Master of Aerospace Engineering
Research Project

Autonomous Robotic Aerial Vehicle's Control in Complex Environments

S2 Project report

Author(s): Karthik Mallabadi, Ryan Xavier Dsouza

Due date of report: 19/06//2021
Actual submission date: 19/06/2021

Starting date of project: 25/Jan/2021

Tutors: Raghuvamsi Deepthimahanthi, Yves Briere

Table of Contents

Declaration of Authenticity

This assignment is entirely our 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. We confirm that no sources have been used other than those stated.

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

Date
19/06/2021

Signature

Karthik .M.

[illegible]

Abstract

Unmanned Aerial Vehicles (UAV) have a wide spectrum of applications such as surveying or delivery, while rovers are commonly used in space and cross terrain missions. However, the combination of both paves' way to a new field of research. The main goal of this project is to program an existing hybrid vehicle which will be fully autonomous and can move both in ground and air.

To make the vehicle autonomous in a dynamic environment, development of a path-planning algorithm is essential. This document comprises the investigation of various path-planning algorithms. A comparison study carried out on these search algorithms to find the best one will be shown in this report, along with the extension of the selected D* lite algorithm from 2D to 3D.

Safety of the vehicle is crucial and in order to satisfy this aspect, a safe landing algorithm which is able to make decision on whether to fly or land based upon the battery information will be simulated in ROS environment. Working of safe landing planner which detects flat areas while landing will also be explained.

Finally, the report will conclude with the key findings of this work and also the futures aspects that will be performed in the upcoming semester

Keywords: UAV, Rover, Path-Planning, D* Lite Algorithm, Safe Landing

1 Introduction

In recent years, rovers and UAVs have been of interest due to their wide spectrum of applications. Rovers are generally used in space and cross terrain missions. These vehicles are highly efficient and can easily traverse all types of terrains whilst being robust in all conditions. Due to this there is a large demand in space exploration, in battlefield and disaster zone type conditions.

On the other hand, Unmanned Aerial Vehicles have risen in popularity for several reasons. The main ones being: 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. UAVs possess high agility which allows them to commute places faster but at an expense of consuming more power.

The main focus of this project is to program an autonomous hybrid model of UAV and rover which combines the capabilities of both systems such that it can travel places quicker whilst being efficient in terms of energy and power. To achieve the desired goal, sensors will be used to detect obstacles that might come on the way. In case an obstruction is detected, the robot should be able to choose the most appropriate path to go past the obstacle: either go around it on the ground or cross it in the air by using the air module. This choice will depend on three key parameters: power, time and energy.
In case the rover lacks power to avoid the obstacle and go past it, the air module takes over. Moreover, when it comes to saving energy, the ground module is preferred, as long as the avoidance can be done on the ground. Finally, when there is a time constraint, air module will be selected since its faster when compared to the ground module.

This document outlines the key objectives and tasks involved in the project, as well as the work that has been completed so far in the second semester and also the goals for the upcoming semester.

2 Project Definition

2.1 Context and key issues

This work has been previously conducted by Anton Sambalov, Sakshi Chaudhary and Nandhini Raghunathan, who have designed and assembled a vehicle composed by a ground and an air module. Furthermore, Anton Sambalov also worked on the set up of different types of equipment as well as on the communication between them. Posteriorly, some simulations of required algorithms were also performed [1].

Previous year, Ricardo Rodriguez and Brian Ismael continued the project work and have setup a baseline path-planning and obstacle avoidance algorithm for the air module along with the complete obstacle avoidance algorithm for the ground module [2] [3]. Communication between the ground and the air module has also been established [2]. The main focus was then to improve and research about the existing algorithm in order to reach the desired goal and also include an extra module which is responsible for the safety of the vehicle.

One of the main issues of this project is to give ability to the vehicle to understand its surrounding environment and accordingly create intermediate waypoints, so as to reach the goal quickly with minimum consumption of energy. In situations where the distance to go around the obstacle is less (obstacle 1), ground module is preferred as it is highly efficient in terms of energy, on the contrary when the obstacle is large (obstacle 2), air module will be used so that the time taken to go past the object is less. These situations are illustrated in figure 1.

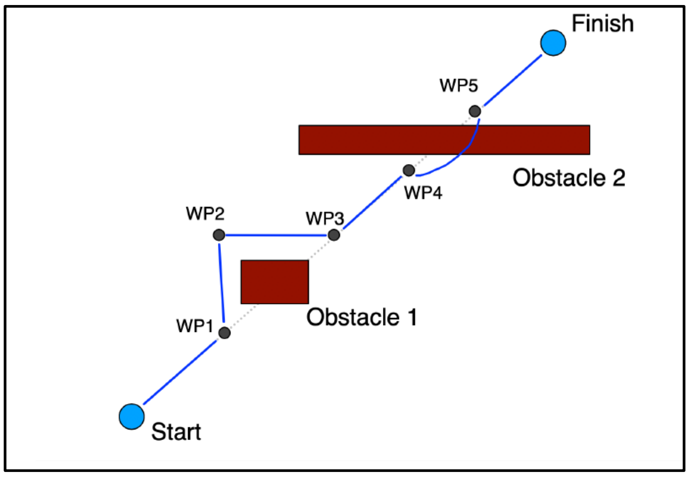


Figure 1 Desired behaviour of the vehicle

Providing intelligence to the vehicle so that it is able to make decisions based on the previously mentioned criterions and switch between the modules accordingly will be a tough task. Next challenge will be to develop a safe landing algorithm, which detects flat areas underneath the vehicle suitable for the landing in case of emergencies or when the vehicle has reached its goal point.

2.2 State of the Art

The work started with the investigation of various path planning algorithms that have been utilised in the past for unmanned aerial vehicles. Nicole Chow et. al. [4] have presented an approach that can be used by autonomous drones to travel from source to destination using Dijkstra's shortest path algorithm. The optimal path was found by capturing the surrounding environment and decomposing it into grid cells, followed by a conversion of the modelled environment into graph.

Shafkat Islam et. al. [5] have proposed a distributed path planning algorithm for a team of UAVs. With this algorithm, UAVs are able to avoid collisions with obstacles. The main difference with similar algorithms is the destination for the vehicle is not fixed and the UAV locate themselves to cover a time-varying mission area.

Marian Lupascu et. al. [6] elaborated a technique to generate collision free path for an autonomous UAV in 3D environment with static obstacles. They considered a classical 2D decomposition technique and extended the same to 3D space based on rectangular cuboids. This approach was compared to the Rapidly-exploring Random Tree (RRT) technique whose main idea is to create pseudo-random points and link these points based on line of sight, thus resulting in a tree-structure from which a trajectory for the drone is found. The comparison showed that, the trajectory from RRT method is the shortest among the two methods and is obtained quicker.

Elaf Jirjees Dhulkefl et. al. [7] in their work have compared two-2D path planning algorithms for UAVs, namely the Dijkstra's algorithm and A^* algorithm. The main focus was to avoid obstacles and generate the shortest path to the goal point. They concluded that, both the algorithms generated the same path length, but the time taken by the A^* algorithm to reach the target was much less than the Dijkstra's algorithm.

Sven Koenig et. al. [8] have proposed a new algorithm called D* Lite for fast replanning method for robot navigation in unknown environment. The algorithm searches from the goal vertex towards the current vertex, by utilizing similar heuristics estimation as A* algorithm. D* Lite was able to produce solutions for search tasks much faster by combining advantages of incremental and heuristic search.

2.3 Justification of the potential degree of novelty

There is a lot of literatures that explore UAV and Rover technology. Rovers are one of the most used robotic vehicles, which are used to perform tasks in areas with hard accessibility and are frequently utilized in the space sector [1]. Significant developments and innovations have been done in the rovers and the UAV technology, but a hybrid model between these systems which is capable of moving both on the surface and in the air has not yet been designed. Furthermore, the existing D* Lite algorithm for 2d case has been extended for 3d environment, also the heuristics function which will be explained in detail in the subsequent sections have been improved accordingly.

Another new idea that has been implemented in this work is the estimation of energy required to reach the waypoint by analyzing the real-time current consumption data along with the time required to reach the waypoint. This idea is incorporated in the safe landing algorithm, which makes decision on whether to go to the next waypoint or to land based on the computed energy.

2.4 Aims and objectives

As mentioned previously, the main focus of this project is to program an autonomous hybrid model of UAV and rover. For the vehicle to be autonomous, firstly there is a need to develop a path-planning algorithm which provides the vehicle the intelligence to reach the destination on its own by studying its environment. In order to provide this intelligence, it is crucial to research about the existing path-planning algorithms, understand the pros and cons of each algorithm, select the most appropriate algorithm and modify it according to our requirements. Safety of the vehicle is crucial and in order to make the vehicle safer, a safe landing algorithm will be developed which allows the vehicle to land in case of battery emergency or once it has reached the destination. This sums up the objectives of the project for the second semester. The methods incorporated in achieving these objectives and the results obtained are explained in the following sections.

3 Investigation Methods

3.1 Vehicle Components

As previously mentioned, the robotic aerial vehicle is a hybrid system composed of air module which is a quadcopter and a ground module which consists a rover. Both modules have different controllers; Raspberry Pi micro-controller board controls the ground module and also communicates with the air module, while Pixhawk PX4 flight controller controls the air module. The full model assembled by Anton Sambalov, Sakshi Chaudhary and Nandhini Raghunathan is shown in figure 2. The components used in both air and ground module are mentioned in the following two sections.

3.1.1 Air Module

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 ground module is comprised of a rover that has a base similar to a tank. The rover consists of continuous track which provides maximum control on rough terrain and furthermore, prevents the vehicle from getting stuck. The ground module is made of the following components:

- 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 AA batteries in series.
- Motor Driver: The TB6612FNG motor driver is able to control speed and direction of the two rear motors. Having a supply range of 2.5V to 13.5V, it suits our motors.
- Computer: Raspberry Pi is used as the onboard computer to let the motor driver know when and which way it should run the motors and also to communicate with the air module.

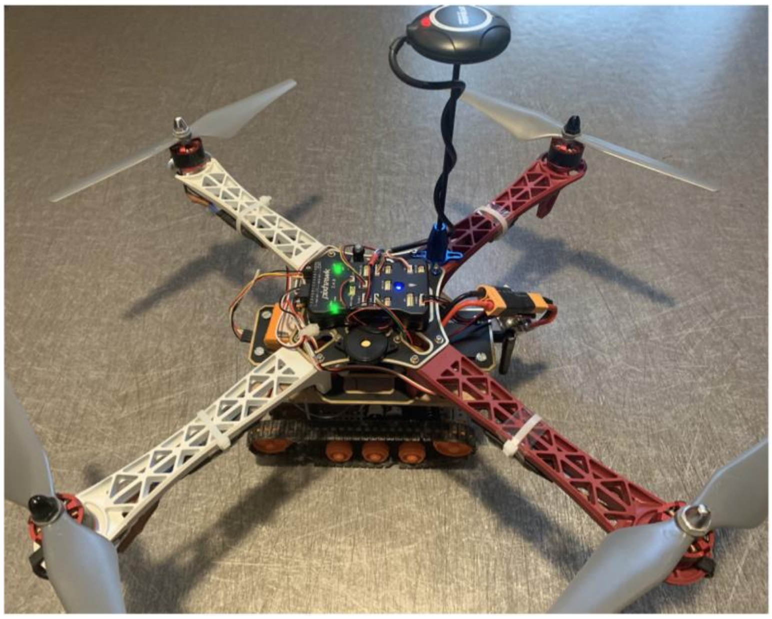


Figure 2 Assembled Vehicle

3.2 Software Modules

This section details out the software systems that were used throughout the project work. The Robot Operating System (ROS) was used as a framework. It consists of libraries and tools which allow to design and coordinate robot software. A ROS-based process is called a node and different nodes communicate between each other by sending messages via ROS. The concept of node will be very important throughout the whole content of this report.

