computes sequentially a ground path and an aerial path and publishes these as a topic. Along with this, it also outputs the length of each path which is then used by the path-selector module.

Path Selection

The two groups of students who have worked on this algorithm have taken two different approaches to select the optimum path based on power and time constraints.

- **First Approach**: The path-selector module is responsible for making a decision between a ground path and an aerial path. It does this by receiving the length data of the two paths from the path planning module and computes a cost for each of the paths. The path corresponding to the lower cost is selected to be followed. Two main criteria used for this decision making are battery consumption and time taken. This method was adopted by Joan Bessa and Alberto Ceballos [15].
- **Second Approach**: The second approach looks at the z coordinate of the final coordinates entered by the user as an input. If this value is above 0.4m, then the algorithm understands that activating the aerial module is necessary, so the vehicle takes off from its initial position and flies to the destination. Otherwise, the ground module is used throughout the journey to reach the destination. This method was adopted by Karthik Mallabadi and Ryan Dsouza [1].

Path Control

Finally, the Path Control module implements the path chosen by the path-selector. Depending on which path has been chosen, the embedded Raspberry Pi communicates with the Pixhawk flight controller for aerial control and with the Motor Driver for ground control.

5.2 Development Areas in the Software Architecture

After understanding the code developed by the seniors, the objective was not only to just further develop the architecture but also merge useful parts of it with the software architecture developed by the other set of students to achieve the most feasible and logical structure. In these set of algorithms, several key areas of development were identified and have been investigated for potentially better alternative solutions.

Within the software architecture, most of the development was focused on the Path Planning and Path Control modules since several opportunities for improvement were identified in these modules. These points are listed below. The next sections elaborate the work done in these areas.

Mixed ground and aerial paths

The approach of calculating two independent viable paths, one on the ground and the other in the air may work for simple environments but this approach cannot produce a solution where the most viable path consists of a mixture of both paths. Such paths consisting of both ground and aerial points will be referred to as mixed paths. An example where such paths might be used would be when the vehicle encounters a wide obstacle behind another tall obstacle. It will first have to use the ground module to go around the tall obstacle and then

fly over the wide one.

Energy Optimization

No cost function had been defined and provided to the OMPL library to plan paths. So, the library defaults to minimizing the path length in the RRT* planner. However, the true objective of the vehicle is to minimize energy. In standard drones or rovers, minimizing energy translates to minimizing length, provided that the robot has a constant energy consumption rate, however, in the case of a hybrid drone such a translation does not hold.

Nascent Controllers

The software architecture had not implemented the PixHawk aerial controller. Moreover, the ground controller developed was only a simple P-controller and needs further improvement.

5.3 The Path Planning Node

The path_planning_node.cpp is the C++ executable file where the RRT* planner is executed through the OMPL library. The OMPL library requires a lot of information in order to start solving for a viable path. Before the computation, the start and goal coordinates are provided and the octomap data is received by the FCL library which computes the positions and sizes of obstacles. This data is used by the **State Validity Checker** that checks each randomly generated point in space to see whether it is inside an obstacle or not. A state is deemed invalid if it does not overlap with any object. Additionally, certain bounds in 3D space are defined in which the RRT* planner is expected to operate and generate random points.
The following code snippet defines all these pieces of data and provides them to the state-space and space-information pointers "space" and "si" respectively.

```
// Create the state space for path planning
        ob::StateSpacePtr space(new ob::SE3StateSpace());

    // Set the bounds for the R^3 (translation) part of SE(3)
        ob::RealVectorBounds bounds(3);

        bounds.setLow(0,-10);
        bounds.setHigh(0,10);

        bounds.setLow(1,-10);
        bounds.setHigh(1,10);

        bounds.setLow(2,0.1);
        bounds.setHigh(2,10);

        space->as<ob::SE3StateSpace>()->setBounds(bounds);

    // Construct an instance of Space Information from this State Space
        ob::SpaceInformationPtr si(new ob::SpaceInformation(space));

    // Set State Validity Checking for this Space
        si->setStateValidityChecker(std::bind(&isStateValid, std::placeholders::_1));

    // Create the start state (start position)
        ob::ScopedState<ob::SE3StateSpace> start(space);
        start->setXYZ(0,0,0.1);
        start->as<ob::SO3StateSpace::StateType>(1)->setIdentity();

    // Create the goal state (goal position)
        ob::ScopedState<ob::SE3StateSpace> goal(space);
        goal->setXYZ(-10,0,0.1);
        goal->as<ob::SO3StateSpace::StateType>(1)->setIdentity();

    // Create problem instance
        ob::ProblemDefinitionPtr pdef(new ob::ProblemDefinition(si));
```

The first task was to simplify the code so that it no longer computes ground and aerial paths sequentially but rather, computes one final path that the vehicle is expected to follow. This path can be a mixture of ground waypoints and aerial waypoints. To do this, the topics that published the ground and aerial waypoints separately were replaced by a singular topic to which the path is published.
An important point is that the whole Path Selector Module is rendered unnecessary with this simplification since there is only one path being published now. This major modification results in overall decreased computation time.

5.4 Custom Energy Objective

The modifications mentioned in Section 5.3 did the job of reducing computation time as well as accounting for mixed paths. The next task focused on optimizing energy consumption of the vehicle. The OMPL library, by default, uses path length optimization in case no specific objective is defined. But this objective would not automatically translate to minimizing energy in all cases. For example, if the vehicle encounters a considerably wide obstacle, the path length objective would plan a path over the obstacle requiring the vehicle to use the aerial mode which consumes more energy. In some cases, this may very well be less energy consuming than driving the long way around. Therefore, one needs to define an energy objective that takes these cases into account.

The OMPL library contains several pre-defined optimization objectives. The following graph describes the inheritance diagram of the class **ompl::base::OptimizationObjective** which is an abstract class containing several virtual functions.

