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

Master of Aerospace Engineering Research Project

Autonomous Robotic Aerial Vehicle

S3 Project Report

Author: Franklin, Brian Ismael

Due Date of Report: 26/03/2021
Actual Submission Date: 26/03/2021

Starting Date of Project: 17/11/2019 *Duration:* 14 Months

Tutors: Deepthimahanthi RaghuVamsi, Mifdaoui Ahlem

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

The main goal of this project is to design, assemble, and program a hybrid vehicle which autonomously moves on both the ground and in the air. Additionally the vehicle should be able to avoid obstacles and follow a path calculated based on given heuristics such as power consumption or time efficiency.

An Initial prototype of the vehicle has been designed and assembled 2 years ago by previous MAE students: Sakshi Chaudhary and Nandhini Raghunathan. Then the following year MAE student Anton Sambalov equipped the aerial module with all the necessary components to be flight ready and he also developed an initial algorithm for the ground module.

During the past second and third semester research on various path following and obstacle avoidance algorithms for both the ground and aerial modules was conducted, chosen, and implemented in a simulation environment. A simulation environment was chosen in which to safely test the algorithms on the vehicle before running HITL tests.

In the following part of the project the obstacle avoidance algorithm will be tested with a 3D model of the physical vehicle in order to validate the algorithms. Finally the algorithms will be implemented on to the physical vehicle and tested in real time.

Contents

1 Introduction

For the past years there has been revolutionary changes in the robotics sector of engineering. More specifically there has been a major increase in two areas: Unmanned Aerial Vehicles (UAV's) and exploratory vehicles better known as rovers.

Unmanned Aerial Vehicles have risen in popularity for a various number of reasons. The main ones are the following: there is no onboard pilot, quite easy to use, comparatively inexpensive, and most of all versatility. Its versatility comes from the fact that the operator can easily optimize the UAV for a large range of missions from photography to reconnaissance to agricultural applications and many more.

On the other hand exploratory vehicles or rovers are commonly used in space and cross terrain missions. These vehicles are highly efficient and can easily traverse all types of terrains while enduring a plethora of conditions. Due to this there is a large demand in space exploration as well as in battlefield and disaster zone type conditions.

Given the high efficiency of both these vehicles to complete their respective missions it is ideal to make a hybrid of these vehicles. The objective and goal of this project is to do just that. We have created a vehicle, an Autonomous Robotic Aerial Vehicle, that combines both the flight capability of a UAV and the cross terrain versatility of a rover. This vehicle will be useful in both space and earth exploratory missions as well as any other mission it is optimized to do.

This vehicle is a combination of 2 interconnected modules of the 2 aforementioned vehicles: the air module (UAV) and the ground module (rover). The combination of both these vehicles will give our vehicle the ability to move freely in a 3D space; 2 dimensional due to the rovers motion and an additional 3^{rd} dimension due to the UAV's ability to fly. The report is structured as follows: 1: Introduction,

2 Project Definition

2.1 Goal of Project

The goal of the project is to give the vehicle the ability to bypass any obstacle in the shortest time possible based on the most efficient use of power, energy, or time based on the chosen module. In addition to that it should be able to perform said missions autonomously from its initial point to its final destination. This will be done by using a group of various sensors in order to avoid obstacles and move along the path accordingly.

2.2 Project Issues

The project began with 3 main issues of this project; communication between both the aerial and ground modules, the path planning and obstacle avoidance algorithms for each module, and finally creating a simulation environment in order to safely test these respective algorithms.
The first of the main issues is the communication between both modules. As mentioned previously the vehicle is made of 2 modules; the air module, a drone, and the ground module, a rover. In order for the vehicle to most efficiently move from one point to another there must be cohesion between both the air and ground modules. In order for that to take place an algorithm that allows for the simultaneous communication between both modules must be created. This will then allow the vehicle to switch between the ground and air modules seamlessly as needed.
The second main issue is the path planning and obstacle avoidance algorithms. Finding a good path planning algorithm is the first step so that the vehicle will be able to autonomously create a path from its initial waypoint to its final point. More so, improving the existing path planning algorithm will allow for a more efficient algorithm. However good the path planning algorithm is, it will not be complete until an obstacle avoidance algorithm is implemented alongside it. This is because the path planning algorithm will only give the vehicle the ability to create a path from point A to point B but does not give the vehicle the ability to avoid incoming, unperceived, or perceived obstacles. The obstacle avoidance algorithm will need to be implemented in order to give the vehicle the ability to autonomously take any path regardless of any obstructions in its way.
Finally, the last main issue is to create a simulation for the vehicle. This is important for safety reasons. It is necessary that we have an environment that provides a good description of the vehicle and allows us to test out various algorithms without damaging the vehicle itself. This will first be done with *MATLAB* and then finally

adapted to *Gazebo* for better control. For the air module we will use *Dronekit*, *QGroundControl*, and *Gazebo* as the simulation environments.

3 Investigation Method

3.1 Vehicle Components

The vehicle is comprised of 2 modules; the air module (a drone) and a ground module (a rover). Both modules have 2 different controllers; the ground module is controlled by a Raspberry Pi micro-controller board while the air module is controlled by the Pixhawk PX4 flight controller. Figure 1 shows the fully assembled vehicle containing both modules [1].

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. The air module of a vehicle consists of the components listed below:

- Frame: F450 Drone frame kit, which is a X-type frame with 450mm wheelbase is used for this project.
- 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. This means a continuous track is used in order to achieve maximum control on rough terrain and furthermore, prevents the vehicle from getting stuck. The ground module is made of the following:

- 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.

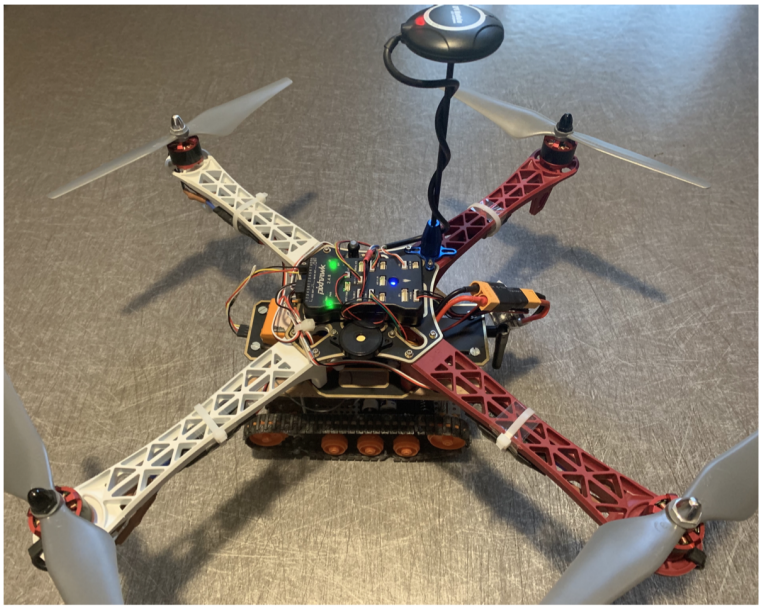


Figure 1: Fully Assembled Vehicle

3.1.3 Sensors

Moreover, the rover is equipped with a variety of sensors. In order to implement obstacle avoidance and path planning algorithms sensors are used to detect obstacles and distances to them along the vehicle's path.
Ultrasonic sensors measure distance to an object by emitting an ultrasonic wave and receiving back its reflection from the obstacle. Calculation of distance is performed by measuring the time from transmission to receipt of the wave. The *HC-SR*04 *Ultrasonic Sensor* is going to be installed on the front part of the vehicle.
These sensors are able to measure distance between 2cm and 400cm, while having a window of 30 degrees. Such parameters are suffcient to detect an obstacle at a safe distance and allow the vehicle to make a decision whether as to pass it on the ground or in the air[1]. 5 HC-SR04 sensors will be installed: 1 in the back, 1 on each side and 2 attached below front propellers, pointing to the front. Placement of the sensors can be seen in Figure 2. This design will allow the vehicle to avoid rotation about own axis when it looks for potential obstacles.