Finding a suitable simulation environment for testing algorithms on the vehicle is very crucial as it allows us to validate our algorithms in a simulated environment before actually testing the vehicle in the real world. Furthermore, having the luxury of observing and analyzing the performance of algorithms in 3D is very important for understanding how the robot is going to behave in the real world. The selected simulation environment was the open-source 3D simulator Gazebo [9] which is compatible with ROS. It provides a robust physics engine which is able to detect collisions between the robot and the surrounding environment, providing the user with a precise estimation of what would happen in reality. It also enables the utilization of sensor models, which provide the sensor data to the robot. For example, the sensor data coming from the Gazebo sensor models can be published on a ROS topic. Topics are the medium through which ROS nodes exchange messages between each other. Therefore, a publisher node can publish data on a topic and other subscriber nodes can then subscribe from that same topic in order to gather information.

QGroundControl (QGC) is a ground control station which provides full flight control and mission planning for any MAVLink enabled drone. It allows us to monitor the vehicle in real-time by continuously providing data about the position, velocity, orientation of the vehicle and also information about the power and battery consumption. QGC gives a real time map based on the GPS location, and this was important in the project to test the safe landing algorithm. As the main idea behind the safe landing algorithm was to detect flat areas where the vehicle can land safely, the map generated by QGC provided a kind of simulation environment when it comes to ground surface.

As stated before, a Pixhawk flight controller is used to control the air module. Gazebo allows the simulation drones which are controlled by the PX4 flight control software. The PX4 autopilot communicates with the system through a ROS node called MAVROS, which uses the MAVLink communication protocol. For example, an obstacle avoidance node collects information from the PX4 autopilot regarding the current vehicle state, using the MAVLink protocol. It then sends the data to the flight controller with the computed setpoints, which the vehicle will then follow.

At this point, the whole configuration of PX4 and ROS has been explained. The relationship between each component is shown in figure 3. It can be seen, PX4 Software in the Loop [10] communicates with the simulator, which is Gazebo, in order to receive sensor data from the simulated world and then send motor and actuator commands. It can also communicate with a Ground Control Station (QGC). The Offboard API represented in figure 3 is ROS.

As previously stated, Raspberry Pi is used as an onboard computer to control the ground module. Another reason why Pi is used is briefed in this section. Due to the fact that ROS is only supported on Linux, a Virtual Machine with Ubuntu OS was used for simulations. Having installed Ubuntu on the Raspberry Pi, makes the process easier as all the nodes being simulated can directly be implemented on the real vehicle. The only differences being that the actual sensors will be feeding data to the PX4 controller instead of Gazebo sensor models and the motor commands will be computed by the PX4 flight controller.

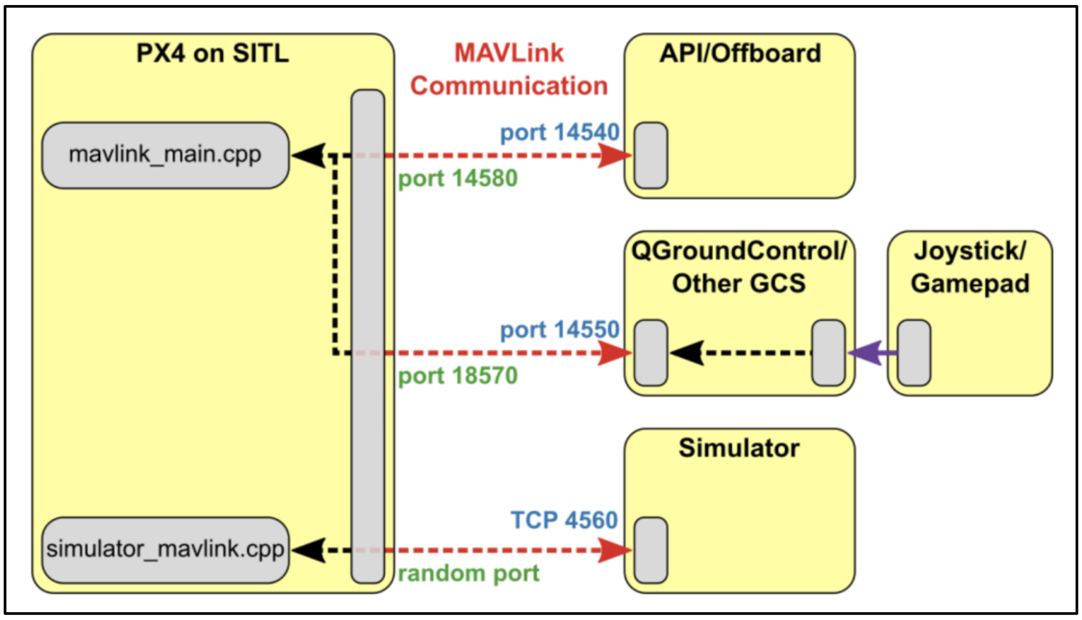


Figure 3 Communication architecture of the vehicle

3.3 Investigation of search algorithms for path planning

The baseline path planning module from previous work has been developed based on A* algorithm which is more suitable for a static environment. In case of dynamic environment, every time there is a change, the algorithm has to compute a new path by estimating the heuristic for all the nodes again. The Dijkstra algorithm that is seen in state of art also recomputes everything to generate the path. Hence, these algorithms are suitable for unknown and dynamic environments. In the following sections, A*, Dijkstra, and D* Lite algorithms are investigated and a comparative study has been carried on them.

3.3.1 A* Algorithm

The A* algorithm finds the path by evaluating the cost function $f(s)$. For a given node 's' in the grid, the cost function is sum of the distance between the current node and the start node, $g(s)$, and the heuristic estimated distance from the current node to the goal node, $h(s)$.

$$f(s) = g(s) + h(s)$$

This algorithm requires static starting point and goal point. The search tree starts from the start node and spread towards the goal node using heuristic estimation. When the start node changes, the search tree has to be recomputed from the begin. This process will require more computation time in obtaining the required path [11].

3.3.2 Dijkstra Algorithm

In Dijkstra algorithm, the map is discretized into graph data with details of vertices and their distances. This algorithm finds all the possible paths between the start and goal vertex and chooses the shortest path based the distance weightage.

Since the algorithm computes all the possible paths, the computation time required to obtain the path is higher. This algorithm ensures that the path obtained is the shortest path, but it is not suitable for the dynamic environments where quicker paths should be computed to avoid collision into the moving obstacles. And this algorithm also needs prior information on the map that it is searching. In an unknown environment it is not possible to generate the path unless the vehicle acquires the map information.

3.3.3 D* Lite Algorithm

D* Lite algorithm is the dynamic implementation on A* algorithm. The main advantage of this algorithm is that the search tree begins from the goal point and moves towards the current position. When the current position is changed, the root of the search tree will be the same. Only the ending branches needs to be updated.

For every node visited by algorithm, the heuristic information is stored as key along with the location of the node. Hence, when the algorithm reaches the same node again, it can use the values stored previously and save the computation time in recomputing the heuristic from the scratch. This ability of the algorithm enables it to be efficient in dynamic environments.

$$\text{Heuristic: } heur(s) = 0.4142 * min\ (dx, dy) + \max\ (dx, dy)$$

$$\text{Key: } [\min(g(s), rhs(s)) + heur(s); \min(g(s), rhs(s))]$$

where, $g(s)$ is the distance estimation from current node to the goal node, and $rhs(s)$ is the sum of distance from its previous node to the current node and $g(s)$ value of its previous node.

The heuristic estimation can also be updated with more than one constraint abiding by the dynamics of the vehicle like the energy consumption. The path generated by such heuristic estimation will not be the shortest path, but it will be the efficient path which consumes less energy and time to reach the goal point with the dynamics of the vehicle.

For the comparison study, a simple test map (see figure 4) is considered to find the path. The map contains 2 obstacles with dimension of 10 x10 (meters). The path is obtained using the above-mentioned algorithms. For this case, only distance constraint is used as heuristic estimation in the D* Lite algorithm. The results from this study case are discussed in the result section of this report.

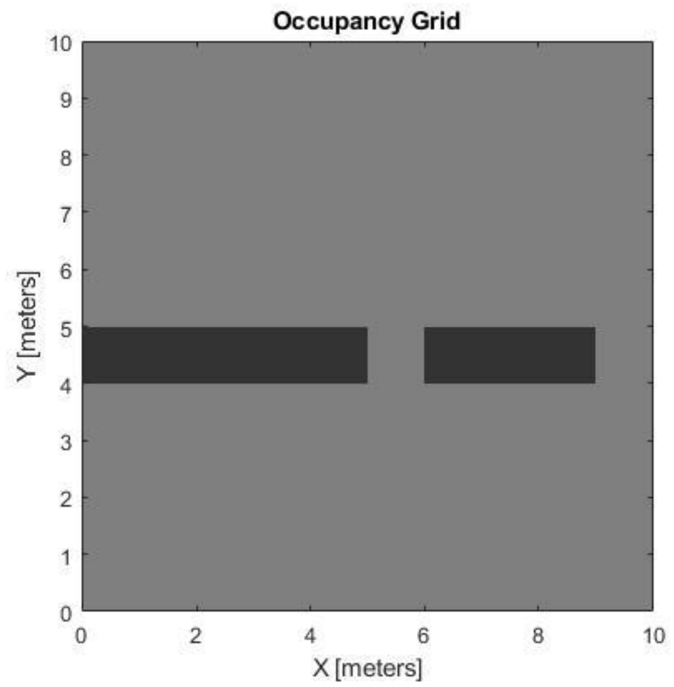
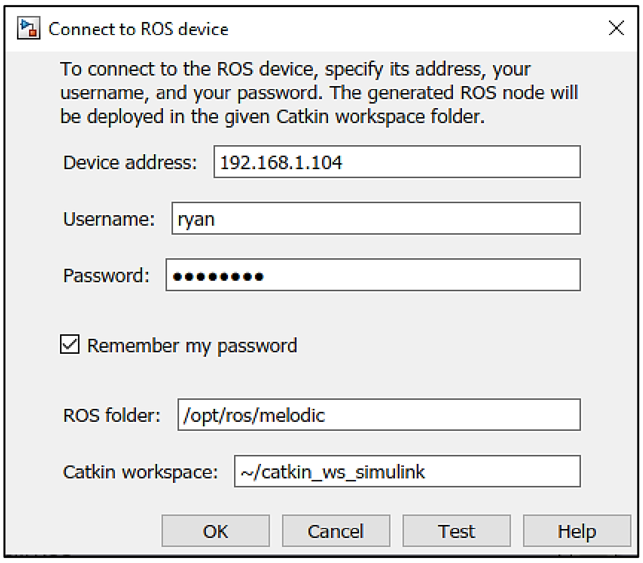


Figure 4 2D map with obstacles

3.4 Simulink model to create ROS node that connects companion computer and Pixhawk

A Simulink model is created that can generate the necessary C++ code for the companion computer to communicate with the Pixhawk board. Initially to setup the model, the configuration setup has been done. The model is connected to the companion computer through its IP address (figure 5).



Connect to ROS device

To connect to the ROS device, specify its address, your username, and your password. The generated ROS node will be deployed in the given Catkin workspace folder.