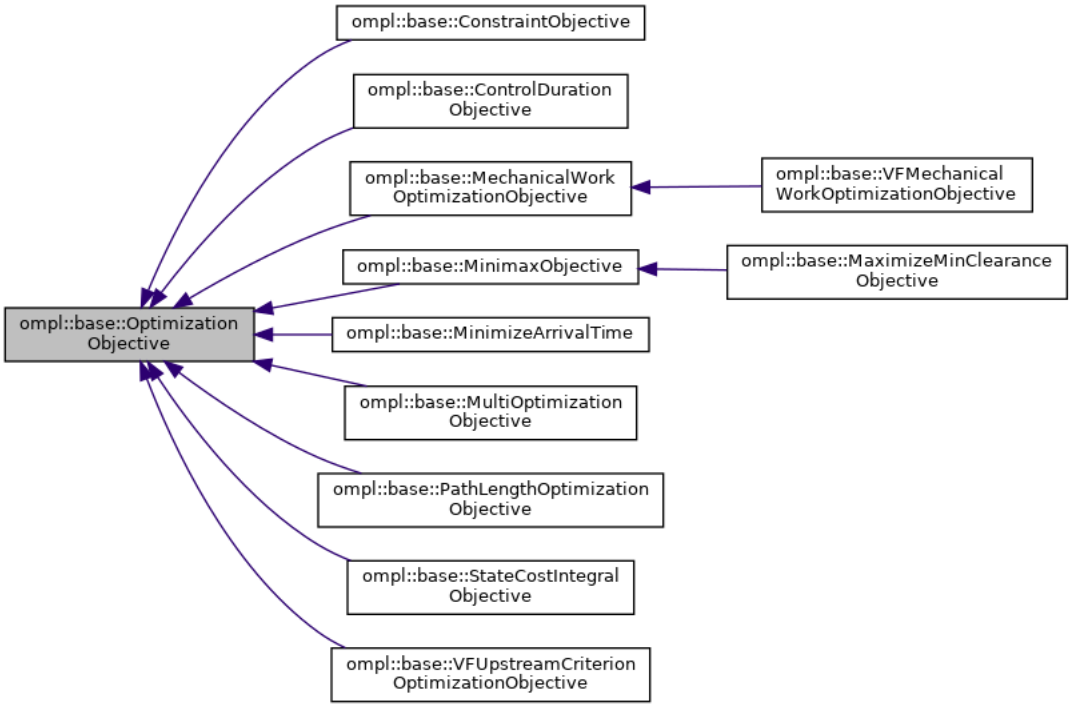


Figure 10: Inheritance Diagram for ompl::base::OptimizationObjective. Source: ompl::base::OptimizationObjective Class Reference [25]

There are many different derived classes that inherit from the base abstract class, each one focusing on optimizing different objectives. Two of these derived classes were tested as candidates for energy minimization, namely, **MechanicalWorkOptimizationObjective** and **StateConstIntegralObjective**.

- **MechanicalWorkOptimizationObjective**: This optimization objective defines path cost using the idea of mechanical work. This objective selects the path which requires the minimum amount of mechanical work to traverse. The scheme for calculation of the cost of motion used by this objective function is defined as the difference between the state costs of two consecutive states. The **stateCost()** method is overridden to return values of 0.8 and 2 for ground and aerial points respectively. It does not, however, take negative costs into account. The scheme is shown in the code below.

```
1         ob::Cost motionCost(const ob::State *s1, const ob::State *s2) const override
2         {
3                 // Only accrue positive changes in cost
4                 double positiveCostAccrued = std::max(stateCost(s2).value() - stateCost(
                      s1).value(), 0.0);
5                 std_msgs::Float32 cost;
6                 cost.data = positiveCostAccrued;
7                 return ob::Cost(cost.data);
8         }
```

- **StateConstIntegralObjective**: This optimization objective is defined where path cost can be represented as a path integral over a cost function defined over the state space. The path integral is approximated using the trapezoidal rule which is given as follows. The **stateCost()** method is overridden in the same manner as before.

$$motionCost = \frac{stateCost(s_1) + stateCost(s_2)}{2} * dist(s_1, s_2) \tag{1}$$

Additionally, a custom energy objective was also developed and tested. This objective consists of a class that inherits from the *OptimizationObjective* base class. The virtual function *motionConst()* is overridden and defines the cost of moving between two given states. The cost is defined as the path length between the two states times a factor which is decided based on the z-coordinate of the two states. If either of the states are aerial points, the vehicle has to fly and the aerial factor must be used to compute the cost, otherwise, the ground factor will be used.
Finally, the function adds a *CostToGoal* term which essentially controls the level of freedom the algorithm has to select a point far away from the goal. A higher setting of the *DistanceToGoalFactor* would result in the vehicle "greedily" moving towards the goal. This concept is based on the greedy algorithm. The class definition is shown below.
Some results for these tests are discussed in Section 6.1.

```
1   class EnergyObjective : public ob::OptimizationObjective
2   {
3   public:
4       EnergyObjective(const ob::SpaceInformationPtr& si, const ob::State* goalState) : ob::
            OptimizationObjective(si), goalState_(goalState)
5       {
6           AerialFactor = 2.0;
7           GroundFactor = 0.8;
8           DistanceToGoalFactor = 5;
9       }
10
11      ob::Cost stateCost(const ob::State* s) const override {}
12
13      ob::Cost motionCost(const ob::State *s1, const ob::State *s2) const override
14      {
15          // Compute the cost of the motion based on the mode of transport
16          // and the energy consumption associated with that mode
17
18          // Cast the abstract state type to the type we expect
19          const ob::SE3StateSpace::StateType *state1 = s1->as<ob::SE3StateSpace::StateType
                >();
20
21          // Cast the abstract state type to the type we expect
22          const ob::SE3StateSpace::StateType *state2 = s2->as<ob::SE3StateSpace::StateType
                >();
23          double factor = 0.0;
24          if(std::round(state1->getZ() * 10) / 10.0 > 0.1 || std::round(state1->getZ() * 10)
                 / 10.0 > 0.1)
25          {
26              factor = AerialFactor;
27          }
28          else
29          {
30              factor = GroundFactor;
31          }
32
33          double costToGoal = si_->distance(s2, goalState_) * DistanceToGoalFactor;
34          double costForMotion = si_->distance(s1, s2) * factor;
35          return ob::Cost(costForMotion + costToGoal);
36      }
37  private:
38      double AerialFactor;
39      double GroundFactor;
40      double DistanceToGoalFactor;
41      const ob::State* goalState_;
42  };
```

An important observation during the development of this code was that since the *OptimizationObjective* class is an abstract class containing virtual functions, it could not be instantiated. Moreover, since the class *EnergyObjective* inherits from this class, it also becomes an abstract class. Therefore, in order to be able to create an object of this class, the line 11 of the code was added which simply overrides the virtual function *stateCost* to a

function without a body. Since both virtual functions are overridden, the *EnergyObjective* class was no longer an abstract class and, hence, was able to be instantiated.

5.5 A* Path Planning Algorithm Development

The Path Planning Module was further investigated to find other alternatives to the RRT* path planning algorithms. The A* path planner was chosen due to its property of being optimally efficient and providing exact solutions. The A* algorithm is a graph based algorithm that finds the optimal path numerically. It works by discretizing the environment into nodes and considering the path only in the nodes. Unfortunately, one of the main drawbacks of this algorithm is that it tends to get quite computationally expensive as the size of the dimensions increases. But since our requirement was an optimal algorithm that works in 3 dimensions, the A* algorithm warranted a closer look. The Multi-Robot-Systems group at the Czech Technical University in Prague has done extensive work in developing an A* planner in 3D [5]. Their algorithms have been used as the framework for this hybrid drone.

