T.C.
ISTANBUL OKAN UNIVERSITY
INSTITUTE OF GRADUATE SCIENCES

THESIS

FOR THE DEGREE OF

MASTER OF SCIENCE

IN AUTOMOTIVE MECHATRONICS AND

INTELLIGENT VEHICLES PROGRAM

Abdalla Ahmed Roshdi MOHAMED

(203005011)

DEVELOPMENT OF AN QUADCOPTER PLANNING ALGORITHM

THESIS ADVISOR

Assoc. Prof. Dr. Ömer Cihan KIVANÇ

ISTANBUL, January 2023

T.C.
ISTANBUL OKAN UNIVERSITY
INSTITUTE OF GRADUATE SCIENCES

THESIS FOR THE DEGREE OF MASTER OF SCIENCE IN AUTOMOTIVE MECHATRONICS AND INTELLIGENT VEHICLES PROGRAM

Abdalla Ahmed Roshdi MOHAMED
(203005011)

DEVELOPMENT OF AN QUADCOPTER PLANNING ALGORITHM

Date Thesis Delivered to Institute: January 13, 2023
Date of Thesis Defense: January 23, 2023
Thesis Advisor: Assoc. Prof. Dr. Ömer Cihan KIVANÇ

Jury Members: Prof. Dr. Ramazan Nejat TUNCAY

Assoc. Prof. Dr. Salih Barış ÖZTÜRK

ISTANBUL, January 2023

ABSTRACT

DEVELOPMENT OF AN QUADCOPTER PLANNING ALGORITHM

A quadcopter is one type of several unmanned vehicle systems (UAV), which means it can be operated without a human operator. The environment surrounding the drone can have a lot of obstacles which for sure will create additional difficulties in navigating and planning a path for the drone. The drone will be required to estimate where an obstacle is, and it has to manoeuvre accordingly to avoid crashing into it. This mentioned problem requires a very important feature: Obstacle avoidance. The aim here is for the drone to first detect any objects in its surroundings and especially on its path. Then it must avoid hitting it by flying around it or for example flying above it. For this task to be done successfully a lot of complex tasks need to work altogether. A mathematical model should be made for the drone to correctly explain the drone dynamics. This ensures that when the drone is given a new trajectory set of points, they can be translated into some motor voltages that can move the drone to those points accurately. This thesis will mainly compare a widely used planning algorithm with the introduced approach of using the *D** lite algorithm. This search algorithm has several advantages over the recent algorithms used for path planning. It can calculate a new path faster while also avoiding newly detected obstacles. Also, the mathematical model, path planning, and path-following algorithms will be explained further in more detail.