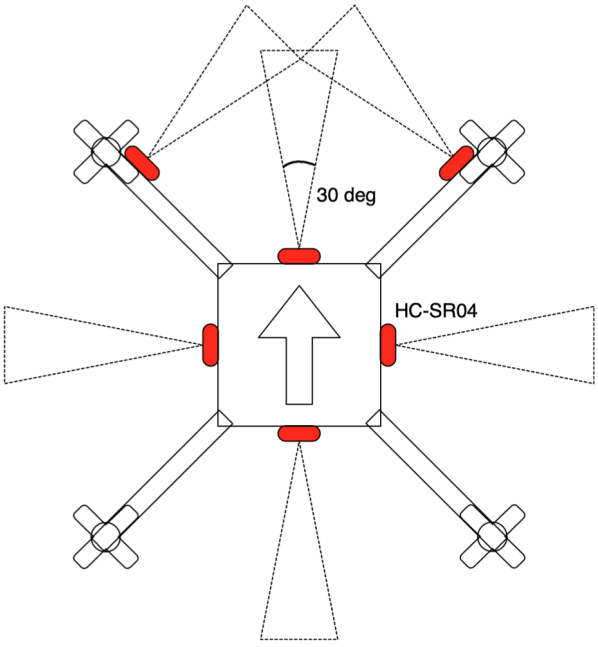


Figure 2: HC-SR04 Ultrasonic Sensor Placement

3.2 Technical Progress

With the first versions of both the Path Planning and Obstacle Avoidance algorithms set the vehicle needs to test its capabilities in more of a real world like environment. In achieve this a similar model to the *MATLAB* one is created using *Gazebo*. Gazebo is a well-designed simulator that makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Additionally, it has the ability of co-simulation with *MATLAB Simulink*. This allows us to specify the control laws of the vehicle in Matlab in order to closely imitate real life while creating an environment in Gazebo that has real world features. The Gazebo software requires a *URDF* file. A *URDF* file is an XML file format used in ROS to describe all elements of a robot [2]. The *URDF* file will be adapted in order to best simulate our vehicle in order to get the best results when simulating.

4 Semester 2 Work

4.1 Ground Module Simulation Environment

Currently the vehicle simulation environment is in *MATLAB* and it implements basic path planning and obstacle avoidance algorithms that work as follows as seen in Figure 3.

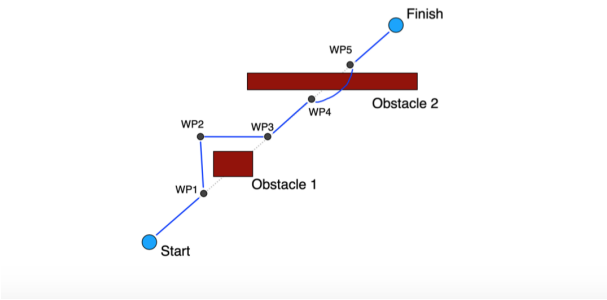


Figure 3: Ground Module Path Planning and Obstacle Avoidance Algorithms

From one waypoint to another, the vehicle must detect obstacles with the use of a sensor. As the obstacle is detected, either the ground or the aerial module can be used in order to avoid it. Avoidance on the ground is done by finding an alternative path around the obstacle and includes automatic creation of intermediate waypoints [1]. If the obstacle cannot be avoided by moving on the ground or if time is a priority then the aerial module will be used in order to avoid the obstacle. In this case the vehicle will use the ATOL (Automatic Take-off and Landing) algorithm. The ground vehicle's current algorithms have been implemented and simulated using *MATLAB*. The Robotics System Toolbox of *MATLAB* contains tools to test and analyze the behaviour of various mobile robots. Path Following for a Differential Drive Robot article [3] has been used as a structure of the rover simulation. To set up the appropriate test environment for the rover, a robot model with differential drive kinematics was created. This model approximates a vehicle with two independently driven wheels with settable radii that are separated by a specified track width as seen below in Figure 4.

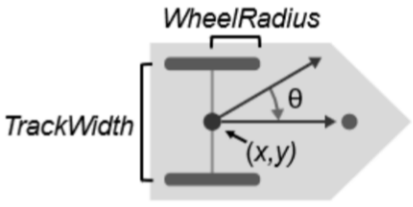


Figure 4: Differential-drive vehicle approximation using Matlab Robotics System Toolbox

The wheel radius of the rover is 4cm and the track width is 10cm. The state of the vehicle, which includes position and heading of the vehicle, is: [x, y, θ] [4].

4.2 Ground Module Path Following Simulation

The ground module implements tools from the Robotics Toolbox provided by *MATLAB*. The path following algorithm chosen to be implemented on the vehicle works and is visualized in the simulation as follows:

1. A list of waypoints for rover to follow is created.
2. Initial state is assigned to the vehicle as the first element of the list and $\theta = 0$.
3. A ControllerPurePursuit object is created to compute and update linear and angular velocities of the vehicle given the path to follow.
4. The desired linear velocity is set to be 0.3 m/s.
5. Look ahead distance is set to the value 0.4m
6. The simulation loop is initialised.
7. Path is plotted and held while the movement of the robot is plotted as a set of transformations.

The main parameters of the controller include the desired linear velocity and the look ahead distance [5]. The look ahead distance tells the vehicle how far away on the path it should be looking to compute the angular velocity. Look ahead distance should be neither too large or too small in order to avoid cutting corners while maintaining stable behavior as seen in Figure 5

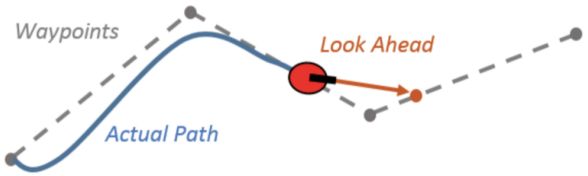


Figure 5: Look Ahead Distance

4.3 Ground Module Obstacle Avoidance

The obstacle avoidance simulation is also done using *MATLAB*. The vehicles current algorithm is the Probabilistic Roadmap (PRM) motion planning algorithm. The PRM algorithm determines a collision free path for the vehicle between the start and end waypoint. The algorithm is shown below in Figure 6 and explained just after.