In order to connect the A* planner developed by the MRS group with the hybrid drone, a new node was coded named A*_caller. The body of this code is shown below.

```
1   class MyPlanner {
2   public:
3       MyPlanner() : nh_("~"), astar_planner_(safe_obstacle_distance,
            euclidean_distance_cutoff, submap_distance, planning_tree_resolution,
            distance_penalty,
4       greedy_penalty, timeout_threshold, max_waypoint_distance, min_altitude, max_altitude,
            unknown_is_occupied, bv) {
5
6           octomap_sub_ = nh_.subscribe<octomap_msgs::Octomap>("/arav/octomap_binary", 1, &
                MyPlanner::octomapCallback, this);
7       }
8
9     void octomapCallback(const octomap_msgs::Octomap::ConstPtr& msg)
10    {
11      // Convert octomap message to octomap::OcTree
12
13      // The octomap message (*msg) is converted to an Octomap data structure which is a
            shared pointer of type AbstractOcTree.
14      // It is then dynamically cast into a shared pointer of another type, namely, OcTree.
15
16      std::shared_ptr<octomap::AbstractOcTree> octree_ptr(octomap_msgs::msgToMap(*msg));
17      std::shared_ptr<octomap::OcTree> octree = std::dynamic_pointer_cast<octomap::OcTree>(
            octree_ptr);
18
19      // Plan a path using A* planner
20      const octomap::point3d start(0, 0, 0.1);
21      const octomap::point3d goal(10, 0, 0.1);
22      const double timeout = 5;
23      std::pair<std::vector<octomap::point3d>, bool> waypoints = astar_planner_.findPath(
            start, goal, octree, timeout);
24    }
25
26  private:
27      ros::NodeHandle nh_;
28      ros::Subscriber octomap_sub_;
29      double safe_obstacle_distance = 0.0001; // Obstacles are inflated by this amount
30      double euclidean_distance_cutoff = 0.1;
31      double submap_distance = 10;
32      double planning_tree_resolution = 1;
33      double distance_penalty = 1;
34      double greedy_penalty = 1;
35      double timeout_threshold = 1;
36      double max_waypoint_distance = 5;
37      double min_altitude = 1;
38      double max_altitude = 5;
39      bool unknown_is_occupied = true;
40      std::shared_ptr<mrs_lib::BatchVisualizer> bv;
41      pathfinder::AstarPlanner astar_planner_;
42  };
43
44  int main(int argc, char** argv)
45  {
46      ros::init(argc, argv, "path_planning");
47      MyPlanner planner;
48      ros::spin();
```

```
49          return 0;
50      }
```

This code defines a class called *MyPlanner* whose constructor initializes the *astar_planner_* class with a number of parameters along with a ROS node handler *nh_*. This nodes subscribes to the rostopic "*/arav/octomap_binary*" which contains the octomap message. The callback function of this subscriber defines start and goal coordinates and calls the *findPath* method with the required arguments.
A number of other parameters had to be included as private members of this class since the *astar_planner_* class contains numerous other methods that make use of them. These methods include operations such as adjusting the resolution of the octomap to a desired resolution, creating a visualisation system after the path is planned, checkers to verify whether the start and goal coordinates exist in the octomap etc. These functions are not required for this project and, thus, the parameters were all set to default values and passed to the class.

5.6 Ground Control Development

The initial configuration of the Path Control Module only catered to paths computed on the ground. The Path Selector Module was responsible for activating the ground path control is this path was deemed more optimal than the aerial one. It did this by publishing an activator boolean that the path controller subscribed to. The entire script execution of the controller was based on this boolean message. Since the Path Selector Module was unnecessary now, the activator boolean topic subscription was removed along with the associated callback function.

The path control module works by subscribing to the waypoint topic in which the final waypoints are published by the path planner. The controller is a simple proportional controller on the yaw angle orientation of the vehicle when it is on the ground. The yaw angle is controlled to make sure the vehicle is moving towards the target waypoint. This controller was further developed to a PID controller on the yaw rate of the vehicle. Both the initial and final controllers are shown below.

```
1  def proportionalController (error, debug):
2
3      # Internal variables #
4
5      global gain
6      global limit
7      out = 0.0
8
9      # Internal Logics #
10
11     out = gain * error
12
13     if out > limit:
14         out = limit
15
16     if out < -limit:
17         out = -limit
18
19     # Test #
20     if (debug == True):
21         print ("Yaw rate >> ", out, "rad/s")
22
23     return out
```

```
1  def PIDController_YawRate (error, debug):
2
3      global start_time
4
5      # Internal variables #
6
7      global Kp, Ki, Kd, limit, integral_yr,
           prev_error_yr, dt
8      out = 0.0
9
10     # Internal Logics #
11
12     integral_yr += error*dt
13     derivative = (error - prev_error_yr)/dt
14     out = Kp*error + Ki*integral_yr + Kd*
           derivative
15     prev_error_yr = error
16
17     if out > limit:
18         out = limit
19
20     if out < -limit:
21         out = -limit
22
23     # Test #
24     if (debug == True):
25         print ("Yaw rate >> ", out, "rad/s")
26
27     return out
```

In addition to the yaw rate, another PID controller was also used to control the linear velocity of the vehicle. The function body is exactly the same but the difference is when it is called, the error term is the difference between the real and desired position of the vehicle.

5.7 Aerial Control Development

As stated before in Section 5.2, the Pixhawk based aerial PID controller had not been implemented yet in the control architecture. Therefore, this controller was also connected to the algorithms. Several changes were made in the path controller to accommodate two controllers at the same time.

Several functions were generalized to handle three coordinates instead of two.

- The **waypointCallback** function appends and stores coordinates of waypoints received from the path planner. This function was modified to append all (X,Y,Z) coordinates.
- The **poseCallback** function collects data about the position of the vehicle in the environment and stores it in a variable. This function was also modified to collect all (X,Y,Z) coordinates from the "*/arav/gazebo/model_states*" rostopic.
- The **selectNextWP** function takes in the waypoint list appended and stored as well as the real position data as arguments and computed the next waypoint to go to and returns the Δ values. This function was modified to calculate the Δz value as well and return it.