Keywords: *A** Algorithm, Planning Algorithms, *D** Lite, Quadcopter, Drone

TABLE OF CONTENTS

LIST OF FIGURES VI

I. INTRODUCTION 1

II. LITERATURE REVIEW 3

2.1. QUADCOPTER DESIGN 3
2.2. QUADCOPTER DYNAMICS 4
2.3. QUADCOPTER FORCES AND CONTROL 5
2.4. SENSORS FOR NAVIGATION 10

III. PLANNING 14

4.1. REQUIREMENTS TO FIND A PLAN 14
4.2. *A** ALGORITHM 16
4.4. D* LITE ALGORITHM 20

IV. SIMPLIFYING THE PATH 25

5.1. LESS WAYPOINTS 25
5.2. GRIDS AND GRAPHS 27
5.3. DEAD-BANDS 29

V. 2D TO 3D REPRESENTATION 31

6.1. THREE DIMENSIONAL GRIDS 31
6.2. 2.5D MAPS 32
6.3. GRAPHS IN THREE-DIMENSIONAL 32
6.4. RANDOM SAMPLING IN CODE 33

VI. CONTROLLER DESIGN RESULTS 39

7.1. SOME RESULTS FOR THE DRONE CONTROL 39

VII. KALMAN FILTER FOR SENSOR FUSION 44

VIII. SIMULATION RESULTS 46

IX. CONCLUSION 55

REFERENCES 56

LIST OF FIGURES

Figure II.1. A Quadcopter's Main Design. 4

Figure II.2. Quadcopter's Body Frame. 4

Figure II.3. Lift Force. 5

Figure II.4. Navigation Task. 10

Figure II.5. Simulation Interface 11

Figure II.6. Drone Takeoff 12

Figure III.1. Motion Planning State Machine 14

Figure III.2. Heuristic Concept. 16

Figure III.3. A simple grid to represent a simple map. 22

Figure III.4. Adding obstacles to the map 22

Figure IV.1. Original path 25

Figure IV.2. Applying collinearity 25

Figure IV.3. Path using diagonals 27

Figure IV.4. Effect of grid size on search algorithms 29

Figure V.1. 2.5D map representation 32

Figure V.2. Result of random sampling 37

Figure V.3. Graph of possible paths 38

Figure VI.1. Three drones reaching a goal point (1) 40

Figure VI.2. Three drones reaching a goal point (2) 40

Figure VI.3. Three drones reaching a goal point adding disturbance (1) 41

Figure VI.4. Three drones reaching a goal point adding disturbance (2) 41

Figure VI.5. Trajectory following 42

Figure VI.6. Drone overshoot 42

Figure VI.7. Drone following the path 43

Figure VIII.1. Drone and its environment 46

Figure VIII.2. Velocity in roll and pitch 47

Figure VIII.3. Altitude of the drone 48

Figure VIII.4. Position of the drone 48

Figure VIII.5. Path visualization and waypoints information 49

Figure VIII.6. The velocity of the drone in roll and pitch 49

Figure VIII.7. The amplitude of the drone for longer path 50

Figure VIII.8. The position of the drone for longer path 51

Figure VIII.9. Target point and drone's real position 51

Figure VIII.10. The error between target and real drone's position using D* lite 52

Figure VIII.11. The error between target and real drone's positon using A* 53

Figure VIII.12. Effect of obstacles discovered on a A* and D* lite 54

I. INTRODUCTION

In a quadcopter is one type of several unmanned vehicle systems (UAV), which means it can be operated without a human operator, which in turn saves labor. This technology got much focus, and a lot of research are made on this topic in a recent couple of years. That's because they can be used in several fields and applications, for example in the military and in civilian applications like farming or transporting medical products. An important advantage of these UAVs is their lightweight and cost-effectiveness. Also, they can be used in hazardous conditions and areas where humans can't be present. To have an autonomous vehicle means that the drone should do the tasks it is designed for without any further input from a human operator. For that, path planning and trajectory-following algorithms are also advanced issues that need to be considered as well. There are many algorithms and many control methods that have been implemented in the last decades, however, there are still further improvements that can be achieved to realize more innovative and smart drone algorithms.

For the drone to fly autonomously, it needs to gather information from its surroundings. Considering the type of environment in which the drone will fly, a depth camera wouldn't be able to achieve the task, since the drone's environment will have fluctuations in light exposure. A better sensor that can be used in this situation is a LIDAR sensor. Other sensors will also be used for localization and path-following tasks. This includes GPS and IMU sensors, which can be used together to achieve more accurate results. The concept of merging sensors to yield better results makes use of the extended Kalman filter (EKF). The EKF makes use of redundant measurements to estimate the correct measurement. No sensor can provide the correct measurement,

especially with changing conditions. Therefore, more sensors are used to measure the same thing and with the use of EKF, a more accurate measurement can be estimated.