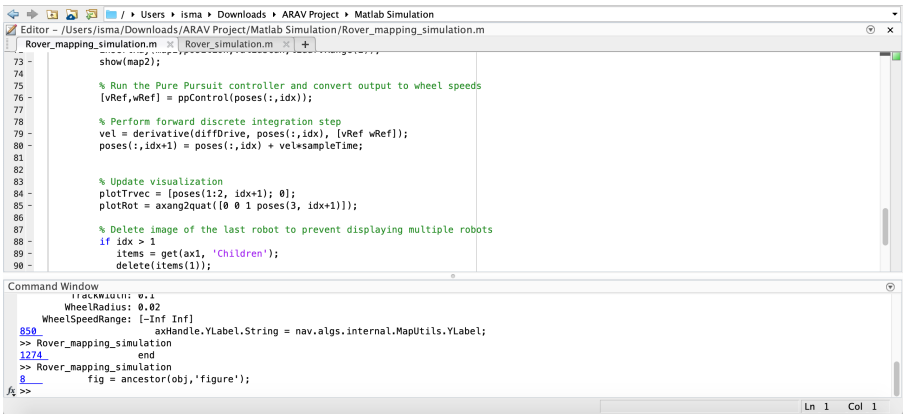


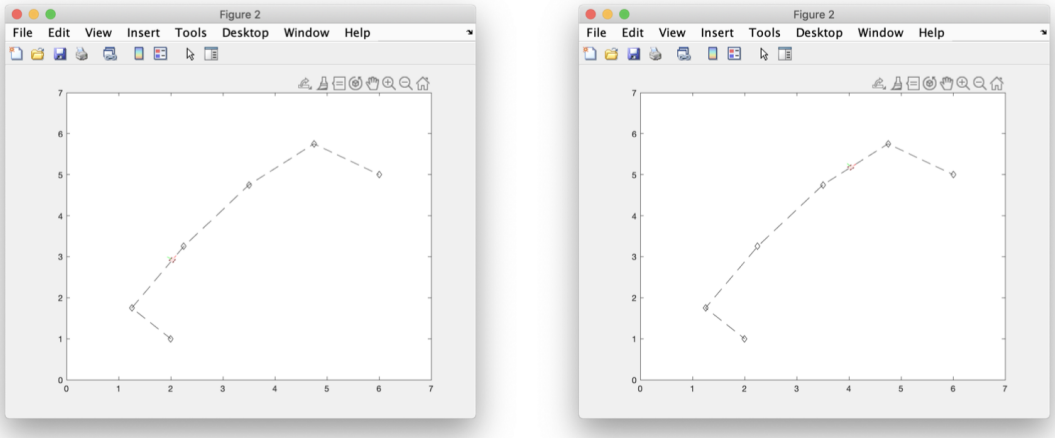
Figure 6: PRM Algorithm Free Path Computation

The algorithm does this by creating a path by connecting randomly sampled nodes in the free space of the environment [6]. The outcome of the PRM Algorithm computation is a probabilistic map that shows the resultant path to take based on numerous algorithm iterations. When implemented in the *MATLAB* simulation environment it creates a 2-D visual output of the walls (black boarders), the algorithm iterations (blue lines), and the chosen best path after iterations (red line).

4.4 Semester 2 Results

4.4.1 Ground Module Path Planning Results

The Path following simulation takes in a set of waypoint and outputs a graph with those waypoints on it and simulates the rover moving along that path. Figure 7 shows the waypoints plotted as well as a snapshot of the robots movement between waypoints 2 and 3 as well as between waypoints 4 and 5.



(a) Movement between wp 2 and 3

(b) Movement between wp 4 and 5

Figure 7: Matlab Rover Path Following Simulation

These results prove the initial path following simulation is a success. It is robust enough to take in a various number of waypoints, forcing the vehicles controllers to move the vehicle from one waypoint to the next and doing so successfully from initial point to final point. In addition to the visualization of the vehicles movement between waypoints the simulation also outputs the pose of the robot at each waypoint. From this we are able to confirm that the vehicle has the correct heading.

4.4.2 Ground Module Obstacle Avoidance Results

Now that there exist a working simulation to test the path following algorithm in the next step is to build in an obstacle avoidance algorithm and furthermore simulate that. This was also be done in Matlab initially as we did previously with the path following simulation. Both these two algorithms have been coded in order to work hand in hand. The obstacle avoidance algorithm used in our simulation is the aforementioned Probabilistic Road motion planning algorithm or *PRM*. This algorithm works by randomly sampling nodes in the free space of the environment and connecting them in order to create a collision free path for the vehicle to follow from the initial waypoint to the final waypoint. More specifically, as seen in Figure 6, the code does the following:

1. Updates the map by taking sensor measurements
2. Runs the Pure Pursuit Controller
3. Converts the output to wheel speeds
4. Performs a forward discrete integration step
5. Updates the visualization and deletes the image of the last robot

The output of the simulation are a binary map that contains the free path chosen amongst all the iterations of points as well as a visualization of the vehicle moving along that path and avoiding obstacles. Figure 8 shows the output of the binary map that highlights the free path in red and highlights all the intermediate randomly sampled nodes in blue.

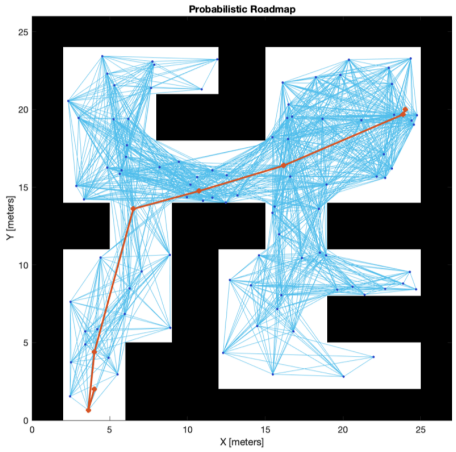


Figure 8: PRM Algorithm Free Path Computation

5 Semester 3 Work

5.1 Goals

Now that initial path planning and obstacle avoidance algorithms have been chosen and tested via simulation for the ground module it is time to do the same fro the aerial module. This semester of work was focused on investigating existing path planning algorithms and obstacle avoidance techniques for quadcopter type UAV's. Pathfinding is a mathematical problem that involves minimizing the cost of a path from starting vertex to goal vertex. When applied to drones its focus is to be implemented in order to find a path in 3-D space which can minimize a cost function defined in terms of chosen heuristics such as distance, risk, or time. Finding the best algorithm to fit hybrid vehicles specific need and adapting this to the physical vehicle is a common problem presented in robotics. Finally the last goal of the semester is finding a way to implement these algorithms on a vehicle in a safe environment for testing. For this a combination of ROS, OctoMap, and RViz was used to implement the algorithms which were passed to the simulation environment as nodes. This section will address how algorithm decision choice was made and further discuss the research behind it

5.2 Path Finding

Pathfinding is quite a complex problem to solve but it can be broken up into 2 main portions: global planning and local planning. Global planners work on finding a path to the goal while the local planner will translate this path into commands for the vehicles motor. Global planners usually build maps of the environment which causes them to be computationally more expensive than local planners.
The goal is to find an algorithm that can compute the beast path from start to goal while also taking into account power requirements. Local planners great feature is that they are computationally less expensive due to the fact that they do not build a map of the environment but this is also its greatest defaut. This causes local planners to create path solutions that are not the optimal path to the goal because they do not store information about already explored areas of the map. Due to this a global planner has been chosen in order to implement the chosen solving algorithm.

5.2.1 Dijkstra's Algorithm

In 1956 Dutch computer scientists Edsgar Dijkstra solved the problem of finding the shortest path in Graph Theory [7] for graphs such as the following in Figure 9

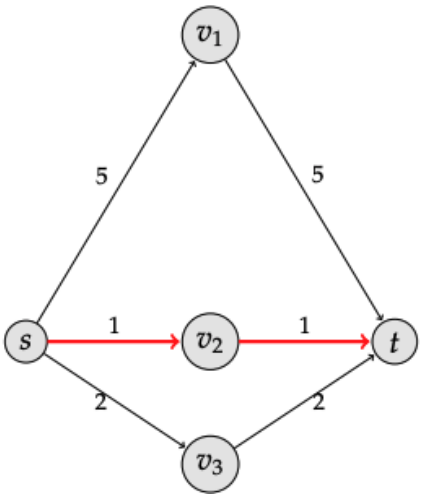


Figure 9: Graph with paths from start,s, to finish,t.