Device address: 192.168.1.104

Username: ryan

Password: ••••••••

☑ Remember my password

ROS folder: /opt/ros/melodic

Catkin workspace: ~/catkin_ws_simulink

Figure 5 Configuration setup between ROS and Simulink

The desired setpoints for the vehicle is obtained from the path planning algorithm. Then the set of waypoints are converted to signals using the Bus Assignment in Simulink. The blank ROS message block determines the structure of the signal. Using ROS publishing block, the message is then published to the topic **/mavros/setpoint_position/local** (figure 6)**.**

A ROS subscriber block is used to retrieve the position of the vehicle as feedback. It subscribes to the topic **/mavros/local_position/pose** where it contains the current position of the vehicle (figure 7).

For a test case before the path planning algorithm, a circular trajectory is selected. From the Simulink model C++ code package (figure 8) is generated and loaded into the ROS workspace. The ROS node is run and it is observed in the simulation that the vehicle is following the circular path. The baseline Simulink model has been setup.

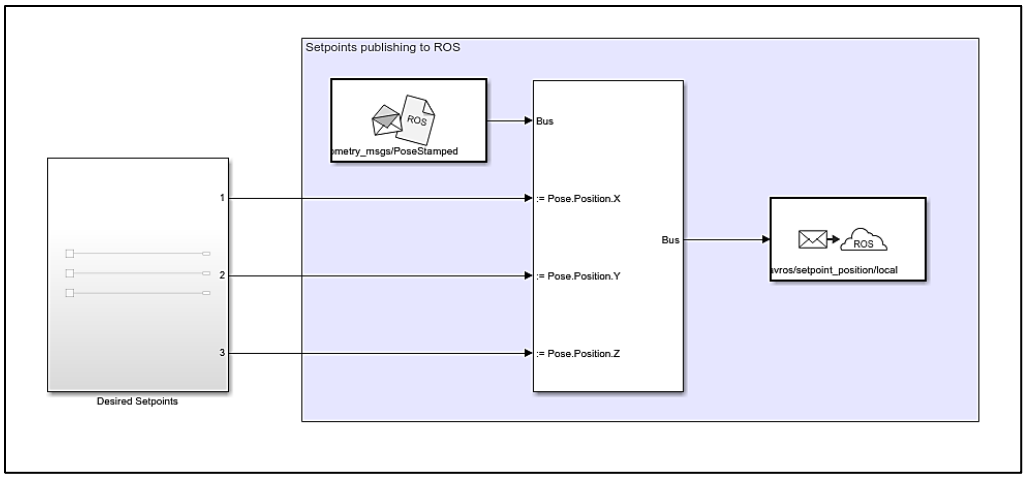


Figure 6 Simulink model section which publishes to companion computer

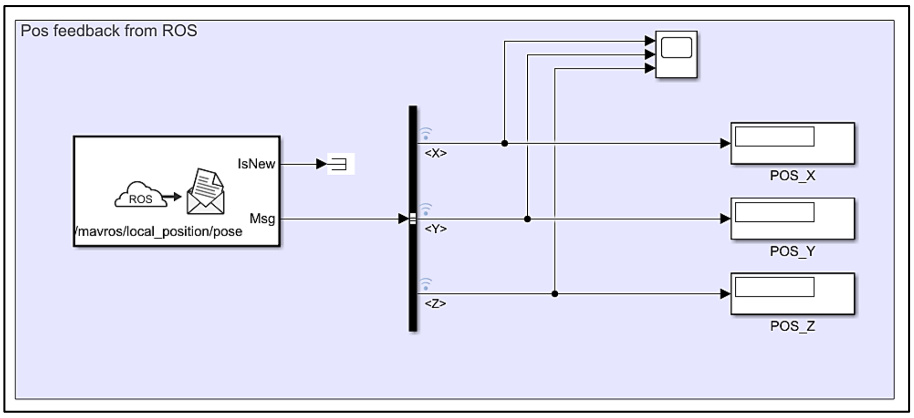
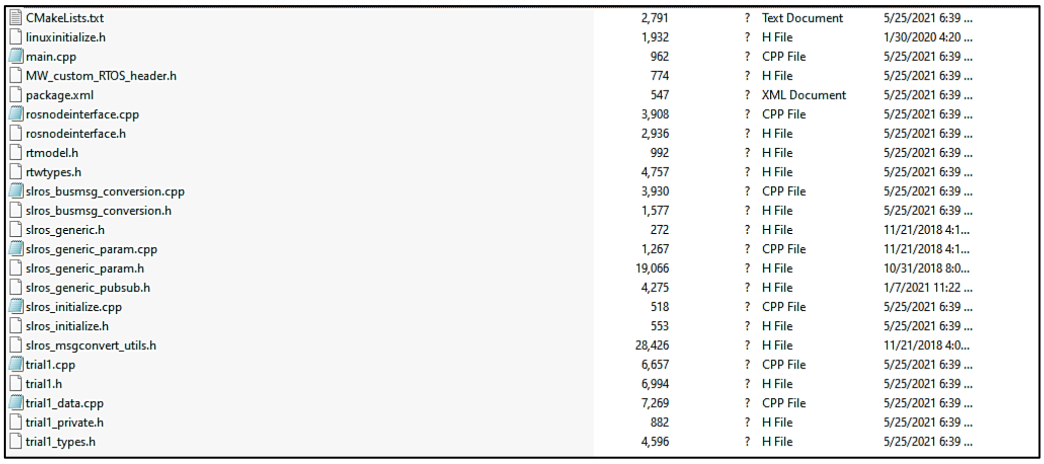


Figure 7 Simulink model section which subscribes from companion computer



CMakeLists.txt	2,791	?	Text Document	5/25/2021 6:39 ...
linuxinitialize.h	1,932	?	H File	1/30/2020 4:20 ...
main.cpp	962	?	CPP File	5/25/2021 6:39 ...
MW_custom_RTOS_header.h	774	?	H File	5/25/2021 6:39 ...
package.xml	547	?	XML Document	5/25/2021 6:39 ...
rosnodeinterface.cpp	3,908	?	CPP File	5/25/2021 6:39 ...
rosnodeinterface.h	2,936	?	H File	5/25/2021 6:39 ...
rtmodel.h	992	?	H File	5/25/2021 6:39 ...
rtwtypes.h	4,757	?	H File	5/25/2021 6:39 ...
slros_busmsg_conversion.cpp	3,930	?	CPP File	5/25/2021 6:39 ...
slros_busmsg_conversion.h	1,577	?	H File	5/25/2021 6:39 ...
slros_generic.h	272	?	H File	11/21/2018 4:1...
slros_generic_param.cpp	1,267	?	CPP File	11/21/2018 4:1...
slros_generic_param.h	19,066	?	H File	10/31/2018 8:0...
slros_generic_pubsub.h	4,275	?	H File	1/7/2021 11:22 ...
slros_initialize.cpp	518	?	CPP File	5/25/2021 6:39 ...
slros_initialize.h	553	?	H File	5/25/2021 6:39 ...
slros_msgconvert_utils.h	28,426	?	H File	11/21/2018 4:0...
trial1.cpp	6,657	?	CPP File	5/25/2021 6:39 ...
trial1.h	6,994	?	H File	5/25/2021 6:39 ...
trial1_data.cpp	7,269	?	CPP File	5/25/2021 6:39 ...
trial1_private.h	882	?	H File	5/25/2021 6:39 ...
trial1_types.h	4,596	?	H File	5/25/2021 6:39 ...

Figure 8 C++ code package for ROS node generated from Simulink model

3.4.1 Advantages of having Simulink model:

- The Simulink model enables to design the controller and tune the values for the desired dynamics of the vehicle.
- C++ code can be generated which can be run as a standalone node in ROS, hence hardware implementation of the developed model is easy.

3.4.2 Initial implementation of path planning and control architecture of the vehicle

The goal point is given as an input to the vehicle. The path planner uses the sensor measurements about the current position and obstacle information and generates set of waypoints for the vehicle to follow. And with a tuned controller the vehicle will reach the goal point without any collision with the obstacles. Figure 9 shows the block diagram for the same.

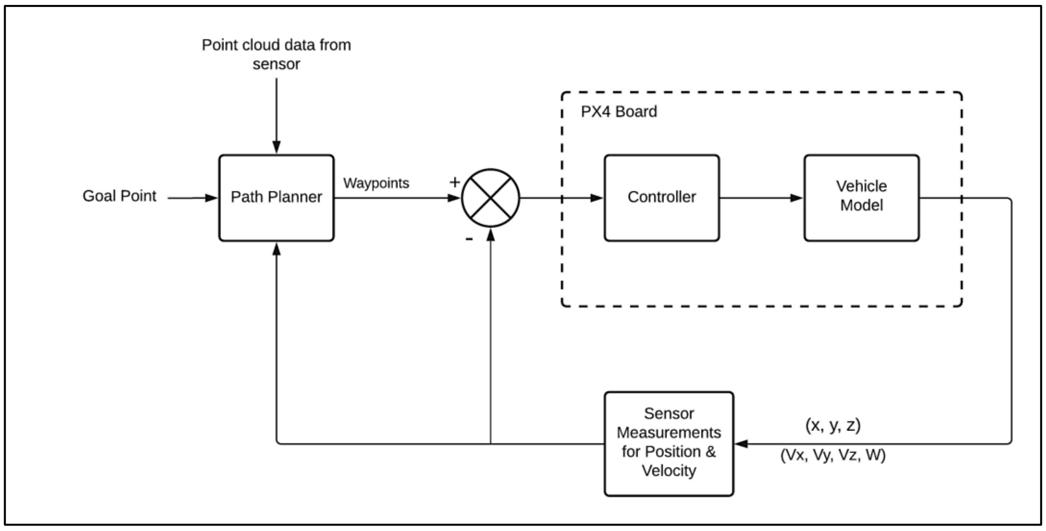


Figure 9 Block diagram of initial path planning and control architecture

The block diagram in figure 10 explains the architecture of the path planner. The point cloud data from the sensors contain information about the environment. The Octomap package in ROS uses this information to generate 3D occupancy grid of the environment. The search algorithm utilizes the grid information to search and generate safe path for the vehicle to move from the its current position to goal point.

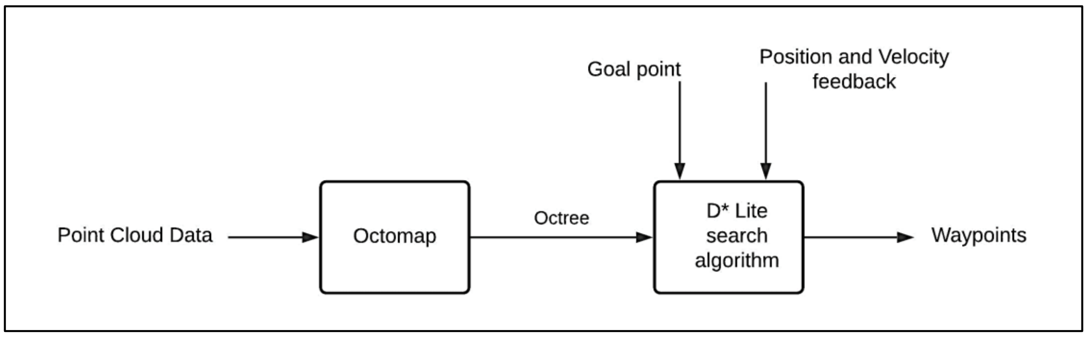


Figure 10 Block diagram of path planning module

3.5 D* Lite Search algorithm for 3D path planning

The D* Lite algorithm from the source [12] is developed for dynamic path planning of mobile robots in 2D map. In this project, the ability of the algorithm to search the path has been extended from 2D to 3D. For this semester, a static environment with two obstacles is considered which has a constant occupancy grid map. Currently, the path planning is focused on only air module, hence, the environment has two buildings as obstacles (see figure 11).