The environment surrounding the drone can have a lot of obstacles which for sure will create additional difficulties in navigating and planning a path for the drone. The drone will be required to estimate where an obstacle is, and it has to maneuver accordingly to avoid crashing into it. This mentioned problem requires a very important feature: Obstacle avoidance. The aim here is for the drone to first detect any objects in its surroundings and especially on its path. Then it must avoid hitting it by flying around it or for example flying above it. For this task to be done successfully a lot of complex tasks need to work altogether. A mathematical model should be made for the drone that can explain the drone dynamics correctly. This ensures that when the drone is given a new trajectory set of points, they can be translated into some motor voltages that can move the drone to those points accurately. This thesis will mainly compare some planning algorithms implemented recently with the introduced approach of using the *D** lite algorithm. This search algorithm has several advantages over the recent algorithms used for path planning. It can calculate a new path faster while also avoiding newly detected obstacles. Also, the mathematical model, path planning, and path following algorithms will be explained further in more detail.

II. LITERATURE REVIEW

It is important to understand the various components and mini tasks associated with achieving the main target of path planning and following. As mentioned in the Chapter I, the task can be achieved if there is a accurate mathematical model that can describe the drone dynamics accurately. Also, it is mentioned that several sensors will be used. In this section, the building blocks and the several different components needed to achieve the target is discussed.

2.1. Quadcopter design

In a quadcopter, four rotors need to be considered for controlling it and doing all the tasks required from it. These rotors are placed equidistant from each other. To simplify the understanding of such a vehicle, a helicopter can be thought of first. In a helicopter there is just one large propeller in the top that spins fast enough to push the air downwards. This will lift the helicopter up to start flying. Also, a helicopter will have another propeller in its back. This propeller's only task is to cancel out the moment caused by the rotation of the main propeller in the top. If the propeller behind is not added, the helicopter's body will start spinning around itself and it will be unstable. Looking at the quadcopter again now, spinning each two propellers in opposite directions has the same effect of the propeller in the helicopter's back. They will help keeping the drone stable when they rotate.

Figure II.1. A quadcopter's main design [20].

It's clearly harder to control the drone since they have four rotors. For that, electronic assistance is required, for example sensors, controllers and other electronic components will be used.

2.2. Quadcopter dynamics

For controlling the drone, some aspects need to be considered first. The frames in which the drone will operate needs to be defined for example. The gravity will be pointing in the negative *z*-direction.

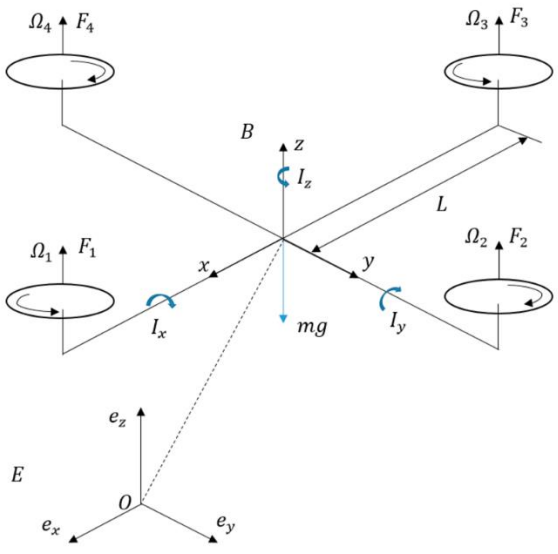

Figure II.2. Quadcopter's body frame [26].

Having the frame defined, it has the different axes in which the drone can rotate. As mentioned before, every two propellers rotate in the opposite direction to the other two propellers. So, two propellers rotate in a specific direction to keep the drone balanced around the *x*-axis. While the other two propellers keep the drone balanced around the *y*-axis. These four propeller rotations in such a way will help in keeping the drone from rotating around its *z*-axis. Changing the drone's position is related to how fast the motors are spinning. So, a change in their speeds will cause a change in the position as well. To be able to define the drone's position and orientation, three-axes will be defined as well, namely, the roll, the pitch, and the yaw.

2.3. Quadcopter forces and control

In order to understand the control and use a suitable controller for the quadcopter, it's important to first understand the different forces acting on the system. One of the forces is the weight, which is simply the mass of the vehicle times its acceleration due to gravity. Then there is the thrust, that's the force caused or generated by the action of the propellers.