The pseudo-algorithm is described as follows in Algorithm 16:

Algorithm 1 Dijkstra's Algorithm

Require: Graph $G = (V,E)$ with non-negative edge weights
Require: Source vertex s
Require: Target vertex t
Ensure: The shortest path in G from s to t, if one exists
  for all $v \in$ V **do**
    d[v] $\leftarrow + \infty$
    previous[v] $\leftarrow$ undefined
  end for
  $d[v] \leftarrow 0$
  $Q \leftarrow V$
  while Q is not empty & $t \in Q$ **do**
    $u \leftarrow$ vertex in Q with min $d[u]$
    $Q \leftarrow Q \setminus \{u\}$
    for all edges $e = (u,t$ outgoing from $u)$ **do**
      if $d[u] + w(e) < d[v]$ **then**
        $d[v] \leftarrow d[v] + w(e)$
        previous $[v] := u$
      end if
    end for
  end while

The line that contains the **while loop** statement is where the algorithm expands its vertex u. It does this by choosing a vertex u that has not been previously chosen and checks, for each neighbor v, whether the distance to u combined with the weight of the edge is smaller than the smallest distance to v. While there are many algorithms that have better asymptotic run times than Djkstra's algorithm is heavily employed in many fields of research due to its simplicity and with a very simple modification one can get Dijkstra to have a run time of $\mathcal{O}(|\ \mathbf{E}\ |\ \mathbf{log}\ (|\ \mathbf{E}\ |))$. However well Dijkstra's algorithm works, it can also be quite slow because it is designed in a way that it focuses on the distance from s instead of focusing on getting closer to end goal t. Due to its simplicity and robustness a similar algorithm with faster solving time is needed.

5.2.2 A-Star

As previously mentioned the major advantage of Dijkstra's Algorithm is that it is simple to use and implement but unfortunately it is speed is too slow. However it is possible to create an algorithm similar to Dijkstra's that accounts for a user defined heuristic to make the run time faster. Dijkstra's algorithm always expands the vertex that is closest to s that has not been expanded yet causing it to expand all vertices even the ones that

contain a lower distance than that of the goal, t.
A-Star is an algorithm that was built off Dijkstra's algorithm that works in a similar fashion to Greedy Best First Search (BFS) algorithms. Greedy BFS algorithms is the same as Dijkstra's algorithm except that instead of selecting a vertex closest to the start goal, s, it selects the vertex closest to the end goal, t, to expand [8]. Due to this it does not guarantee the shortest path but it runs with a much shorter run time than Dijkstra's algorithm. This is where it gets its name "greedy" from; it tries to expand the vertex closest to the goal even if it is not the right path. This can be illustrated in the situation depicted by Figure 12 [9].

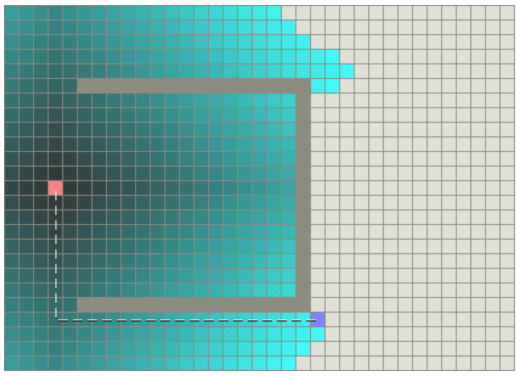


Figure 10: Dijkstra's Algorithm

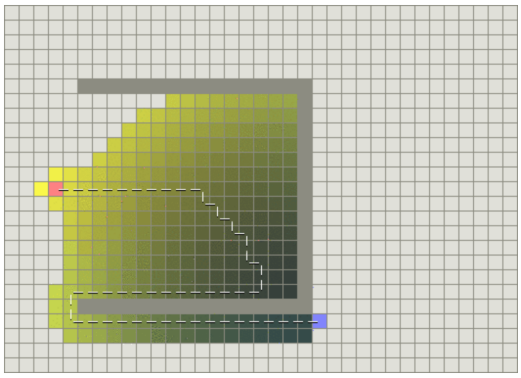


Figure 11: Greedy BFS Algorithm

Figure 12: Comparison of PathFinding Algorithm with Obstacles

Figure 12 shows an environment with a "U" shaped obstacle with both algorithms having the same start and end goal. As Dijkstra's algorithm expands all vertices it takes quite a while to reach the goal vertex but it chooses the good path. On the right the Greedy BFS shows its "greedy" flaw by expanding the vertices closest to goal point without considering the obstacles. It expands diagonally towards the goal but then has to go all the way around the obstacle causing the path to be much longer than need be. From these test it is evident that the chosen algorithm needs to be some combination of the simplicity of Dijkstra with the quick runtime of Greedy BFS algorithms.
In 1968 Stanford computer scientists' Peter Hart, Nils Nilsson, and Bertram Raphael published the A-Star algorithm. It combines the heuristic guidance employed in Greedy BFS algorithms while maintaining the simplicity of Dijkstra's algorithm [10]. Equation 1 shown below is the premise of A-Star:

$$f(n) = g(n) + h(n) \tag{1}$$

where $g(n)$ represents the exact cost of the path from starting point at any vertex n, $h(n)$ represents the heuristic estimated cost from vertex n to the goal. Figure 13 shows the same situation as depicted in Figure 12 but being solved by the A-Star algorithm [10].

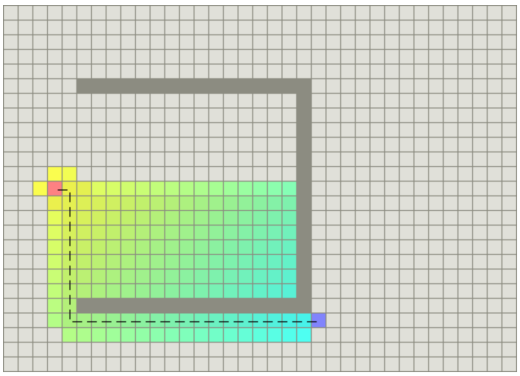


Figure 13: A-Star Algorithm with Obstacles

As you can see the algorithm provides the correct shortest path considering the obstacle. Additionally the graph shows how the algorithm created its path because the yellow color tiles are the nodes with the highest heuristic value (a high cost to get to the goal) and the darker the node the less expensive; as seen in the graph output

the yellow nodes remained near the start because the algorithm tended towards exploring vertices that have the smallest heuristic value.
The pseudo-code for the A-Star algorithm is shown below in Algorithm 16:

Algorithm 2 A-Star Algorithm

Require: Graph $G = (V,E)$ with non-negative edge weights
Require: Source vertex s
Require: Target vertex t
Require: Consistent heuristic function $h(v)$, $v \in$ V
Ensure: The shortest path in G from s to t, if one exists
  for all $v \in$ V **do**
    d[v] $\leftarrow +\infty$
    previous[v] $\leftarrow$ undefined
  end for
  $d[s] \leftarrow 0$
  $Q \leftarrow V$
  while Q is not empty & $t \in Q$ **do**
    $u \leftarrow$ vertex in Q with min $d[u] + h(u)$
    $Q \leftarrow Q \setminus \{u\}$
    for all edges $e = (u,v)$ outgoing from u **do**
      if $d[u] + w(e) < d[v]$ **then**
        $d[v] \leftarrow d[u] + w(e)$
        previous $[v] := u$
      end if
    end for
  end while

5.2.3 Chosen Algorithm

Three total, two main, algorithms have been researched in order to validate their ability to be implemented on the hybrid vehicle. The chosen algorithm will be tasked with solving the safest (lowest risk) path as well as the shortest path problem. In fact the safest path problem is transformed into the shortest path problem using logarithmic transformation [11]. Dijkstra's algorithm works well for the hybrid vehicles because of its simplicity but lacks speed and Greedy BFS algorithm works fairly quickly but lacks the guarantee of shortest path as a consequence the chosen algorithm for the vehicle will be the A-Star algorithm. The A-Star algorithm is flexible in terms of application and implementation and it runs very quickly due to its heuristic guided search as such the chosen algorithm for the vehicle was A-Star. Figure **??** shows another situation in which all three algorithms were tasked to solve [12].