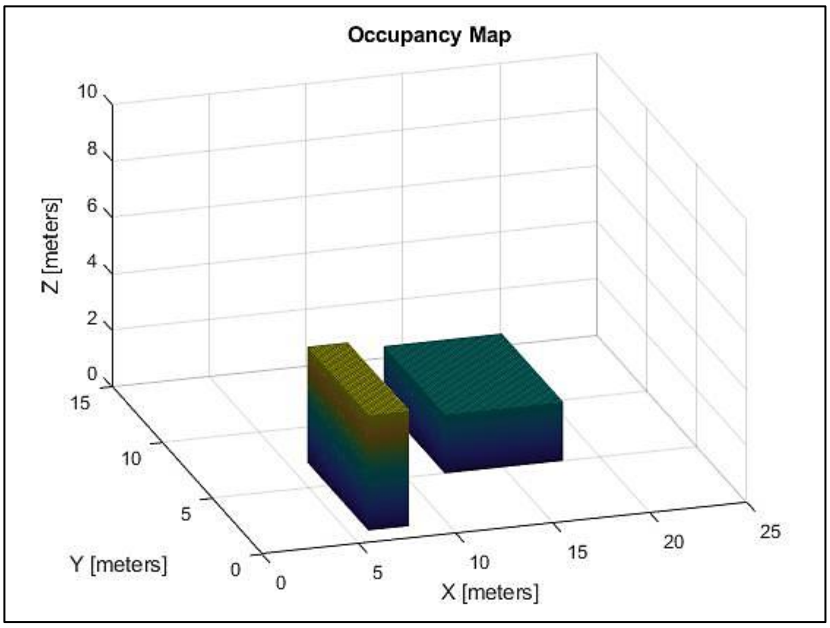
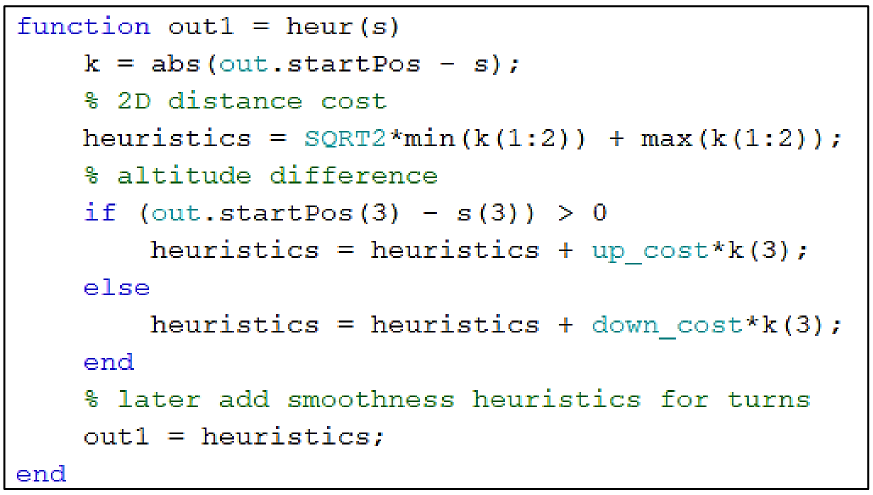


Figure 11 3D occupancy grid plot of the map

For version 1, along with the 2D searching additional constraint on z axis is introduced in the heuristic estimation. This constraint ensures that if the current position of the vehicle is in different plane than the goal point, then it moves to up or below node depending on the direction from heuristic. The corresponding heuristic function definition is shown in figure 12.



```
function out1 = heur(s)
    k = abs(out.startPos - s);
    % 2D distance cost
    heuristics = SQRT2*min(k(1:2)) + max(k(1:2));
    % altitude difference
    if (out.startPos(3) - s(3)) > 0
        heuristics = heuristics + up_cost*k(3);
    else
        heuristics = heuristics + down_cost*k(3);
    end
    % later add smoothness heuristics for turns
    out1 = heuristics;
end
```

Figure 12 Heuristic function for version 1 of D Lite algorithm*

For each node, there are 10 neighbor nodes. 8 neighbor nodes in the same plane and 1 in up direction and 1 in down direction. The distance cost to move to a neighbor in same plane is 1 (figure 13 (a)). The up_cost and down_cost are different as the energy consumed by the vehicle to move upwards is more. But both these costs are greater than 1 because while in motion changing the altitude of the vehicle requires more energy.

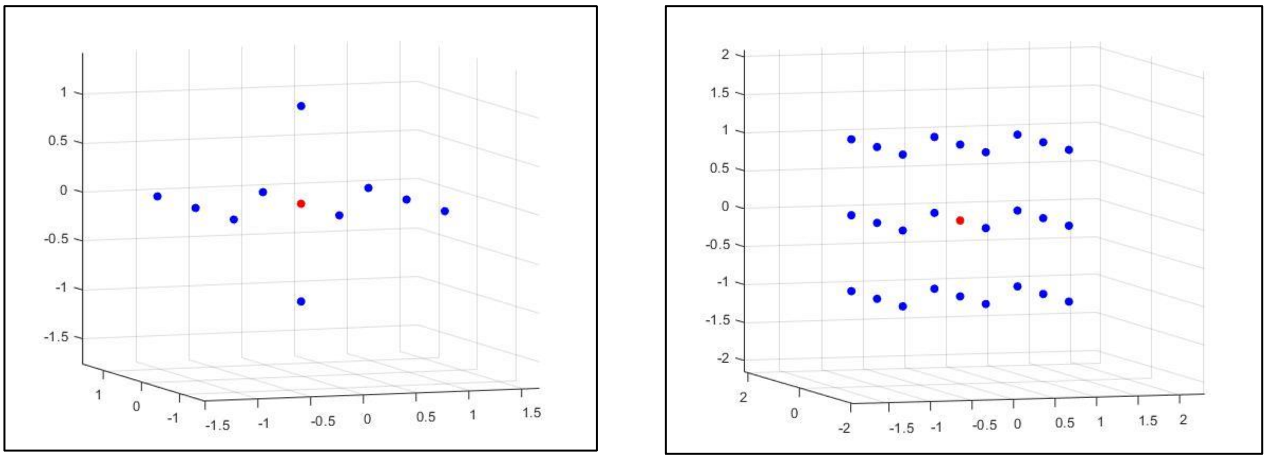
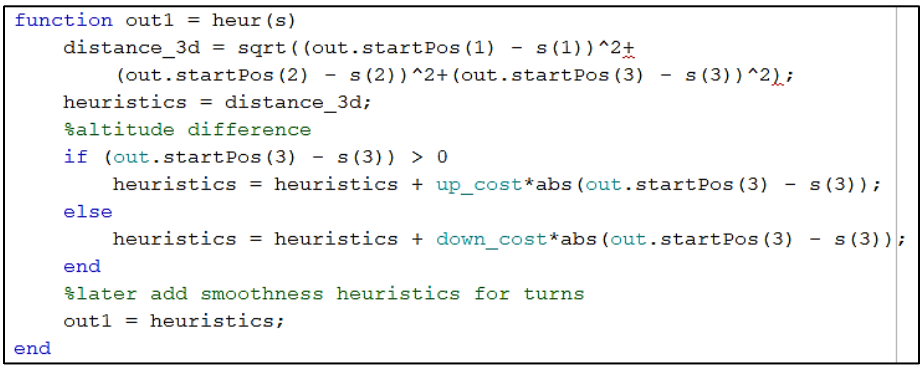


Figure 13 (a), (b) Neighbour nodes considered in version 1 and 2

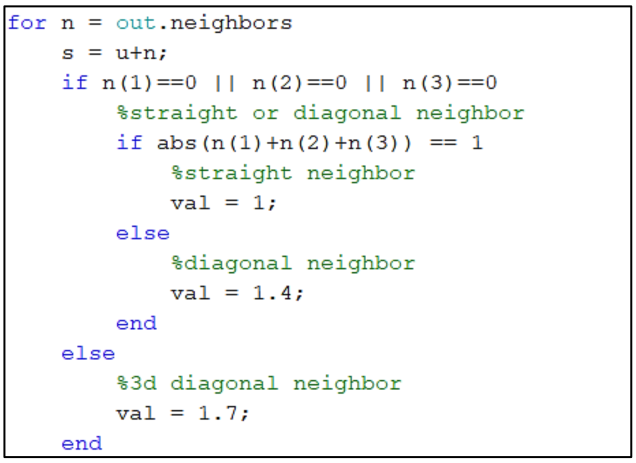
Later for version 2, the heuristic calculation is updated with 3D distance and more neighbor nodes. Total 26 neighbor nodes are introduced for every node. 8 neighbor nodes in the same plane. 9 neighbor nodes in the upper plane and 9 neighbor nodes in lower plane (figure 13 (b)), such that the path can be generated more efficiently. The updated heuristic function can be seen in figure 14.



```
function out1 = heur(s)
    distance_3d = sqrt((out.startPos(1) - s(1))^2+
        (out.startPos(2) - s(2))^2+(out.startPos(3) - s(3))^2);
    heuristics = distance_3d;
    %altitude difference
    if (out.startPos(3) - s(3)) > 0
        heuristics = heuristics + up_cost*abs(out.startPos(3) - s(3));
    else
        heuristics = heuristics + down_cost*abs(out.startPos(3) - s(3));
    end
    %later add smoothness heuristics for turns
    out1 = heuristics;
end
```

Figure 14 Heuristic function for version 2 of D Lite algorithm*

The distance from the current node to the neighbor is updated depending on the type of the neighbor. If the neighbor node is right next to, above, or below the current node, it is a straight neighbor and has distance cost of 1. If the neighbor node is 2D diagonal to the current node, then it is called diagonal neighbor and its cost is 1.4. If the neighbor node is 3D diagonal to the current node, then it is called 3d diagonal neighbor and its cost is 1.7 (figure l5). The complete algorithm is given in the appendices. The results obtained for different cases from the algorithm are discussed in section 4.



```
for n = out.neighbors
    s = u+n;
    if n(1)==0 || n(2)==0 || n(3)==0
        %straight or diagonal neighbor
        if abs(n(1)+n(2)+n(3)) == 1
            %straight neighbor
            val = 1;
        else
            %diagonal neighbor
            val = 1.4;
        end
    else
        %3d diagonal neighbor
        val = 1.7;
    end
```