Furthermore, a class **Mode()** was introduced with a method that checks the mode of travel to be used by the vehicle based on the z coordinates. The definition of this class is shown below. The method **checkMode()** assigns the strings "AERIAL" or "GROUND" to the private member "mode" based on the value of the z coordinates.

```
class Mode:
    def __init__ (self):
        self.mode = None

    # A type of flag to determine which controller to use

    def checkMode(self, zReal, zWP):
        if (zReal > 0.1 or zWP > 0.1):
            self.mode = "AERIAL"
        else:
            self.mode = "GROUND"

    return self.mode
```

The final step was to bring the Pixhawk PID controller to this interface. In the previous semester, a simple script, named offb_node.cpp to enable OFFBOARD mode in a drone was used to conduct flight tests as well as tune the PID controller using QGroundControl. The script asks for a specific waypoint from the user after which it requests arming which allows the vehicle to fly and mode change to OFFBOARD mode, which means that the drone will be controlled using an external API(ROS, in this case).
For our purposes, the code was modified so that instead of user provided waypoints, the data is received via a rostopic named "*/arav/chatter*" that the offboard node subscribes to. Based on these waypoints, the node publishes the commanded local position. The following code snippet shows the subscription to the "*/arav/chatter*" rostopic as well as its callBack function.

```
ros::Subscriber pos_x_sub = nh.subscribe<geometry_msgs::PoseStamped>("/arav/chatter", 10,
    chatter_cb);

geometry_msgs::PoseStamped pose1;
void chatter_cb(const geometry_msgs::PoseStamped::ConstPtr& key)
{
    pose1 = *key;
}
```

In order to execute this modified script, named "*offb_node_user_input*", once the vehicle has to enter the "AERIAL" mode, a function, named, "*air_module_offb_node()*" was defined to launch the script and enable the "OFFBOARD" mode in Pixhawk. Once the script has started, it listens to the */arav/chatter* rostopic. The following code shows the function definition that launches the "*offb_node_user_input*" executable and creates a publisher that publishes the waypoints on the "*/arav/chatter*" rostopic. The function call as well as the message publish take place once the **checkMode()** method returns the "AERIAL" mode. Section 6.3 discusses the results of the execution of this algorithm.

```
aerialcommandTopic = "/arav/chatter"

def air_module_offb_node():
    global offb_node
    package = 'beginner_tutorials'
    executable = 'offb_node_user_input'
    node = roslaunch.core.Node(package, executable)

    launch = roslaunch.scriptapi.ROSLaunch()
    launch.start()

    offb_node = launch.launch(node)

aercmdPub = rospy.Publisher(aerialcommandTopic,PoseStamped,queue_size=10)
```

5.8 "Luffy" Robot Description

The first robot model, **arav**, developed by Albert Ceballos [15] consisted of a ground part which was in the form of a tank and an aerial part which was a quadcopter. Apart from this, the vehicle was also equipped with the different sensors for environment perception as well as a ground controller plugin defined in its urdf model. Since the aerial controller was not developed by the seniors, the quadcopter defined consisted of immovable propellers and arms. The Pixhawk PID controller definition was not present in arav.
The second robot model, **dd_bot**, developed by Karthik Mallabadi [1] consisted of the ground part which was a box attached with four wheels along with the **iris** model available in the PX4-Autopilot Software Repository [27]. The iris model is PX4-enabled which is why the flight tests were able to be simulated using dd_bot.

Both models included definitions of only some of the parts of the robot model. Thus, a new model definition was created with the complete functionality coded in the robot, named **luffy**.
This new model was defined by taking the ground module of the dd_bot which is a much simpler and easy-to-use model than the arav counterpart. The two camera sensor plugins and three ultrasonic sensor plugins were taken from arav and connected to the base link of luffy. The ground controller plugin which was used in arav was connected as well to the wheels defined in the luffy model. Finally, the iris model was connected to the ground module of luffy in the same way as dd_bot. The Flowchart 11 explains visually which components were connected to define the luffy model and Figure 15 shows the different model versions developed this year and last year.

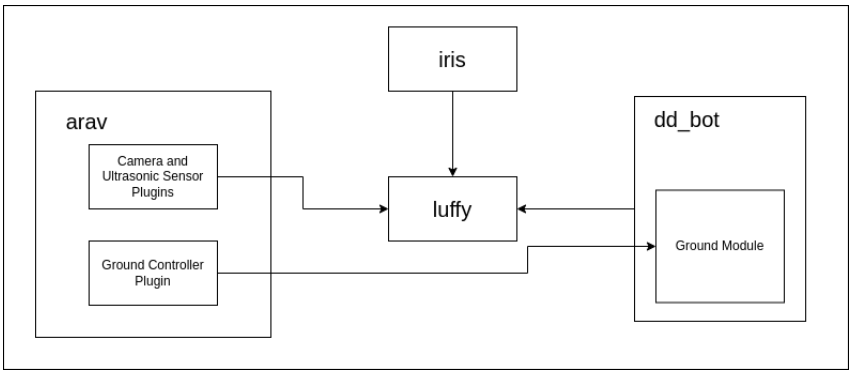


Figure 11: Flowchart of Luffy Model Definition. Source: Prepared by the author

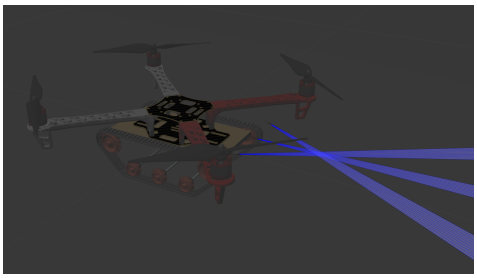


Figure 12: ARAV

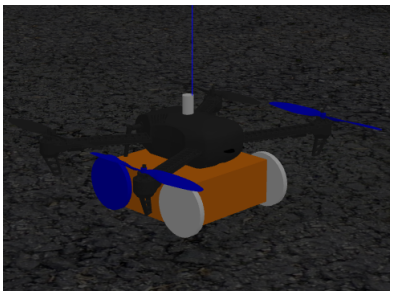


Figure 13: dd_bot

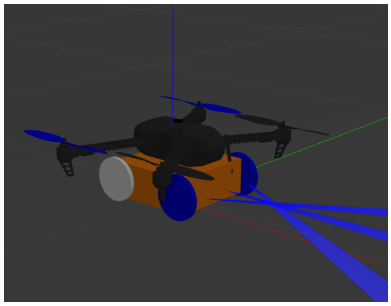


Figure 14: Luffy

Figure 15: Earlier and latest robot models Source: Prepared by the author

6 Results and Analysis

The results obtained after the work done in the second semester are split into sections covering elements of the path planning optimization and development of the path control module. In order to capture all the data that was of interest for analysis, **rosbag** [28] was used which is a useful command line tool that creates bag files for recording and playing back ROS topics of interest. Additionally, all the graphs for the simulation tests were obtained by feeding these bag files into **plotjuggler** [29] which is a ROS application for graphing data carried by ROS topics.
Another important tool used to run simulations smoothly was **CUDA** [30]. CUDA or Compute Unified Device Architecture, is a parallel computing platform that lets users utilize their Graphical Processing Units (GPUs) for general purpose processing. This application was particularly useful for running the object detection and image processing algorithms.