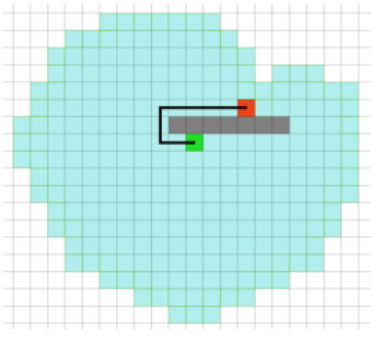


Figure 14: Dijkstra

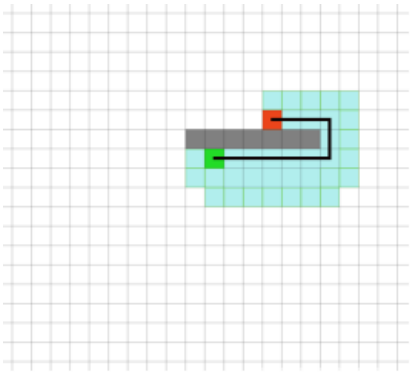


Figure 15: Best-First Search

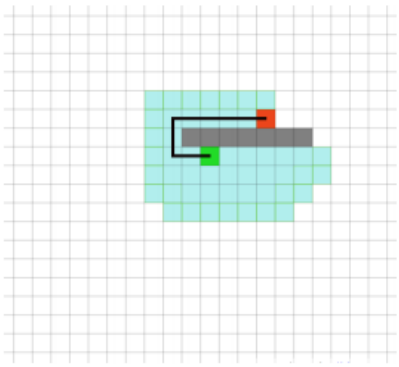


Figure 16: A-Star

Figure 17: Comparison of all 3 PathFinding Algorithms

The above, Figure 14, clearly shows how many vertices were expanded for Dijkstra's algorithm to solve the path which is correlated to the run time. In the middle, Figure 15, we have best first search algorithm that expands far fewer vertices but solves a path that is not the shortest path. Finally on the far left, Figure 16, we have the

A-Star algorithm that has also expanded far fewer vertices and produces a path solution that is the shortest path.
The vehicle prioritizes speed and safety during path planning because the heuristics are considered in our cost function. The cost function must be comprised of length, $L(P)$ and risk, $R(P)$ for a given path P. The problem with the combination of these two heuristics in the cost function is that it causes the function to be pareto-optimal. This means one heuristic cant be better without the loss or impact of the other heuristic. The pareto-optimality of the function causes the problem to be NP-Hard because there are an exponential number of paths [13]. As such some function is necessary for penalizing cost (distance) and risk (safety) so that the algorithm is able to produce a solution that is both safe and short. Many in the field of research of robotics have used the Minimum Cost Reliability Ratio (MCRR) function to penalize heuristics [14] shown by Equation 2.

$$F_{MCRR}(P) = \frac{L(P)}{1 - R_p(P)} \tag{2}$$

MCRR is usually applied to deal with communication networks for sending packets however when applied to a UAV it is not the same. This is because the MCRR function cannot know the impact of different types risks as some failures can lead to damage or complete loss of the vehicle. For example MCRR is unable to decide between finishing a mission or aborting a mission if the risk amounts to losing the vehicle. To solve this problem the MCRR function has been converted to something that will consider both the risk and the cost as the probability of the number of failures expected, R_E(P) shown in Equation 3.

$$R_E(P) = \sum_{e \in P} r(e) \tag{3}$$

Now that the heuristics are using an approach based on probability of expected failure, the cost function is chosen as a linear combination of distance and expected number of failures as depicted by Equation 4.

$$F(P) = L(P) + \lambda R_E(P) \tag{4}$$

5.2.4 Dynamic Search Algorithms

Algorithms such as Dijkstra and A-Star are great solutions to path planning problems but they are not dynamic. When calculating a path using one of the aforementioned algorithms it must calculate the shortest path multiple times as the vehicle changes states. This is considered a Constraint Shortest Path (CSP) problem which is also NP-Hard and thus it is important to consider the implementation of a dynamic search algorithm such as D*, D* Lite, or Anytime Dynamic A* (ADA*) to tackle this problem. Dynamic search algorithms make it easier to recompute the shortest paths for multiple graphs especially in cases where the graphs differ only a little from each other.
They work by assuming the end goal t will stay constant while the start s changes since the vehicle is moving. Additionally dynamic search algorithms account for sensors by accounting for the measurements in the space near the drones position causing changes in the graph to occur near s as depicted by Figure 18 below:

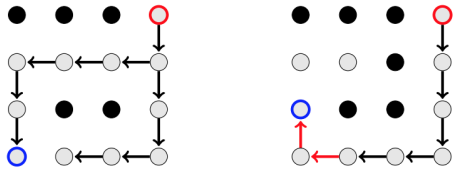


Figure 18: Dynamic Search

The figure above shows a path computed by a dynamic search algorithm where blue circle is the start, the red circle is the goal, and the black circles are obstacles. It shows the dynamic calculating from end goal to start instead of the reverse as the other algorithms do and it shows, on the right, how it recalculates when a new obstacle appears. This observation is quite important because it points to the dynamic search algorithms ability to intuitively only change a small part of the path from the previous state.
As great as the dynamic search algorithms are it cannot be employed on the current vehicle configuration due to the lack of power. Firstly, dynamic search algorithms change often it requires that all the data gathered by the vehicle must be stored somewhere accessible. Secondly, using the same logarithmic transformation applied

earlier to change the safest path problem into the shortest path problem causes there to be 10 vertices for each vertex. This means for each measurement the algorithm will need to do 10 updates in the search graph. Lastly, changing the cost function makes the whole search-tree obsolete because the vehicle will need to store multiple graphs which in turn increases the number of updates and measurements needed meaning longer run times and more power use.

5.3 Simulation Environment

5.3.1 Software Environment

Robotic Operating System (ROS) is a flexible framework for writing robot software. ROS uses nodes as its main communication method. A *node* is an executable that uses ROS to communicate with other nodes. These nodes communicate by publishing and subscribing to *topics* that receive messages. These messages are a ROS data type that is used when subscribing or publishing to a topic. ROS nodes can be written in either *python* or *C++*. The simulation module will use ROS nodes that implement the global planner and path planning algorithms. Those ROS nodes will then communicate with other nodes to implement the calculated path onto the simulated vehicle in the Gazebo simulation environment.

The **Gazebo** simulation environment is a well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Along with its compatibility and support of ROS it proved to be the perfect simulation environment. Gazebo creates a 3-D environment that can produce sensor feedback and mimic physical interaction between robots and objects. The algorithm will make use of this Gazebo feature and will use a depth camera to inform the vehicle about its surroundings. The depth camera measures the distance to points in its Field Of View and publishes the data as a point cloud (set of points in 3 dimensions) to a ros topic.
Additionally Gazebo and ROS will make use of the **OctoMap** library to track possible obstacles captured by the depth camera. ROS node *octomap_server* will create a 3-D probabilistic occupancy grid where each unit of space is stored as an *octree* as shown in Figure 19 [15]. Each octree represents the probability that that 3-D unit of space contains an obstacle.