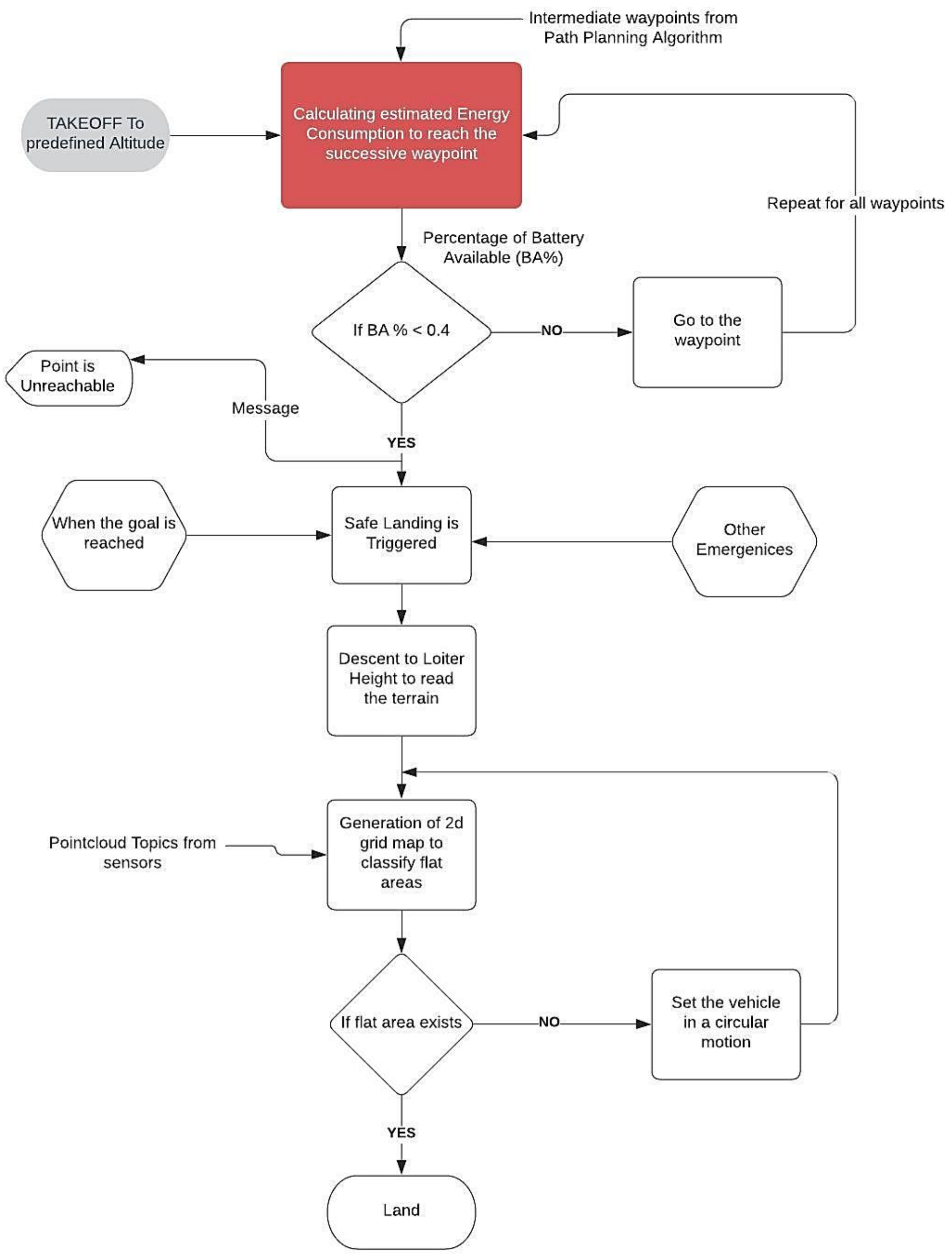
Figure 15 Neighbour node distance allocation in the algorithm

3.6 Safe Landing Algorithm

The safe landing algorithm is a part of the path-planning algorithm mainly focusing on the safety part of the vehicle. In this semester, the algorithm has been developed separately by specifying pre-defined waypoints. However, in the upcoming semester, the algorithm will be integrated with the path-planning module where the waypoints are generated by the latter and are sent to the safe landing algorithm.

Implementation of safe landing algorithm is crucial for the safety of the vehicle as it avoids crashes due to critically low battery and also protects the vehicle from getting damaged while landing due to rough or irregular terrains. The algorithm mainly performs two functions: the energy estimation to reach the next waypoint and the safe landing planner which detects flat areas while landing.

The flowchart which depicts the working of the algorithm is shown below,


Intermediate waypoints from Path Planning Algorithm
TAKEOFF To predefined Altitude
Calculating estimated Energy Consumption to reach the successive waypoint
Repeat for all waypoints
Percentage of Battery Available (BA%)
If BA % < 0.4
NO
Go to the waypoint
Point is Unreachable
Message
YES
When the goal is reached
Safe Landing is Triggered
Other Emergenices
Descent to Loiter Height to read the terrain
Pointcloud Topics from sensors
Generation of 2d grid map to classify flat areas
If flat area exists
NO
Set the vehicle in a circular motion
YES
Land

3.6.1 Energy Estimation

The first task of the safe landing algorithm once the vehicle has taken off to a specified altitude and waypoints from the path-planning algorithm are received is to estimate the energy in terms of battery capacity. This estimated energy is the energy that will be consumed by the vehicle to reach the next waypoint. Once this estimation is done, the algorithm will then check if the waypoint can be reached with the available battery. If there is sufficient amount of energy remaining, the vehicle will move towards the waypoint or else the safe landing planner will be triggered and the vehicle lands. The working of safe landing planner will be discussed in the next section.

The energy estimation is done based on the current consumption data and the time required to reach the next waypoint from the current position. The current consumption data will be provided by the power control module of the PX4-Autopilot. The time required to reach the waypoint is calculated using the ground speed of the vehicle and the distance to next waypoint. Ground speed is computed by the GPS module mounted on the vehicle and the distance to the waypoint is computed using the shortest distance formula,

$$Distance = \sqrt{((x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2)}$$

Where, (x_0, y_0, z_0) represent the current position of the vehicle and (x_1, y_1, z_1) is the location of the next waypoint.

The computed estimated energy to reach the waypoint is expressed in terms of battery percentage since the mavros node of the PX4-Autopilot publishes battery information in terms of percentage. The battery which we will be using on the vehicle for testing is a 3S 11.1V 3300 mAh LiPo battery with a C-rating of 40. In reality, only 80% of the battery is effective because, if the battery level falls below 20% the battery sustains a permanent damage. Once the energy is expressed in terms of battery percentage, a check is done if the available battery percentage published by the PX4 is higher than the required percentage to go to the next point. If there is enough energy to go to the point, the vehicle will follow the path computed by the path-planning algorithm. If the battery supply is less, then the safe landing planner will take the control over the vehicle and land it safely.

3.6.2 Safe Landing Planner

As explained in the previous section, the safe landing planner will be triggered if there is lack of battery supply to reach the waypoint. Not only in this situation the safe landing planner takes control over the vehicle, but also when the goal point is reached. The pilot can activate the planner manually if there is some erroneous data that is being monitored in real time, which corresponds to component or system failure that is being monitored in real time.

The safe landing planner detects flat terrains by analysing the pointcloud data from the sensors. The vehicle descends to a loiter height, which is 5m for this work. Once it reaches this altitude it studies the terrain underneath the vehicle via the pointcloud data. Pointcloud topics are nothing but the co-ordinates (x, y, z) of the surface being monitored by the ultrasound sensors. Once this information is received the planner generates a 2d grid of several bins using the x and y components of the pointcloud data. The generated 2d grid from the pointcloud data is shown in figure 16.

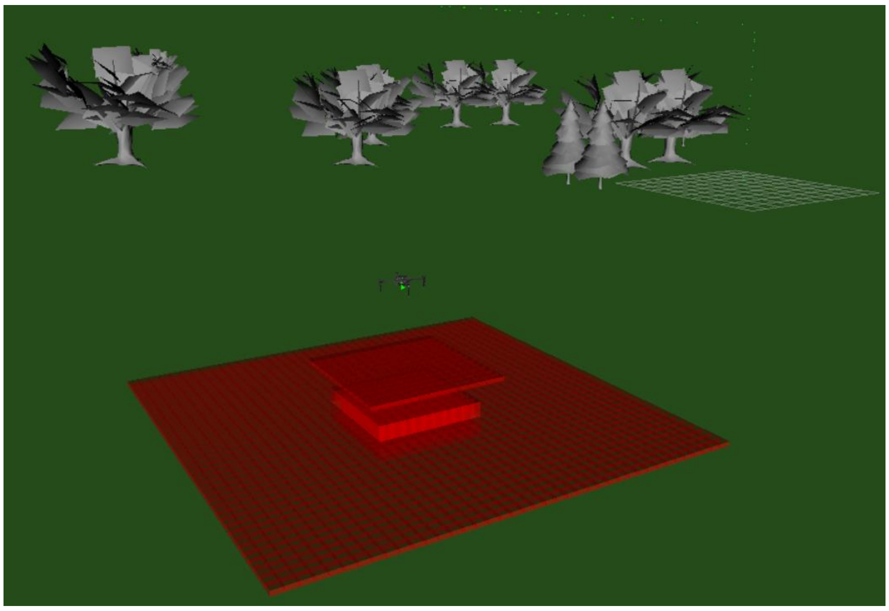


Figure 16 2D grid generation from the pointcloud data

For each of the bins present in the grid, the mean and standard deviation of the z component of the pointcloud data are computed. Standard deviation tells us how far the data is spread from the mean value. In this case, since we are computing the deviation and mean of the z values for each bin, it gives an estimation of whether the surface is regular or irregular. If the detected surface is flat, the vehicle will land safely. If an irregular terrain is detected, then the vehicle is set in an outward spiral movement to detect the neighbouring areas and find a suitable spot for landing.

3.6.3 Running the Simulation

To run the simulation, a launch file is generated which launches the px4 mavros, establishes connection to the gazebo simulation environment and also starts the *offb_node*. Once the mavros is launched, several mavros topics that are being published by the px4 and topics to which the node which contains the algorithm can subscribe are generated. Two nodes are publishing and subscribing to suitable mavros topics in this case: the *offb_node* and the *pos_teleop_node*.

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

PX4 has a timeout of 500ms between successive Offboard commands. If this timeout is exceeded, the autopilot will enter Failsafe mode, which lands the vehicle xy position. Hence, the publishing rate needs to be faster than 2 Hz, which already accounts for possible latencies. Before publishing anything, the *offb_node* waits for the connection to be established between MAVROS and the autopilot. Then, a set of setpoint commands are sent to the vehicle prior to engaging the Offboard mode. In the code, there is a switch to Offboard mode, after which the vehicle is armed and ready to fly. Once the vehicle is ready to fly, the *pos_teleop_node* can do its tasks. The communication between the two nodes and the mavros topics are shown in figure 17.

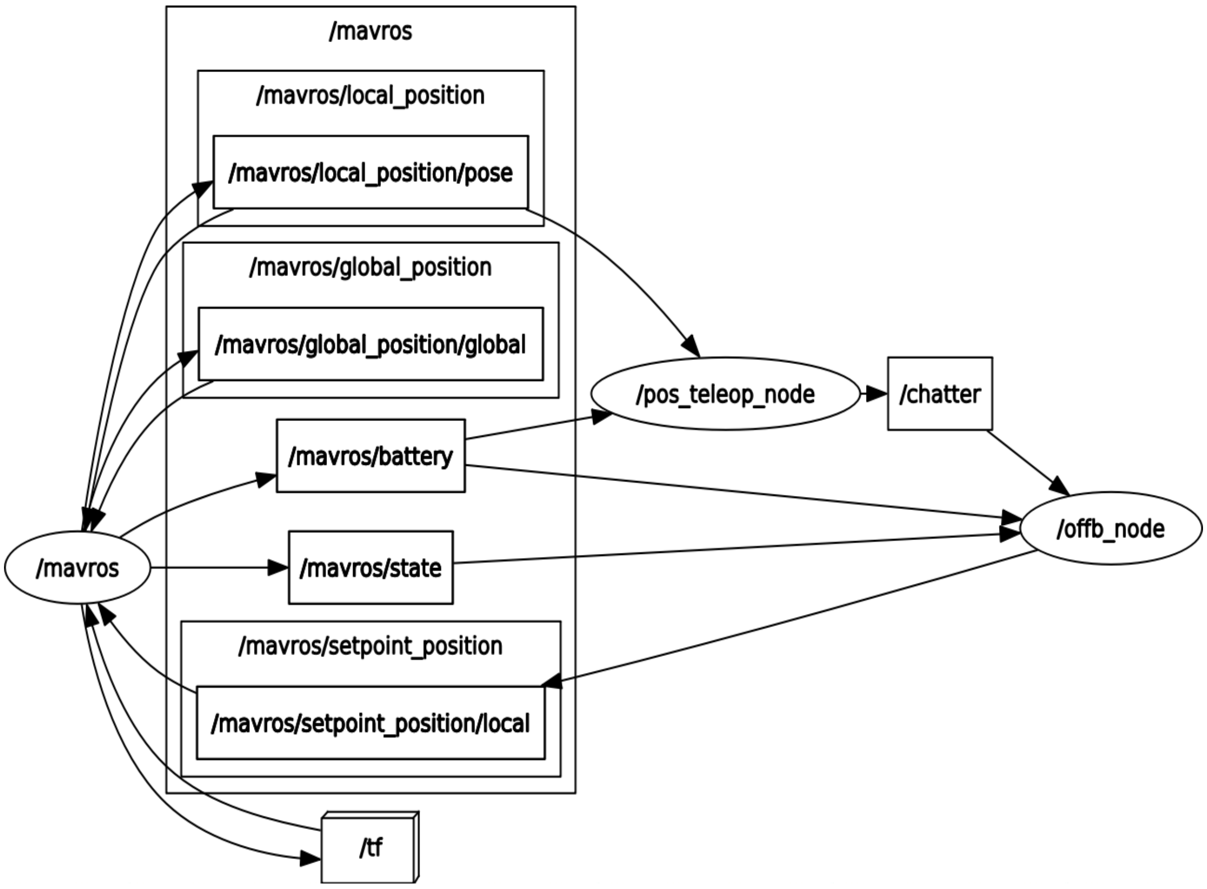


Figure 17 Communication between mavros and various nodes

It can be seen that, the */mavros* node is publishing several topics such as local position and global position of the vehicle, the battery information, the current state of the vehicle etc. The *offb_node* as stated previously subscribes to the */mavros/state* topic to get information about the current state of the vehicle. This means to say if the vehicle is in take-off, loiter, offboard or land mode.