6.1 Optimization Objective Tests

As mentioned in Section 6.1.3, several objective functions were tested to determine the one which represents energy minimization most realistically. These functions employ different schemes of calculation of the cost of a path to represent the energy required for movement. The RRT* planning algorithm was run 3 times for 3 different objective functions and the trajectory of the planned path was visualized. The test case chosen for this comparison was a wall standing 5m away from where the vehicle starts its mission. The start and goal points were (0,0,0.1) and (10,0,0.1) respectively. Figure 16 shows the gazebo environment where the obstacle and model spawn.

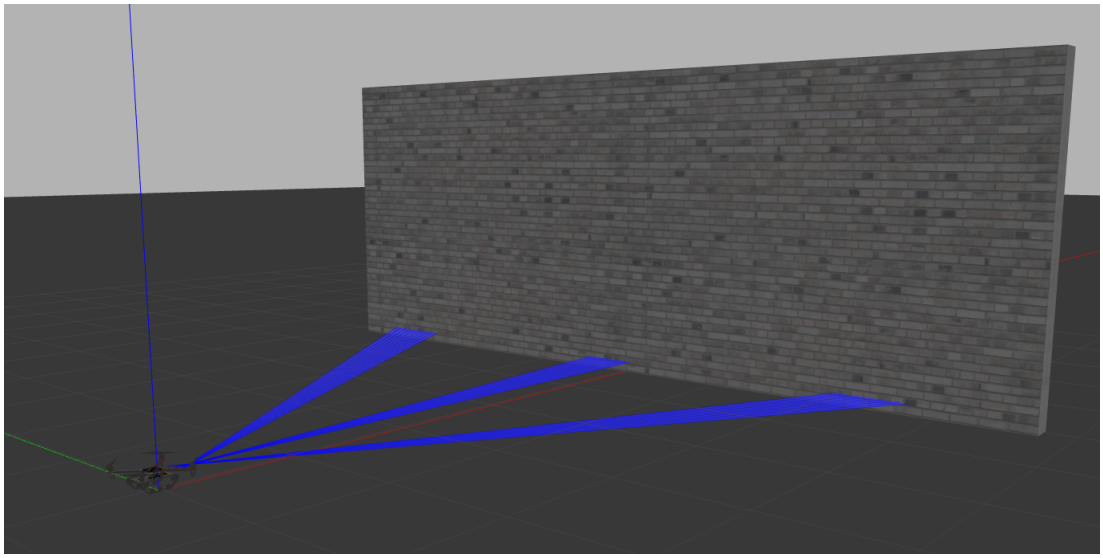


Figure 16: Gazebo Representation of the wall obstacle. Source: Prepared by the author

6.1.1 Mechanical Work Objective

The results obtained by using this objective are not favourable at all. Not only does the path not stay on the ground, which should be the most optimal path in this case, the aerial path is chaotic. Figures 21 show the visualization of the trajectory calculated by the planner along with the X,Y,Z coordinates graphed.

6.1.2 State Cost Integral Objective

This objective produces results that are closest to the most optimal one with the path computed successfully goes around the wall. However, if we look at the z coordinate of the planned trajectory, it rises to more than 1.2m above ground which means the vehicle would be required to fly in this region. This part of the trajectory increases the cost of the path by a significant amount. Figures 26 show the trajectory and the graphed coordinates where the Z coordinate deviation is observed.

6.1.3 Custom Energy Objective

The third cost objective manages to create a path that is not chaotic as with the case of the trajectory with the Mechanical Work Objective. However, it again fails to capture the ground optimal path and instead plans

a fully aerial path which is extremely energy expensive. The results of this test are shown in Figure 31.

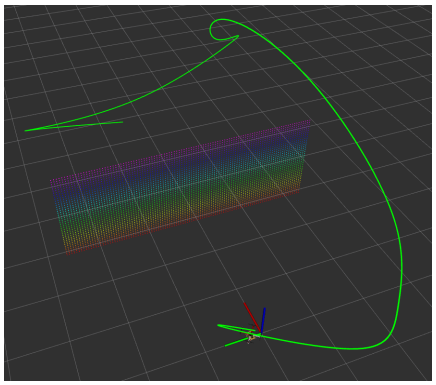


Figure 17: MW Objective Visualization in Rviz

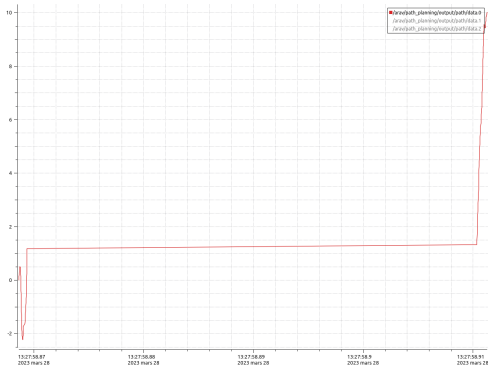


Figure 18: X position vs Time : MW Objective

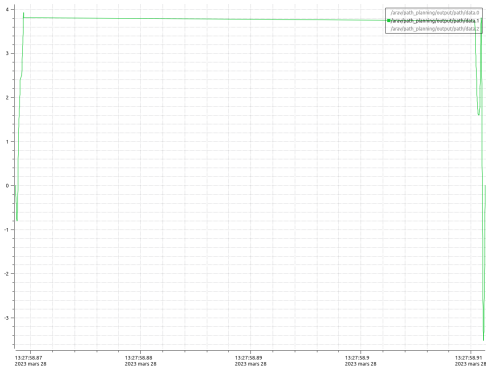


Figure 19: Y position vs Time : MW Objective

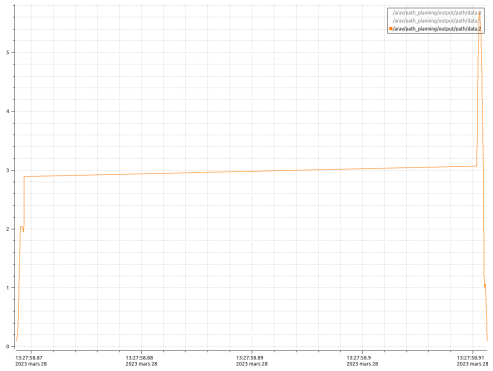


Figure 20: Z position vs Time : MW Objective

Figure 21: Trajectory Results with Mechanical Work Objective Source: Prepared by the author