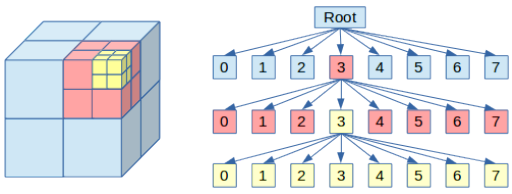


Figure 19: Octree with various levels

The **PixHawk PX4** autopilot will control the aerial module. The drone is simulated by its software in the Gazebo environment. The autopilot communicates with the system through the ROS node *Mavros* which uses the MAVLink communication protocol. The autopilot receives information on when to initialize the system, arm, and controls motors when a new path is received,

5.3.2 Communication

As mentioned before the communication between the various modules will be done using the ROS interface. Meaning all programs have been coded in the form of a ROS node. ROS provides the user with many introspection and command tools in order to better understand ones nodes'. The utilisation of the ROS command *rqt_graph* allows the user to display a graph of the running ROS nodes with connecting topics. Below is the output of *rqt_graph* for the global planner:

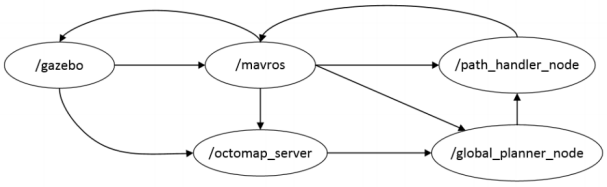


Figure 20: ROS Graph of the Global Planner Intercommunication

In Figure 20 the *ellipses represent nodes* and rectangles would represent topics. The large surrounded boxes represent various subsystems.
The ROS node *global_planner_node* handles the communication for the global planner. It receives information about the current position, goal position, and runs the planning algorithm. It then publishes the new set of waypoints calculated by the algorithm and sends it to a different ROS node called *path_handler_node*.

The *path_handler_node* communicates between the *global_planner_node* node and the *Mavros* node. Below Figure 21a shows the message flow (topics) from the PX4 to the global planner and Figure 21b shows the message flow from the global planner to the PX4; it also shows the data (message) that is flowing between the two nodes.

PX4 Topic	MAVLink	MAVROS Plugin	ROS Msg	Topic
vehicle_local_position	LOCAL_POSITION_NED	local_position	geometry_msgs::PoseStamped	mavros/local_position/pose
vehicle_local_position	LOCAL_POSITION_NED	local_position	geometry_msgs::TwistStamped	mavros/local_position/velocity
vehicle_trajectory_waypoint_desired	TRAJECTORY_REPRESENTATION_WAYPOINT	trajectory	mavros_msgs::Trajectory	mavros/trajectory/desired

(a) Message Flow from PX4 to Global Planner

ROS topic	ROS Msgs	MAVROS Plugin	MAVLink	PX4 Topic
/mavros/setpoint_position/local (offboard)	geometry_msgs::PoseStamped	setpoint_position	SET_POSITION_LOCAL_POSITION_NED	position_setpoint_triplet
/mavros/trajectory/generated (mission)	mavros_msgs::Trajectory	trajectory	TRAJECTORY_REPRESENTATION_WAYPOINT	vehicle_trajectory_waypoint

(b) Message Flow from Global Planner to PX4

5.4 Implementation

Implementation of these algorithms is focused on how to minimize our heuristics within our cost function. The cost function will aim to minimize traversal time from start to end of a given path, the energy consumed for a given path, and risk of taking a given path. This will then be implemented on a companion computer that will run all the ROS nodes needed to calculate the path accounting for the heuristics separately from the autopilot.

5.4.1 Companion Computer

Companion Computers can be used to interface and communicate with ArduPilot on a flight controller using the MAVLink protocol [16]. The companion computer gets all the MAVLink data produced by the autopilot (including GPS data) and can use it to make intelligent decisions during flight. The companion computer used on the vehicle will be the raspberry pi that is also in charge of controlling the ground module. It was ideal to make the program that controls the ground module and the aerial module obstacle avoidance algorithms on the same computer in order to make interfacing between the two modules seamless.

5.4.2 Cost Function

While the cost function aims to minimize the **distance**, **smoothness**, and **risk** it is imperative to extrapolate that from the quantitative data that is available which are *path length*, *tortuosity*, and *risk*.

Distance can be divided into either horizontal movement, vertical movement, or a combination of both. The horizontal movement can be written as a fixed cost that is comprised of the amount of time and energy needed to move one meter horizontally. The vertical movement will be written similarly but will have a higher cost than the horizontal movement because it requires more power to move the vehicle vertically than to move it horizontally. This is then divided into the following coefficients that make up Equation 8

- $L_h(P)$: horizontal distance of P
- $L_u(P)$: upward distance of P

- L_d(P): downward distance of P
- c_h: cost of horizontal distance
- c_u: cost of upward distance
- c_d: cost of downward distance

$$F_d(P) = c_h L_h(P) + c_u L_u(P) + c_d L_d(P) \tag{5}$$

with

$$c_h = c_t t_h + c_e e_h \tag{6}$$

where c_t and c_e are the unit-costs of time and energy and t_h and e_h are the time and energy needed to move one meter horizontally respectively. With these two equations we now have a cost function for distance that has been transformed into cost and distance.
Along with minimizing the distance of a path creating a smooth path is just as important. The **smoothness** of a path is defined as the tortuoisity of a path which is the quantification of the number of degrees in a turn. This is an important to consider within the total cost function because making turns increases the amount of energy and time consumed by the vehicle and as such it is important to quantify to know the true best path. As such Equation 7 shows how angles are penalized [12].

$$f_s(\theta) = c_t t_s(\theta) + c_e e_s(\theta) \tag{7}$$

where $f_s(\theta)$ is the cost for turning θ degrees and $t_s(\theta)$ and $e_s(\theta)$ are the time and energy needed to make a turn of θ degrees. In order to penalize the cost of turns a quadratic function was applied meaning that making a 90°is 4 times more costly than making a 45°turn. This is done so that the vehicle is more tempted to make two 45°turns because it is smoother and cost less than making one sharp 90°turn.

Finally the **risk** heuristic is based on probability. The probability of a 3D subspace being occupied, probability that the neighboring subspace is occupied, and finally the probability of occupancy based on altitudes. The probability of the 3D subspace being occupied will be given by the Octomap occupancy grid. The probability of a neighboring subspace being occupied is also given by Octomap. This probability is the probability of the drone being in a neighboring vertex or the drones ability to stay within a defined path and not end up in the subspace surrounding the current path. Finally the probability of occupancy based on altitude is useful for planning a path in unexplored space. It uses a prior probability that is the probability of a vertex v containing an obstacle [12]. For example if higher altitudes tend to have a higher probability of being occupied then when calculating a path the vehicle will tend more to fly at lower altitudes in order to reduce risk.

5.4.3 Heuristics

As previously mentioned the obstacle avoidance algorithm will make use of the A-Star algorithm in order to solve our path problem. In order for the A-Star algorithm to produce the optimal path in an optimal time, it must make use of its famed heuristic guidance feature. Now that the cost function has been defined a heuristic function should be defined similarly.