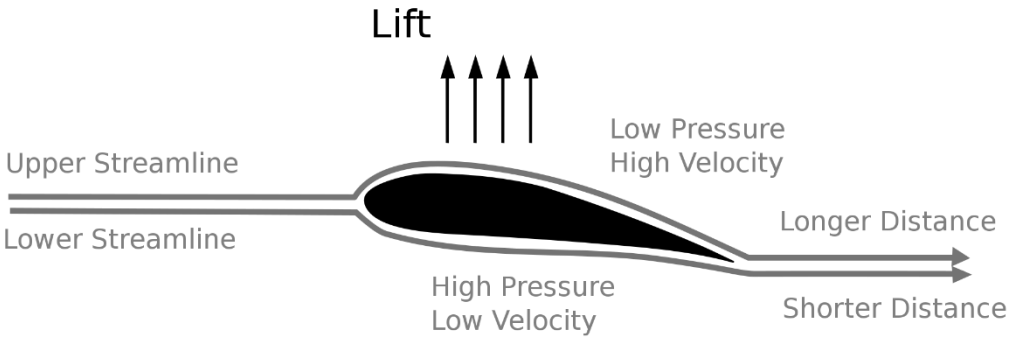

Figure II.3. Lift force [7].

One other force acting on the drone is the lift (L). An important point to keep in mind here when talking about the lift is the dynamic pressure. Dynamic pressure is the pressure that is caused by the motion of the drone's propellers through the air in this case. The movement and design of the propellers are made in a way so that air moving above the propellers will have lower pressure than air flowing below the propellers. Lower pressure means less force. From that effect, there will be lower pressure above the propeller than below it. This difference in pressure means there will be a net force pointing upwards on the wing producing lift that pushes the propellers upwards. First, it is important to know how to define the frames and how to relate them to one another. There is the inertial frame of reference and the body frame of reference. They can be related by introducing a rotation matrix R. If the inertial frame is represented by x, y, and z, while the body frame of reference is represented by a, b, and c. Then they can be related as follows,

$$\vec{a} = R\vec{x};\ \vec{b} = R\vec{y};\ \vec{c} = R\vec{z} \tag{II.1}$$

where R can be written in relation to the roll, pitch, and yaw angles as,

$$R = \begin{matrix} \cos\psi\cos\theta - \sin\phi\sin\psi\sin\theta & -\cos\phi\sin\psi & \cos\psi\sin\theta + \cos\theta\sin\phi\sin\psi \\ \cos\theta\sin\psi + \cos\psi\sin\phi\sin\theta & \cos\phi\cos\psi & \sin\psi\sin\theta - \cos\psi\cos\theta\sin\phi \\ -\cos\phi\sin\theta & \sin\phi & \cos\phi\cos\theta \end{matrix} \tag{II.2}$$

Calculating this lift force is dependent on three terms: the wing surface area (S), the dynamic pressure (q), and the coefficient of lift (CL). To calculate the dynamic pressure, the equation can be seen below,

$$L = Cl * q * S \tag{II.3}$$

The next force to be discussed is the drag force. This force is always acting parallel to the direction of the airflow relative to the quadcopter. This means that this force

resists the quadcopter's motion. Calculating the drag force (D) is dependent on other three terms: the drag coefficient, the dynamic pressure, and the surface area of the wing. From these terms, two of them are discussed before. The third term which is the drag coefficient depends on two coefficients; the parasitic drag and another term which contains the coefficient (C_d) of lift.