The *pos_teleop_node* is the one which performs the main functions of the safe landing algorithm such as computing the distance to the next waypoint, energy required to reach the waypoint, decides if the point can be reached with available battery and activates the safe landing planner when it has decided to land. From the figure, it can be observed that /*pos_teleop_node* is subscribed to */mavros/battery* topic to receive information about the current state of the battery. It is also subscribed to the topic */mavros/local_position/pose* to know the current location of the vehicle.

The *pos_teleop_node* contains the waypoints that are generated by the path-planning algorithm and these waypoints are published into a custom topic called */chatter* whenever the algorithm decides to go to the waypoint. When the algorithm decides to land, the z co-ordinate of the member pose of the topic */mavros/local_position/pose* is set to zero and published into the topic */chatter*. The topic /chatter is in turn subscribed by the */offb_node* which publishes the information to the topic */mavros/setpoint_position/local*. This mavros topic changes the drone position by generating suitable motor commands, such that the vehicle goes to point being published (either the waypoint or land position).

The safe landing planner was simulated using QGC as it contains predefined maps. A mission was defined by specifying waypoints in QGC as shown in figure 18. Rviz which is another simulator compatible with ROS was used to observe the behaviour of the vehicle. Rviz is a 3d visualization tool for ROS applications. It provides a view of your robot model, capture sensor information from robot sensors, and replay captured data. It can display data from camera, lasers, from 3D and 2D devices including pictures and point clouds.

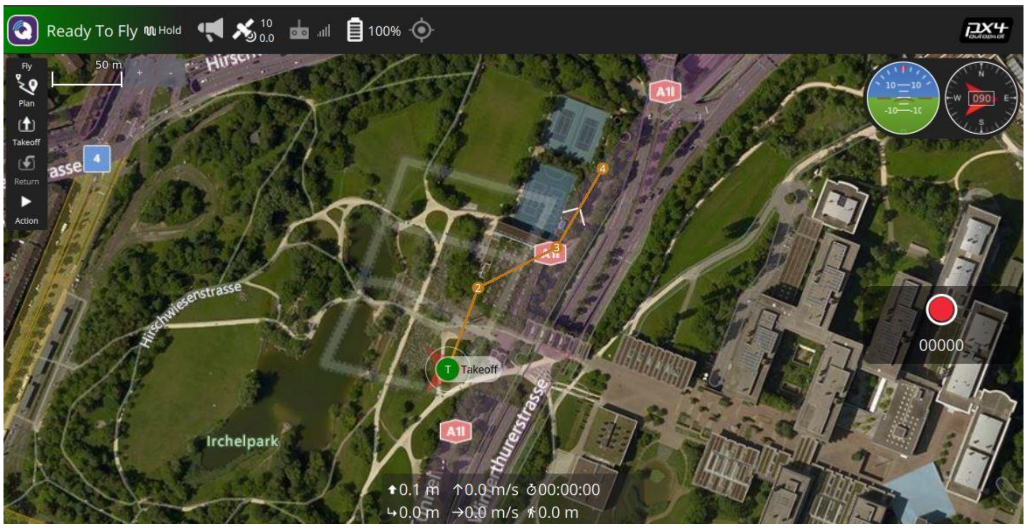


Figure 18 Mission definition in QGC

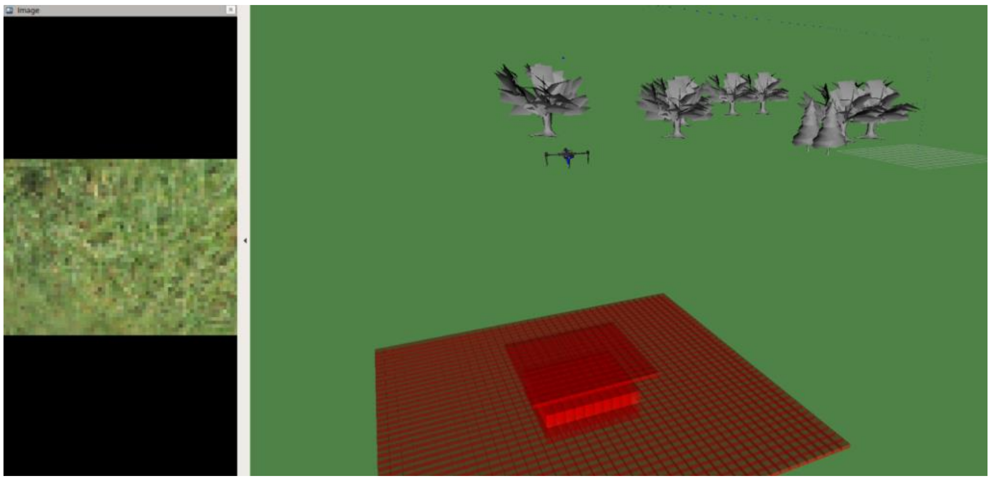


Figure 19 Real time display of ground in Rviz

The launch file of the safe landing algorithm establishes a connection between the ground station and Rviz, which allows Rviz to display images of terrain underneath the vehicle using the inbuilt depth camera sensor model as shown in figure 19. The window on the left is the real-time image of the ground under the vehicle. Since Rviz was running on a Virtual Machine, the resolution of the image is poor. However, the pointcloud data coming from the sensor isn't affected by this

4 Results and analysis

4.1 Comparison of search algorithms

Comparison study has been done for A*, Dijkstra, and D* Lite algorithms. 2D map with 2 simple obstacles has been considered. The starting point for the search is (0,0) and the goal point is (9,4) (see figure 20).

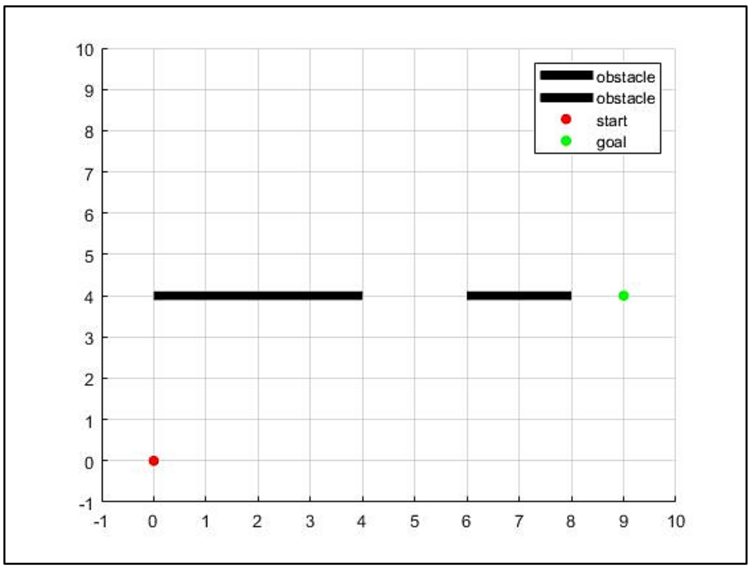


Figure 20 2D map with obstacles, start and goal point

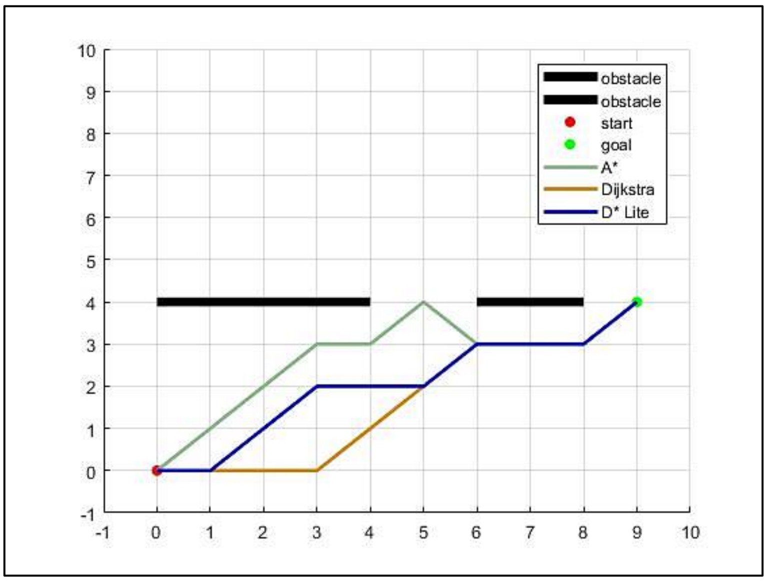


Figure 21 2D map showing paths generated from A, Dijkstra, and D* Lite algorithms*

In the figure 21, path generated by all the algorithms are observed. The distance between the start and end point is 9.85 meters. Table 1 shows the comparison of the length of the paths generated by each algorithm.

Table 1 Path length generated by each algorithm.

Algorithm	Path Length (m)
A*	12.9
Dijkstra	10.65
D* Lite	10.65

It is observed that A* algorithm has the longest path among the three algorithms. This is because of the heuristic estimation in the algorithm tries to reduce the gap between the current node and the goal node. Hence it starts moving in diagonal in the beginning. In between two obstacles, the current node is first reached according to the heuristic and then it encounters an obstacle, then it tries to avoid the obstacle and moves towards the goal. Since no other search branches present nearby, the path generated is longer. Dijkstra and D* Lite have same path length but the search criteria are different as seen in figure 21.

From the comparison study, it is clear that D* Lite algorithm is better suited for our application as it has ability to adapt to dynamic environment.

4.2 D* Lite algorithm in 3D path planning

The D* Lite algorithm was extended for 3D path planning. In version 1, two neighbor nodes, one up and one down, are added to the search criteria of the algorithm and the heuristics function is updated correspondingly. The starting point is (1,4,1) and the goal point is (20,7,2). The time taken by the algorithm to search the path is 0.54 second. In the figure 22, the generated path is observed. Since the heuristic is 2d distance between the nodes and moving up and down to manage the altitude, the generated path is longer than the desired path.