There are three main heuristics for pathfinding that are deployed in robotics; the Straight-Line distance, the Euclidean distance, and the Manhattan distance. In this case the Straight-Line distance is a good option but it will be hard to define a lower bound. The Euclidean distance can be used in this case as it does not reflect path distances on grids. Finally the Manhattan distance works but it is not a lower bound and does not consider diagonal moves. Thus an altered Manhattan distance has been used, shown by Equation 8 [12].

$$h_d(u) = |u_x - t_x| + |u_y - t_y| + |u_z - t_z| - (2 - \sqrt{2}) \min\left(|u_x - t_x|, |u_y - t_y|\right) \tag{8}$$

This altered form has the same first three terms as the Manhattan distance but has a fourth term that accounts for using diagonal moves.
The smoothness heuristic is found by calculating the number of 45°turns that are needed to get from current pose to end goal because taking 45°turns always minimizes the quadratic cost. Equation 9 below shows the heuristic for the cost of smoothness:

$$F_S(P) \geq 2 f_s\left(45^\circ\right) \tag{9}$$

The risk heuristic function is defined by assuming the case of a path through completely unknown space. Meaning the heuristics for the cost of the risk is the cost of creating the shortest path in which every 3D subspace (voxel) is an unknown. Thus the following heuristic, Equation 10, created by members of ETH Zurich was used:

$$h_r(u) = p_{\text{prior}}\ (u_z)\,(h_d(u) - 1) + \text{risk}(t) \tag{10}$$

The heuristic functions mentioned above developed by ETH Zurich [12] were implemented into the A-Star algorithm used on the vehicle of this thesis. As the requirements for path planning were similar it proved to be the right decision but as we were unsure we ran these algorithms in a simulation environment to be safe.

5.5 Semester 3 Results

5.5.1 Operating Simulation

The global planner ROS node is launched using the ***roslaunch*** command line tool. This tool takes one or more *.launch* files as arguments. Figure 22 below shows the command used to launch the global planner:

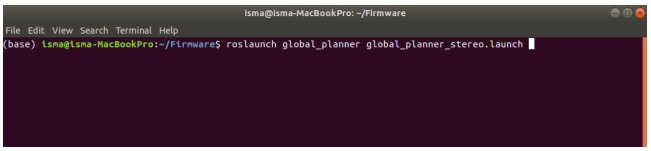


Figure 22: Roslaunch of Global Planner Node

Once ROS launches the global planner node it automatically opens up the world simulation environment with the drone inside as shown below in Figure 23.

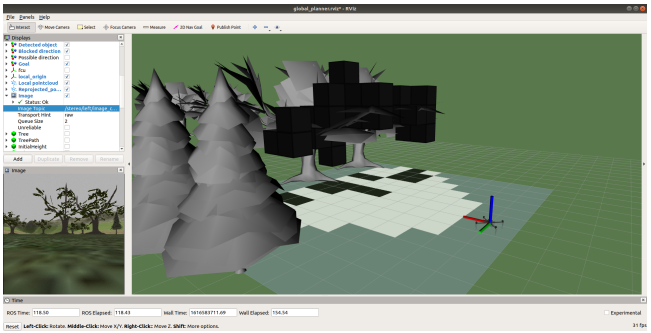


Figure 23: Initialization of Simulation Environment

On the left side we find the world environment with the drone initialized. On the top left we have all the various topics connected to the simulation environment and finally in the bottom left we have the visual of the image topic from the depth sensor found on the drone. Once the drone is initialized it is possible for the user to define points with the click of a mouse however because we want to imitate real missions using *QGroundControl*. In order for anyone to upload a mission to a vehicle via QGroundcontrol it must be in *OFFBOARD* mode and it needs to be armed as it would be on a physical vehicle. In order to do the same thing in the simulation the user must use the commands shown in Figure 24 another terminal to put the vehicle in *OFFBOARD* mode and then arm it.

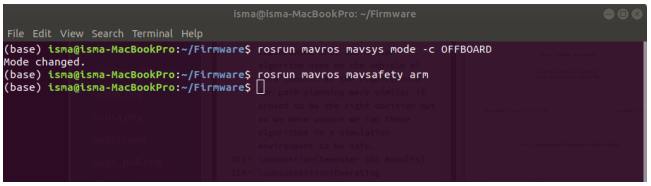


Figure 24: Commands to arm and put PX4 in OFFBOARD mode

After *QGroundControl* recognizes that the vehicle has been armed and placed in OFFBOARD mode the simulated vehicle will fly 3 meters up and wait for a new mission or end goal as shown in Figure 25.

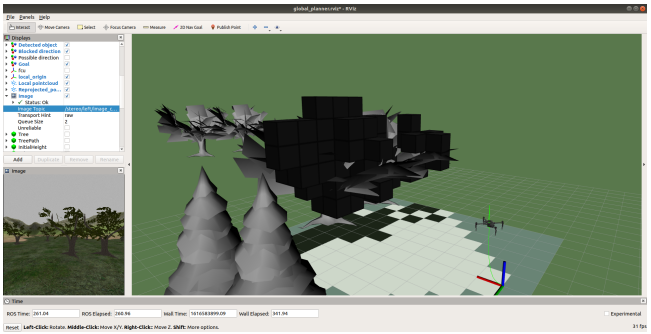


Figure 25: Simulated vehicle waiting for new mission

Now that the vehicle is waiting for a mission it is time to upload one from *QGroundControl* as we would in a real mission. In *QGroundControl* the user can define a mission comprised of a waypoints and then upload it to the vehicles PX4 autopilot. The mission will look as shown below in Figure 26, with the created mission displayed and with the option for the user to confirm the upload.

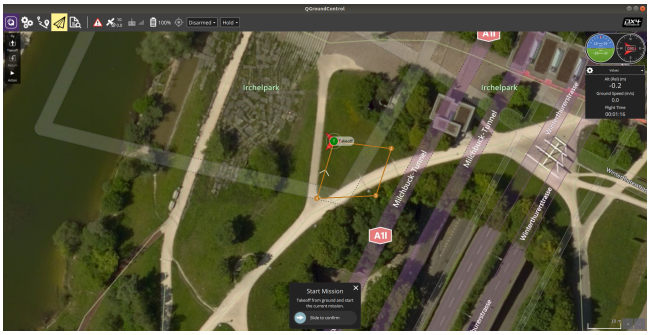


Figure 26: QGroundControl Mission

5.5.2 Simulation Output

Now that a mission has been defined in *QGroundControl* everything is ready for running the simulation. Once the mission is confirmed the vehicle will follow the mission and employ the path planning algorithm between each pair of waypoints i.e. between waypoint 1 and waypoint 2. The simulation has been developed to create a green line to trace the path the vehicle has taken within the simulation as shown in Figure 27.

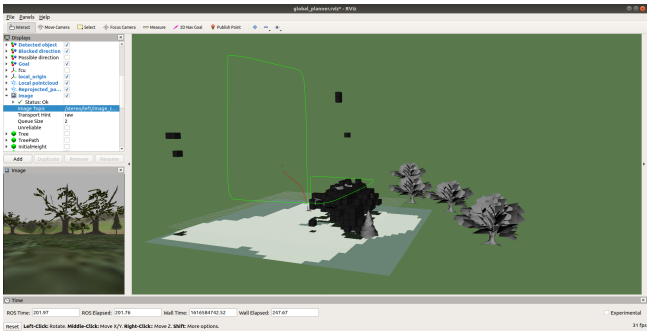


Figure 27: Completed Mission in Simulation Environment

This shows a successful implementation of the path planning algorithms in the simulation environment. While the simulation is running the terminal will also display information regarding the global planner as shown in Figure 28.