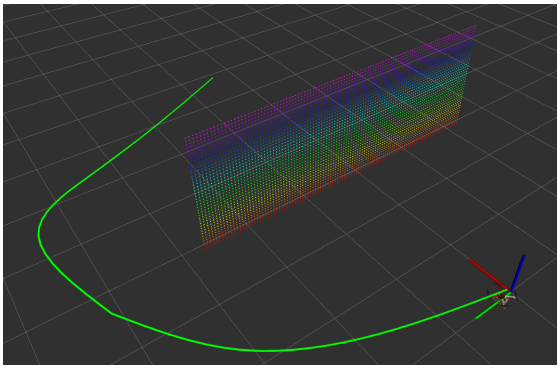


Figure 22: SCI Objective Visualization in Rviz

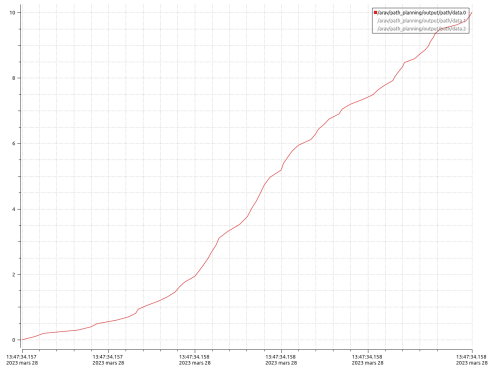


Figure 23: X position vs Time : SCI Objective

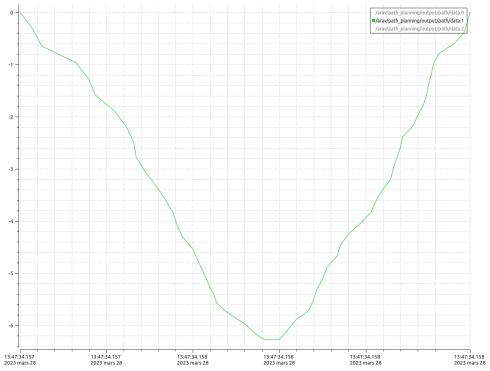


Figure 24: Y position vs Time : SCI Objective

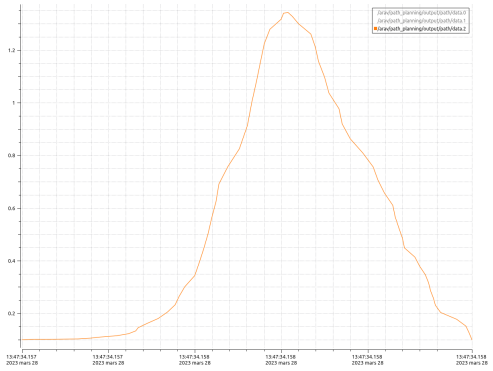


Figure 25: Z position vs Time : SCI Objective

Figure 26: Trajectory Results with State Cost Integral Objective Source: Prepared by the author

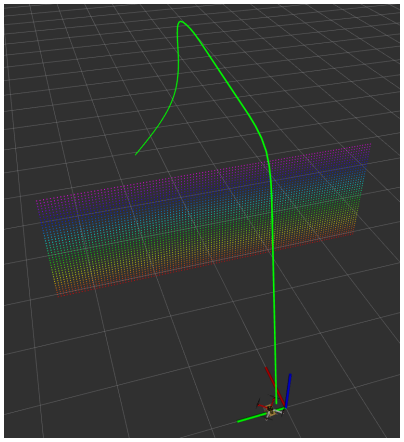


Figure 27: E Objective Visualization in Rviz

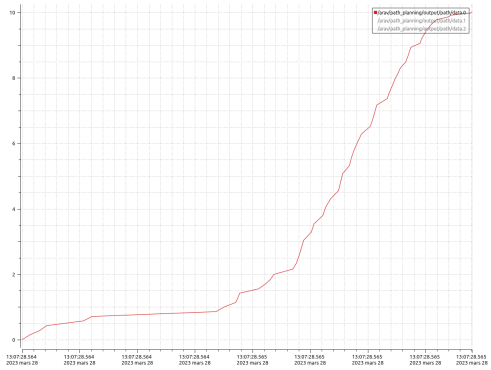


Figure 28: X position vs Time : Custom Energy Objective

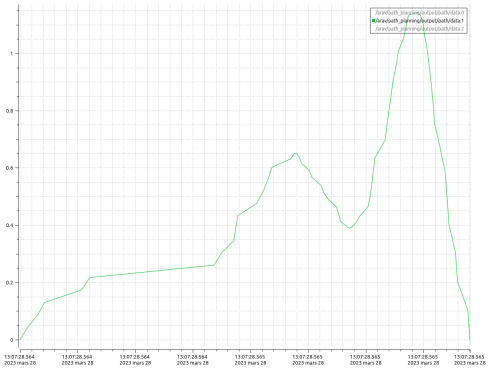


Figure 29: Y position vs Time : Custom Energy Objective

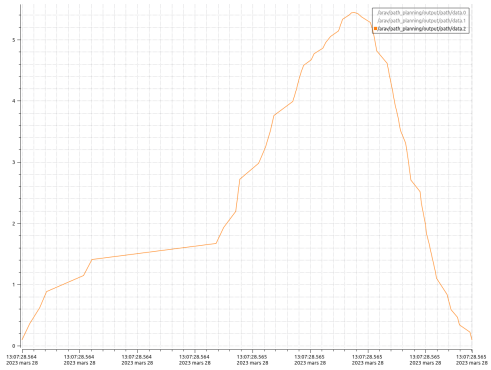


Figure 30: Z position vs Time : Custom Energy Objective

Figure 31: Trajectory Results with Custom Energy Objective Source: Prepared by the author

After conducting several tests with these 3 objective functions, there are some observations seen. Even though only one test result per function is shown here, every test done prior to these yielded different results for each function. The final trajectory was not constant, but rather kept changing randomly. Additionally, the path was wholly or atleast partly aerial in almost all the cases even though the optimal path was clearly a fully ground one.
An explanation for this is that the path-planning algorithm RRT* is a sampling-based algorithm which means that it samples nodes from the state space randomly and connects them before starting its optimization logic to rewire paths with more optimal nodes. Also, before starting the path planning, the planning problem has to be provided with certain boundaries in space in which these nodes would be generated randomly. Start and goal positions can only be set within these boundaries as well. The boundaries for the Z coordinate of the state space was set as (0.1,10). Therefore, points can be generated at atleast the height of the vehicle (0.1m) upto a height of 10m. The planner considers all points above a height of 0.2m as aerial points. Considering these boundaries, one can observe that there exists a higher probability for the planner to generate aerial points than ground points. Since the optimization logic only compares points that have been generated randomly, the objectives end up comparing only aerial points, resulting in an aerial path.

For a hybrid drone, these results prove that a sampling-based algorithm may not be the most approriate choice for path planning. Therefore, alternative algorithms such as A* which is a graph-based algorithm, were investigated and developed.