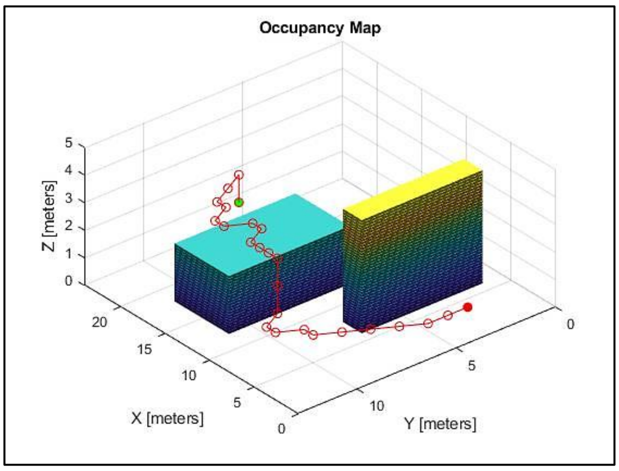


Figure 22 3D map showing path generated from version 1 of D Lite algorithm*

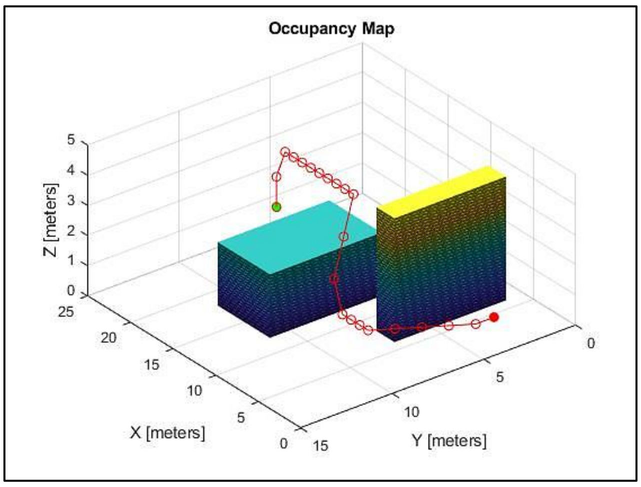


Figure 23 3D map showing path generated from version 2 of D Lite algorithm*

In version 2, The heuristic function is further updated to adapt to the 3D distance between the nodes and the neighbor nodes are updated such that every node surrounding a given node is searched for the finding the path. It is clear from the figure 23 that the path generated is much smoother and closer to the desired path. The algorithm takes 0.93 second to generate the path.

Different start and goal points are given and the path is generated. The new start point is (21,5,1) and the goal point is (1,9,3). Figure 24 shows the comparison of the paths generated by both versions of the algorithm. Table 2 gives the comparison of the time taken by each version to compute the paths.

Table 2 Time taken by each version to compute the paths

Version	Start Point	Goal Point	Time Taken (s)
Version 1 Version 2	(1,4,1) (1,4,1)	(20,7,2) (20,7,2)	0.54 0.93
Version 1 Version 2	(21,5,1) (21,5,1)	(1,9,3) (1,9,3)	0.4 1

In figure 24, it is observed that, the path generated by version 2 is longer than the version 1, but still is the best one. Because the vehicle requires less energy to follow path from version 2 than version 1. In path from version 1, when vehicle is near the goal point, it spends more energy to kill the momentum in x- and y-axes and then reduce the altitude to the goal point. But in the version 2, the vehicle kills the momentum after it reaches the goal point.

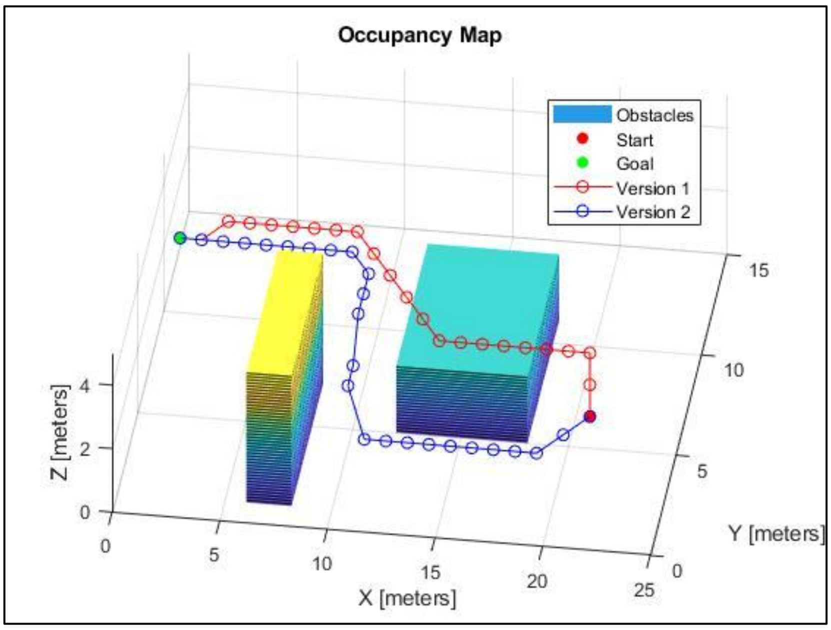


Figure 24 3D map showing path generated for different start and goal point

4.3 Safe Landing Algorithm

As previously mentioned in section 3.3, the safe landing algorithm has been developed separately in this semester without being integrated into the path planning algorithm. Therefore, the set of waypoints the vehicle must follow has been pre-defined in the algorithm. The results obtained using this algorithm is shown in figure 25. Figure 26 shows the battery consumption with respect to time. A condition was set in the algorithm such that, if the percentage of battery required to go a point exceeds the threshold (40% in this case), then the vehicle must land.

The simulated battery of the PX4 SITL models in Gazebo only deplete to 50% of its capacity by default, thus being implemented to never run out of energy. Therefore, in the *pos_teleop_node* a publisher was created which gives a battery percentage up to 40%. For simulation purpose, the discharge rate of battery was modified to be higher.

The information about the set of waypoints is given in Table 3.

Table 3 Waypoints information

Waypoints (x, y, z)	Battery Required in %	Battery Available in %	Decision
(0, 0, 5) → (0, 20, 5)	15%	100%	Go to the point
(0, 20, 5) → (10, 50, 8)	24%	85%	Go to the point
(10, 50, 8) → (10, 80, 10)	22.5 %	60 %	Go to the point
(10, 80, 10) → (20, 100, 15)	17 %	38 %	Land

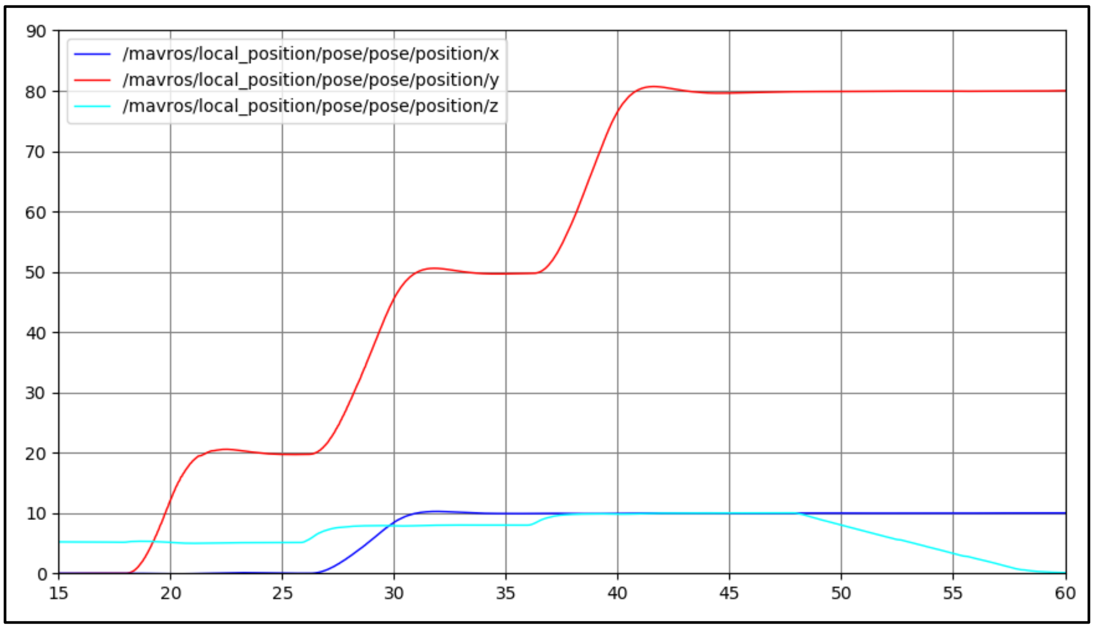


Figure 25 Position of the vehicle

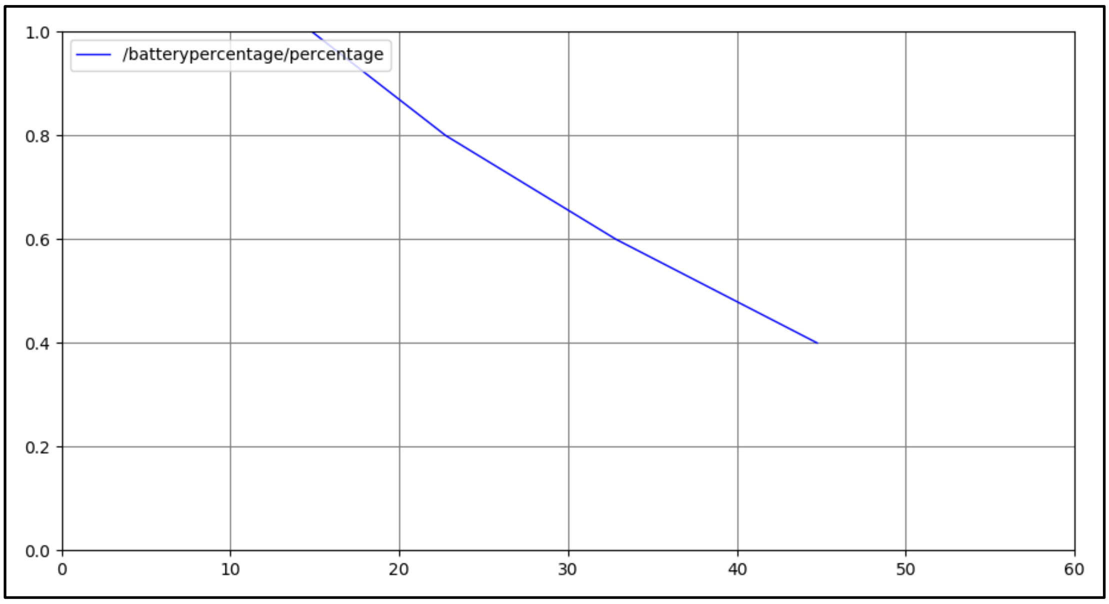


Figure 26 Variation of battery percentage with time

It can be seen from figure 25 and figure 26, when the simulation starts, the battery percentage is 1, which means it is fully charged. The waypoints are followed until the available battery percentage reached the 40% threshold. Around t=45 seconds, the battery falls below this threshold and the z-coordinate of the vehicle's position decreases until it becomes zero (see figure), in other words, the vehicles lands.

5 Conclusion and perspectives

The project was handed to us with the vehicle designed and assembled with working controllers to control both the aerial and ground modules. The baseline path-planning algorithm for air module and a complete avoidance algorithm for the ground module had been created. However, the path-planning algorithm for the air module with A* algorithm as the search algorithm is efficient for the static environments. Hence, investigation was done on various path-planning algorithms and a comparison study was carried out to determine which algorithm suits the best in an unknown and a dynamic environment. D* Lite algorithm was found to be the optimal choice as it took less time to generate path. The existing D* lite algorithm obtained from literature was in 2D and hence an extension of the same to 3D was done.