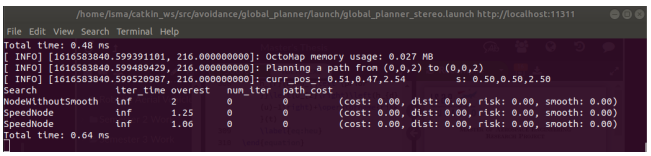


Figure 28: Terminal Output

The terminal shows how much memory the OctoMap node is using. Then it tells the user the start and end goal for the path planning algorithm to solve a path for. Finally it begins to plan the path and displays the cost of each heuristics. This information was useful to output in order for us to better understand the behavior of the path planning algorithm for further optimization.

6 Conclusion

In 2019 the project was handed to me with the vehicle created and with working computers to control both aerial and ground modules. Additionally a preliminary obstacle avoidance algorithms for the ground module had been created. I was tasked with researching and implementing path planning and obstacle avoidance algorithms that the hybrid rover drone would support. After the obstacle avoidance algorithm is completed I was to find a way to make it so that the user of the vehicle could choose a mission type that employs heuristics to minimize power consumption or reach the end goal as quickly as possible. Finally I along with my partner working on the controls part, Ricardo Rodriguez, was tasked with creating a communication module for the vehicle to be able to switch between utilizing the ground or aerial modules. There are currently little to no obstacle avoidance or path planning modules that have employed for hybrid rover-drone vehicles so combining pre-existing algorithms and adapting them or creating new algorithms for our hybrid vehicle was the basis of my project and thesis for the past 14 months.

Semester 2 focused on finding and further developing the obstacle avoidance algorithm that came with the vehicle. Considering the added weight of the aerial vehicle on top of the rover when the vehicle would be operating with the ground module, it was pertinent to implement an obstacle avoidance and path planning algorithm using differential drive kinematics. This will allow for stability in moving with the vehicles irregular form. As such the *ControllerPurePursuit* from the MATLAB Robotics Toolbox was used. This decision was also partially made because this controller allows the user to define the look ahead distance. This is good in the case of the vehicle in this thesis because it allows us to increase the look ahead margin to work in accordance to the size of our vehicle while still staying on the path. The obstacle avoidance algorithm uses the same controller to force the vehicles wheels to move. It then runs a Probabilstic Roadmap to create a collision free path. The outcome is a map that shows the collision free path after iterating through many various paths. After running this in the MATLAB simulation with our vehicles description the outcome has proved that it can be used for preliminary Hardware In The Loop (HITL) test.

After establishing initial obstacle avoidance algorithm in semester 2 semester 3 was focused on finding and implementing obstacle avoidance algorithms on the aerial module. This involved first deciding which type of planner works best for our current vehicle, second deciding on algorithm best suited for our needs, and third creating a way to simulate these algorithms on a $PX4$ autopilot controlled vehicle. After conducting a literature review it was decided that a global planner was better suited than a local planner due solely to the local planners inability to guarantee an optimal path to the goal. For the choice of algorithm there were many from the popular Dijkstra algorithm to the speedy Anytime Dynamic A-Star algorithm but our choice was the also well famed A-Star algorithm. The reason for this is we needed an algorithm that would be quick and still produce an optimal path due to the computing capacity of our companion computer the raspberry pi. The A-Star algorithm uses guided heuristics to make it faster and remain precise which also served as a good premise for the next step of the obstacle avoidance algorithm; varying mission types. In order to use this it was decided that the heuristics will need to reflect our vehicles mission type; power saving and minimal time. For this the heuristic functions developed by master's students at ETH Zurich were used as they met the requirements for the mission types that this vehicle would eventually employ.

With the obstacle avoidance algorithm for the aerial module implemented the next step was to create a simulation environment to test in for two reasons: the current global situation does not allow for field test and it is necessary to test the vehicle in a safe environment first to prevent damage to the physical vehicle. ROS +

Gazebo were chosen as simulation environments for its open source framework, its large support system, and its vast range of robotics compatibility. ROS is largely used in the robotics community and as a result it has many libraries necessary for simulating the physics involved with the irregular vehicle shape and its consequent irregular flight. Developing with ROS also gave us the ability to create a node that will simulate the $PX4$ autopilot software. This allows us to create real like missions of predefined waypoints in QGroundControl and feed them into the obstacle avoidance algorithm nodes in order to see how they vehicle would perform in a real time situations.

7 Future Works

At the end of this thesis the vehicle has initial path planning and obstacle avoidance algorithms that are ready for further situational simulation tests and initial hardware tests. The obstacle avoidance algorithms for both modules have been tested in simulation separately and have proved to work well. In order to get more accurate information regarding the behavior of these algorithms on our hybrid vehicle it is important that two things take place; the vehicle must be modeled in more detail in the simulation environment and preliminary Hardware In The Loop tests must be done in order to understand how to integrate these algorithms onto physical hardware. In order to validate the simulation of these algorithms even further, a vehicle with the same dimensions and inertia's needs to be developed in 3D. It then needs to be fitted with the correct sensors and MAVLink connections. This will allow us to get a better idea of how the inertia of the vehicle will change the vehicles interaction with the algorithms.
Secondly it is important to begin testing these algorithms on hardware components to better understand how to make the connections and to understand what parts may need to be changed. Conducting hardware tests will also clarify all the necessary components needed to get the vehicle flying and moving initially in *OFFBOARD* mode via the obstacle avoidance algorithms.
Finally it is important that the communication module between the ground and aerial modules be solidified. This is key because it will allow for the vehicle to switch between using either module when necessary; which is an imperative portion of the project. After these works have been completed it will be possible to have a working module that can conduct a simple autonomous mission with low failure rate.

References

[1] Anton Sambalov. Autonomous Robotic Aerial Vehicle: S3 Project Robotic. March 2020.

[2] URDF in Gazebo, http://gazebosim.org/tutorials?tut=ros_urdf.

[3] Path Following for a Differential Drive Robot., https://www.mathworks.com/help/robotics/examples/path-following-for-differential-drive-robot.html.

[4] differentialdrivekinematics, https://fr.mathworks.com/help/robotics/ref/differentialdrivekinematics.html.

[5] Pure Pursuit Controller, https://fr.mathworks.com/help/robotics/ug/pure-pursuit-controller.html.

[6] Path Planning in Environments of Different Complexity, https://fr.mathworks.com/help/robotics/examples/path-planning-in-environments-of-difference-complexity.html.

[7] E. W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numerische Mathematik*, 322(1):269–271, 1959.

[8] Introduction to A*, 1997.

[9] Introduction to A*, http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html.

[10] P. E Hart, N. J. Nilsson, and B Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics.*, 4(2):100–107, 1968.

[11] Myungsoo Jun and Raffaello D Andrea. Path Planning for Unmanned Aerial Vehicles in Uncertain and Adversial Environment. *Cooperative Control: Models, Applications and Algorithms*, pages 95–111, 2003.

[12] Vilhjalmur Vilhjalmsson. Risk-based Pathfinding for Drones. pages 8–9, May 2016.

[13] N Katoh. A Fully Polynomial Time Approximation Scheme for Minimum Cost-Reliability Ratio Problems. *Discrete Applied Mathematics*, 35(2):143–155, 1992.

[14] matthew Greytak and Franz Hover. Motion Planning with an Analytic Risk Cost for Holonomic Vehicles. *MIT*, 2013.

[15] David. Advanced Octrees 1: preliminaries, insertion strategies and maximum tree depth, 2014.

[16] ArduPilot Dev Team. Companion computer.