6.2 A* Path Planning Test Errors

As mentioned in Section 5.5, The work done by the MRS group at the Czech Technical University in Prague [5] was coupled with the software architecture of this vehicle. However, due to certain unresolved errors, the algorithm failed to obtain results. This section discusses the errors that persisted from the execution of the A* algorithm.
As previously discussed, an executable file called Astar_caller.cpp was developed that calls the method to compute the optimal path with the A* algorithm. The arguments for this method were the start and goal positions, the octomap Octree data structure containing information about the obstacles and a timeout param-

eter. The octomap passed to this findPath() method is generated using the octomap_server package from the octomap_mapping stack [23]. When this node was run, the error message shown in Figure 32 was displayed on the shell repeatedly.

```
Size of the Octomap: (0 0 0)
[ INFO] [1679943448.253754460, 5.664000000]: Goal Coordinates not found in the map
[ INFO] [1679943448.253792285, 5.664000000]: [Astar]: Start : 0, 0, 0.1
[ INFO] [1679943448.253818053, 5.664000000]: [Astar]: Goal: 10, 0, 0.1
[ WARN] [1679943448.253840479, 5.664000000]: [Astar]: PATH DOES NOT EXIST!
Size of the Octomap: (0 0 0)
[ INFO] [1679943454.913685917, 12.299000000]: Goal Coordinates not found in the map
[ INFO] [1679943454.913739052, 12.300000000]: [Astar]: Start : 0, 0, 0.1
[ INFO] [1679943454.913762812, 12.300000000]: [Astar]: Goal: 10, 0, 0.1
[ WARN] [1679943454.913804630, 12.300000000]: [Astar]: PATH DOES NOT EXIST!
Size of the Octomap: (0 0 0)
[ INFO] [1679943461.337648145, 18.693000000]: Goal Coordinates not found in the map
[ INFO] [1679943461.337725587, 18.693000000]: [Astar]: Start : 0, 0, 0.1
[ INFO] [1679943461.337743830, 18.693000000]: [Astar]: Goal: 10, 0, 0.1
[ WARN] [1679943461.337783601, 18.693000000]: [Astar]: PATH DOES NOT EXIST!
```

Figure 32: A* algorithm error message. Source: Prepared by the author

Upon further investigation, it was found that before implementing the algorithm, the method checks to see whether the start and goal coordinates are inside the octomap or not. This check returned false and no solution for the path was found. To check the size of the octomap, the function **BBXBounds()** from the OcTree class was called [31]. This function revealed that size of the octomap being provided was 0. This information seemed to contradict the fact that data was indeed flowing in the topic "*/arav/octomap_binary*" which is the topic on which the octomap_server package publishes the octomap. This error did not occur with the case of the RRT* algorithm implementation where the same octomap was handled using the Flexible Collision Library. As of now, this error is still unresolved.

6.3 Path Control Tests

The path control module involved the development of both the ground as well as the aerial controllers. The development of the "luffy" robot model which was PixHawk enabled allowed the vehicle to implement aerial paths. In order to test the performance of this new vehicle in the software architecture, a special case of the RRT* path planner was computed in which boundaries of Y coordinates were restricted to (-0.5,0.5) which meant that the vehicle could not take any large turns. Since the wall was positioned right in between the vehicle and the goal, the resulting path had to be completely aerial. The luffy model was tested with the modified path control algorithm. Figure 36 shows the trajectory results of this implementation of the RRT* algorithm.

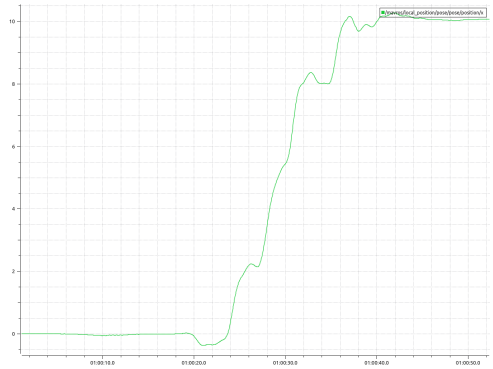


Figure 33: X position vs Time : Aerial Path

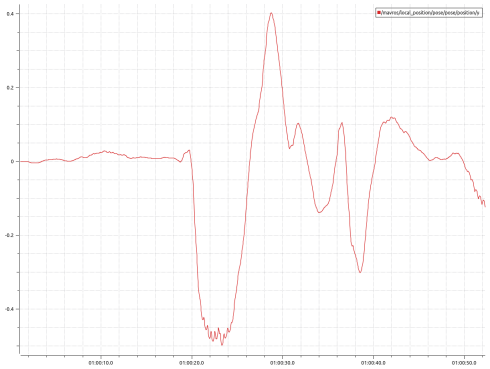


Figure 34: Y position vs Time : Aerial Path

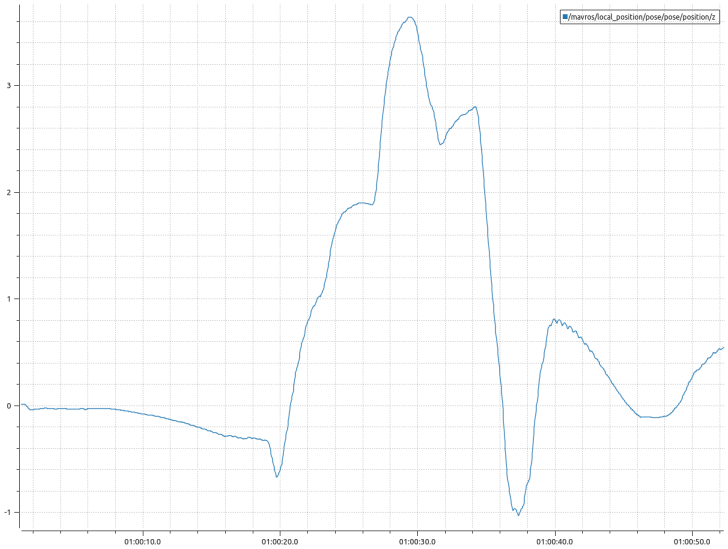


Figure 35: Z position vs Time : Aerial Path

Figure 36: Trajectory Results with the Aerial Controller Source: Prepared by the author

We observe that aside from some noise seen in the values, the controller does a good job of controlling the vehicle trajectory. Even though the Y coordinates were restricted, there was still some chatter seen which was unavoidable. Another important observation is that the movement of the vehicle does not seem to be fluid enough in the air. This was seen in the actual simulation as well where the controller was moving the vehicle towards a waypoint and would stop at the point before starting the next waypoint. This behaviour is seen in Figure 35 where there are sudden changes in the progression such as around Z=2,4 and 3.

7 Conclusion and Future Work

This project aims to bring together the robustness of ground rovers on rugged terrain with the agility of aerial drones to create a hybrid drone with the advantages of both these domains of technology. It is a 5 year long project with significant contributions by each team which has resulted in significant growth and maturity. The contribution made to it this year has focused on combining two different methodologies of constructing a software architecture to create an elegant solution for the functionality of this vehicle. More specifically, the path planning and path control modules of this vehicle were investigated and developed further.