ROS was chosen as an offboard API due to the fact that it is largely used in the robotics community thus having a vast number of libraries which are necessary for implementing all the algorithms. This also gave us the ability to create nodes simulating the PX4 autopilot software and allowing to create real life missions of predefined waypoints in QGroundControl, which can then be given to the algorithms.

Furthermore, a safe landing algorithm which can compute the energy needed to reach the waypoints and detect suitable areas for landing was developed. The algorithm was validated using ROS and gazebo environment, it was found that when the energy required to reach a waypoint causes the battery level to fall below the threshold the vehicle lands. While landing, the safe landing planner was triggered and the validation of this planner was carried out in QGC and Rviz.

At the end of this semester, the vehicle has a path-planning algorithm for air module in a static environment with unknown obstacles. Safe landing algorithm has been developed separately by defining custom set of waypoints.

In the upcoming semester, the path-planning algorithm will be extended for dynamic environment. The safe landing algorithm will be integrated with the path-planning algorithm so that the waypoints generated from the latter are fed to the former and the former can then perform its functions. A master algorithm comprising of all the algorithms including the obstacle avoidance for the ground module will be generated. This algorithm will decide on whether to choose ground or air module depending on energy, power and time requirements. The algorithm will be validated in a simulated environment which resembles the real world before carrying out the real testing on the vehicle with actual sensors.

6 References

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

[2] R. Rodrigues, “Autonomous robotic aerial vehicle: S3 project report”, March 2021.

[3] B. Ismael, “Autonomous robotic aerial vehicle: S3 project report”, March 2021.

[4] N. Chow, G. Gugan and A. Haque, "RADR: Routing for Autonomous Drones," *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, Tangier, Morocco, 2019, pp. 1445-1450, doi: 10.1109/IWCMC.2019.8766530.

[5] S. Islam and A. Razi, "A Path Planning Algorithm for Collective Monitoring Using Autonomous Drones," *2019 53rd Annual Conference on Information Sciences and Systems (CISS)*, Baltimore, MD, USA, 2019, pp. 1-6, doi: 10.1109/CISS.2019.8693023.

[6] M. Lupascu, S. Hustiu, A. Burlacu and M. Kloetzer, "Path Planning for Autonomous Drones using 3D Rectangular Cuboid Decomposition," *2019 23rd International Conference on System Theory, Control and Computing (ICSTCC)*, Sinaia, Romania, 2019, pp. 119-124, doi: 10.1109/ICSTCC.2019.8886091.

[7] Elaf Jirjees Dhulkefl, & Akif Durdu. “Path Planning Algorithms for Unmanned Aerial Vehicles”. International Journal of Trend in Scientific Research and Development, (2019) 3(4),359–362. http://doi.org/10.31142/ijtsrd23696.

[8] Koenig S, Likhachev M. D* lite. Aaai/iaai. 2002 Jul 28;15.

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

[10] “PX4 Development | PX4User Guide”, Docs.px4.io [Online]. Available: https://docs.px4.io/master/en/development/development.html

[11] Zammit C, Van Kampen EJ. Comparison between A* and RRT algorithms for UAV path planning. In2018 AIAA guidance, navigation, and control conference 2018 (p. 1846).

[12] "LazyFalcon/D_star_PathPlanning", GitHub. [Online]. Available: https://github.com/LazyFalcon/D_star_PathPlanning.

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

[14] ZOMPAS A. Development of a three-dimensional path planner for aerial robotic workers (Master's thesis, University of Twente).

[15] Sahoo RR, Rakshit P, Haidar MT, Swarnalipi S, Balabantaray BK, Mohapatra S. Navigational path planning of multi-robot using honey bee mating optimization algorithm (HBMO). International Journal of Computer Applications. 2011 Aug;27(11):1-8.

[16] Jun M, D’Andrea R. Path planning for unmanned aerial vehicles in uncertain and adversarial environments. In Cooperative control: models, applications and algorithms 2003 (pp. 95-110). Springer, Boston, MA.

[17] Knispel L, Matousek R. A performance comparison of rapidly-exploring random tree and Dijkstra’s algorithm for holonomic robot path planning. Institute of Automation and Computer Science, Faculty of Mechanical Engineering, Brno University of Technology. 2013:154-62.

[18] Raheem FA, Hameed UI. Path planning algorithm using D* heuristic method based on PSO in dynamic environment. American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS). 2018 Dec 11;49(1):257-71.

[19] Kim J, Jo K, Kim D, Chu K, Sunwoo M. Behavior and path planning algorithm of autonomous vehicle A1 in structured environments. IFAC Proceedings Volumes. 2013 Jun 1;46(10):36-41.

[20] Minh VT, Pumwa J. Feasible path planning for autonomous vehicles. Mathematical Problems in Engineering. 2014 Jan 1;2014.

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

Appendices

D* Lite algorithm

```
function path = dstar_algorithm(start,goal)
    resolution = 10;
    up_cost = 2;
    down_cost = 1.5;
    kM = 0;
    Xlim = 24;
    Ylim = 13;
    Zlim = 7;
    field1 = 'map'; value1 = occupancyMap3D(resolution);
    field2 = 'startPos'; value2 = [start; 0; 0];
    field3 = 'endPos'; value3 = [goal; 0; 0];
    field4 = 'neighbors'; value4 = zeros(5,26);
    field5 = 'stack'; value5 = [];
    field6 = 'graph'; value6 = zeros(Xlim,Ylim,Zlim,3);
    field7 = 'path'; value7 = [];
    coder.varsize('value5');
    coder.varsize('value7');
    out = struct(field1,value1,field2,value2,field3,value3,
             field4,value4,field5,value5,field6,value6,field7,value7);

    Obs1 = [7 2 1;
            7 2 3;
            7 4 1;
            7 4 3;
            7 6 1;
            7 6 3;];
    Obs2 = [13 6 1;
            13 8 1;
            13 10 1;
            15 6 1;
            15 8 1;
            15 10 1;
            17 6 1;
            17 8 1;
            17 10 1;];
    updateOccupancy(out.map,Obs1,0.8);
    updateOccupancy(out.map,Obs2,0.8);
    inflate(out.map,1);
    out.neighbors = [
        0 1 0 0 0;
        -1 0 0 0 0;
        0 -1 0 0 0;
        1 1 0 0 0;
        -1 -1 0 0 0;
        1 0 0 0 0;
        1 -1 0 0 0;
        -1 1 0 0 0; %middle layer
        0 1 1 0 0;
        -1 0 1 0 0;
        0 -1 1 0 0;
        1 1 1 0 0;
        -1 -1 1 0 0;
        1 0 1 0 0;
        1 -1 1 0 0;
        -1 1 1 0 0; %upper layer
        0 1 -1 0 0;
    -1 0 -1 0 0;
    0 -1 -1 0 0;
    1 1 -1 0 0;
    -1 -1 -1 0 0;
    1 0 -1 0 0;
    1 -1 -1 0 0;
    -1 1 -1 0 0; %lower layer
    0 0 1 0 0;
    0 0 -1 0 0;
    ]';
out.graph(:,:,:,1:2) = inf;
out.graph(:,:,:,3) = false;
setRhs(out.endPos, 0);
setQ(out.endPos);
out.endPos(4:5) = [heur(out.endPos); 0];
out.stack(:,end+1) = out.endPos;

found = false;
computeShortestPath();
if found
    setg(out.startPos, rhs(out.startPos));
end

out.path = resolve();
path = out.path(1:3,:)';

function setQ(s)
    out.graph(s(1), s(2), s(3), 3) = true;
end
function rsetQ(s)
    out.graph(s(1), s(2), s(3), 3) = false;
end
function out1 = rhs(s)
    out1 = out.graph(s(1), s(2), s(3), 2);
end
function setRhs(s, val)
    out.graph(s(1), s(2), s(3), 2) = val;
end
function out1 = heur(s)
    distance_3d = sqrt((out.startPos(1) - s(1))^2+(out.startPos(2) -
   s(2))^2+(out.startPos(3) - s(3))^2);
    heuristics = distance_3d;
    %altitude difference
    if (out.startPos(3) - s(3)) > 0
        heuristics = heuristics + up_cost*abs(out.startPos(3) - s(3));
    else
        heuristics = heuristics + down_cost*abs(out.startPos(3) - s(3));
    end
    %later add smoothness heuristics for turns
    out1 = heuristics;
end
function out1 = g(s)
    out1 = out.graph(s(1), s(2), s(3), 1);
end
function setg(s, val)
    out.graph(s(1), s(2), s(3), 1) = val;
end
function out1 = calculateKey(s)
    out1 =  [ min( g(s), rhs(s) ) + heur(s) + kM;
              min( g(s), rhs(s) )];
end
function out1 = testNode(s) % true if available
    out1 = true;
    for n = out.neighbors
        pos = s + n;
        if getOccupancy(out.map,pos(1:3)') > 0.7
            out1 = false;
            return
        end
    end
end
function out1 = cmp(s1, s2)%compare keys s1 > s2
    out1 = s1(1) < s2(1) || ( s1(1) ==s2(1) && s1(2) < s2(2));
end
function updatevertex(u)
            if g(u) ~= rhs(u)
                    u(4:5) = calculateKey(u);
                    out.stack(:,end+1) = u;
                    setQ(u);
            else
                    rsetQ(u);
            end
end
function out1 = sortStack(stack)
    sortedstack = sortrows(stack',4,'descend');
    out1 = sortedstack';
end
function computeShortestPath()
    terminate = false;
    count = 0;
    limit = inf;
    while ~terminate && ~isempty(out.stack)

        count = count +1;
        if count >= limit
            break
        end
        out.stack = sortStack(out.stack);
        u = out.stack(:,end);
        out.stack(:,end) = [];

       if ~(  cmp(  u(4:5), calculateKey(out.startPos))  || rhs(out.startPos) ~=
   g(out.startPos) )
            found = true;
            break
        end

        if isequal(u(1:3), out.startPos(1:3))
            found = true;
        end
        rsetQ(u);
        gu = g(u);
        ru = rhs(u);
        if gu > ru
            setg(u, ru);
            gu = ru;
            for n = out.neighbors
                s = u+n;
                if n(1)==0 || n(2)==0 || n(3)==0
                    %straight or diagonal neighbor
                    if abs(n(1)+n(2)+n(3)) == 1
                        %straight neighbor
                        val = 1;
                    else
                        %diagonal neighbor
                        val = 1.4;
                    end
                else
                    %3d diagonal neighbor
                    val = 1.7;
                end
                if s(1)> Xlim || s(2) > Ylim || s(3) > Zlim || s(1) == 0 || s(2) == 0
          || s(3) == 0
                    continue
                else
                    if rhs(s) == inf % unvisited
                        if testNode(s)
```

```
                                setRhs(s,  val + gu);
                                updatevertex(s);
                            else
                                setg(s, -1);
                                setRhs(s, -1);
                            end
                        else % visited
                            if rhs(s) > val+gu
                                setRhs(s,  val + gu);
                                updatevertex(s);
                            end
                        end
                    end
                end
            end
        end
    end
    function out1 = resolve()
        s = out.startPos;
        coder.varsize('out1');
        out1 = s;
        out.graph(s(1), s(2), s(3), 1) = inf;
        while ~isequal(s(1:3), out.endPos(1:3))
            minval = inf;
            it = [];
            for n = out.neighbors
                u = s+n;
                if u(1)> Xlim || u(2) > Ylim || u(3) > Zlim || u(1) == 0 || u(2) == 0 ||
      u(3) == 0
                    continue
                else
                    uval = g(u);
                    if uval < minval && ~isinf(uval) && uval > -1
                        minval = uval;
                        it = n;
                    end
                end
            end
            s = s + it;
            out.graph(s(1), s(2), s(3), 1) = inf;
            out1 = [out1, s];
        end
    end
end
```