In the first phase of the development, work was focused on the path planning module where certain gaps were identified with potential for growth. This investigation pointed out some cases where the vehicle would not be able to perform its intended function of conserving energy. These cases included complex environments where the vehicle was required to cover part of its mission using the ground mode and the rest using the aerial mode. Therefore, the path planning algorithm was amended by defining specific energy minimization objectives and providing them to the planner. Different candidates for optimization objectives were tested along with a custom designed one. These tests of the objective functions pointed out a shortcoming in the RRT* path planning algorithm for use with a hybrid vehicle such as this where it probabilistically favours aerial paths that are more energy-consuming. Therefore, an alternative A* algorithm was coupled and tested with the software architecture. Unfortunately, certain errors have yet to be debugged in order to analyze the results of this algorithm.

The second phase of the development was on the path control module which lacked an aerial controller as well as a mature ground controller. The ground controller was initially a proportional controller controlling the yaw angle of the vehicle in the ground mode which allowed it to move towards to the intended waypoint. However, this controller would not function properly in case of sharp turns in complex environments. The vehicle was also only operating at a constant linear velocity. Therefore, a PID controller was developed to control yaw as well as the position of the vehicle that yielded positive results.
In order to couple an aerial controller to the vehicle, the model was redefined by combining parts of code from the earlier versions of the model, namely arav and dd_bot to create the luffy vehicle. Luffy was fitted with the same sensor plugins and ground controller plugin as arav while the general ground module and the Pixhawk enabled iris quadcopter used was provided by dd_bot. The resulting luffy model also produced favourable results.

This project has been an extremely useful exercise to gain knowledge and experience in complex robotics, path planning as well as different platforms such as ROS and Gazebo. The results of the different tests carried out after developing the modules, especially the path control module, have been promising. The following points highlight areas of development and issues that persist where further progress can be made in the future.

- Since the results with the path planning algorithms did not produce positive results. The investigation needs to be continued to develop alternative algorithms. In this context, the error persisting in the development of the A* algorithm needs to be analyzed and debugged.
- The general computation time of the algorithm computation is still too long and requires significant processing power even after the use of CUDA. Therefore, this latency needs to be addressed.
- The concept of the flight domain developed in the previous semester needs to be added to the path control architecture to take into account the constraint of time.
- This software architecture does not utilize SLAM algorithms which constraints the vehicle to only gain information about its environment within the range of the sensors. Therefore, this approach would increase the utility of the vehicle.
- The vehicle is not suited yet for dynamic environments with moving obstacles. This is a major area of research and development.
- This software architecture will also have to be tested on the actual developed prototype to verify the credibility of the simulation results.

References

[1] K. Mallabadi and R. Dsouza. *Autonomous robotic aerial vehicle: S3 project report*. March, 2022.

[2] Po-Lung Yu. *Multiple-Criteria Decision Making: Concepts, Techniques, and Extensions*. eng. Vol. 30. Mathematical Concepts and Methods in Science and Engineering. Boston, MA: Springer, 1985. ISBN: 9781468483970.

[3] Kun Zhang et al. "A UAV Autonomous Maneuver Decision-Making Algorithm for Route Guidance". eng. In: *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2020, pp. 17–25. ISBN: 9781728142784.

[4] Tomas Baca et al. "The MRS UAV system: Pushing the frontiers of reproducible research, real-world deployment, and education with autonomous unmanned aerial vehicles". In: *Journal of Intelligent & Robotic Systems* 102.1 (2021), p. 26.

[5] Tomáš Báča. *MultiRotor UAV System*. 2021. URL: https://github.com/ctu-mrs/mrs_uav_system.

[6] Karen Northon. *Mars Helicopter to Fly on NASA's Next Red Planet Rover Mission*. Last Updated: Mar 16, 2020. 2018. URL: https://www.nasa.gov/press-release/mars-helicopter-to-fly-on-nasa-s-next-red-planet-rover-mission.

[7] *NASA's Perseverance Rover Begins Its First Science Campaign on Mars*. 2021. URL: https://www.nasa.gov/feature/jpl/nasa-s-perseverance-rover-begins-its-first-science-campaign-on-mars.

[8] Leah Crane. *First photo of Chinese Yutu-2 rover exploring far side of the moon*. 2019. URL: https://www.newscientist.com/article/2189606-first-photo-of-chinese-yutu-2-rover-exploring-far-side-of-the-moon/.

[9] Kyunam Kim et al. "A bipedal walking robot that can fly, slackline, and skateboard". In: *Science Robotics* 6.59 (2021), eabf8136. DOI: 10.1126/scirobotics.abf8136. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abf8136. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abf8136.

[10] *Documentation - ROS Wiki*. last edited 2022-05-20. URL: http://wiki.ros.org/.

[11] *Understanding ROS nodes*. URL: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes.

[12] S. CHAUDHARY and N. RAGHUNATHAN. *Autonomous robotic aerial vehicle: S3 project report*. March, 2019.

[13] A. Sambalov. *Autonomous robotic aerial vehicle: S3 project report*. March, 2020.

[14] R. Rodrigues and B. Ismael. *Autonomous robotic aerial vehicle: S3 project report*. March, 2021.

[15] J. Bessa and A. Ceballos. *Autonomous robotic aerial vehicle: S3 project report*. March, 2022.

[16] *PX4 Autopilot User Guide (master)*. Last Updated: 9/6/2021. URL: https://docs.px4.io/master/en/.

[17] *The Raspberry Pi Foundation*. URL: https://www.raspberrypi.org/.

[18] *OptiTrack - Motion Capture Systems*. URL: https://www.optitrack.com/.

[19] *Gazebo - Robot Simulation made easy*. URL: https://classic.gazebosim.org/.

[20] *QGroundControl User Guide*. URL: https://docs.qgroundcontrol.com/master/en/.

[21] *ROS (1) with MAVROS*. URL: https://docs.px4.io/v1.12/en/ros/ros1.html.

[22] *MAVLink Developer Guide*. URL: https://mavlink.io/en/.

[23] *Octomap_ Server ROS Package*. URL: https://github.com/OctoMap/octomap_mapping/tree/kinetic-devel/octomap_server.

[24] *OctoMap 3D Models with ROS/Gazebo*. URL: https://octomap.github.io/.

[25] *Open Motion Planning Library*. URL: https://ompl.kavrakilab.org/core/index.html.

[26] *Flexible Collision Library*. URL: https://flexible-collision-library.github.io/index.html.

[27] *Pixhawk Autopilot Software*. URL: https://github.com/PX4/PX4-Autopilot.

[28] *Rosbag Documentation*. URL: http://wiki.ros.org/rosbag.

[29] *Plotjuggler Documentation*. URL: http://wiki.ros.org/plotjuggler.

[30] *CUDA Toolkit*. URL: https://developer.nvidia.com/cuda-toolkit.

[31] *Octomap Namespace Reference*. URL: https://octomap.github.io/octomap/doc/namespaceoctomap.html.