$$D = C_d * q * S \tag{II.4}$$

It's worth mentioning that all mathematical equations and models that are represented and will be represented later are just not perfect models. Instead, they are always designed in a way to simplify calculations but still, it must define reality as accurately as possible, or else they will be not useful. To start with modeling the drone dynamics, twelve states will be identified. The x, y, and z are variables specifying the location of the drone. Then there are ϕ, θ, and Ψ to specify the orientations of the drone and it's in Euler angles representation so they are the roll, pitch, and yaw. Also, the velocities and angular velocities of the drone will make twelve variables in total. The velocities include linear velocities in x, y, and z. The angular velocities are labeled as p, q, and r. It's worth mentioning that p, q, and r are not equal to the derivatives of ϕ, θ, and Ψ. Instead, p, q, and r are angular velocities around the x, y, and z axes and are named the body rates. These variables can be measured by a gyroscope that will be mounted on the drone. Now, the twelve states are known. The next step is to identify the forces and moments that will affect the drone. The thrust force first will be formed from the rotation of the four rotors of the drone. Collectively they can be summed up into one thrust force acting in the negative z direction since the positive z direction is defined as pointing down. So, it is required to calculate the translational motion

generated from the forces. Then the rotational motion generated by the moments acting on the drone must also be calculated. An important equation that would be used is the equation of moment. It simply says that a moment would cause a rotational acceleration about the axis in which the moment is acting. Keep in mind that in the case of three-dimensional representation, the moment will be a vector of length three and the rotational acceleration will also be a length three vector. The equation can be seen in Equation (II.5).

$$M = I * \dot{w} \tag{II.5}$$

The term I is the inertial matrix, and it will be a 3x3 matrix. In the case of a symmetrical system design the I matrix would look like this form shown in Equation (II.6).

$$I = \begin{matrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{matrix} \tag{II.6}$$

This equation is valid and works only when dealing in the world frame. However, all the moments and accelerations are defined in the body frame. Even the gyroscopes measurements are relative to the body frames. This means that this equation alone can't be used, and more complex mathematical terms need to be added. The equation after adding the terms,

$$M = (I * \dot{w}) + w \times (Iw) \tag{II.7}$$

Using the moment equation in x, y, and z directions it is possible now to calculate omega dot, from which it is possible to calculate and update nine of the twelve states by doing simple integrations. The only terms left are ϕ, θ and Ψ. To calculate them a rotation matrix will be needed for that. This rotation matrix will define how to

transform from the world frame to the body frame. The equation can be seen in Equation (II.8).

$$\begin{matrix}\dot{\phi} \\ \dot{\theta} = \\ \dot{\Psi}\end{matrix}\ \begin{matrix}1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)/\cos(\theta) & \cos(\phi)/\cos(\theta)\end{matrix}\ \begin{matrix}p \\ * \ q \\ r\end{matrix} \tag{II.8}$$

The 3x3 matrix in the middle is the rotation matrix that transforms the Euler angles in the body frames to the angular velocities in the world frames. At this point, the mathematical models have been derived. The next step is to discuss the controllers. The controllers used for this drone will mainly be PID controllers. In a PID controller, P is a proportional term that emphasizes the error terms more. And I term is an integral term, it's good to use the integral terms in the outermost loops where instability would not cause serious problems. An integrator would ensure that the system would eventually reach the target, or in other words, it will ensure that the error term will reach zero. Finally, there is a derivative term D. This term would help in decreasing the overshoot as well as decreasing the oscillations in the system. It's good to add a derivative term in a system that has inertia but low damping. There are three parts in the system that needs to be controlled. The altitude, the lateral position, and the yaw angle. The z trajectory will be handled by the altitude controller. The altitude controller's goal is to set the collective thrust to reach the required altitude. Then there are the x and y trajectories, they will first be handled by the level position controller. The yaw trajectory will be handled by the yaw controller. The altitude controller is fed by the reference z and the measured and estimated z. The output from the altitude controller will be fed also to the roll-pitch controller. That is because the thrust also can affect not only the z position but also the x and y positions of the drone. The other inputs to the

roll and pitch controller come from the lateral controller. The outputs from the roll-pitch controller are the required roll and pitch rates. These outputs will be fed to the body rate controller. This is a simple P controller that converts p, q, and r commands into three rotational moment commands. The r command is received from the yaw controller. The altitude controller is a PD controller. The yaw, roll and pitch and body rate controllers are all of first order. Thus, they are designed as just using P controllers.

2.4.Sensors for Navigation

The task of navigation is what will keep the drone flying. It is simply what makes the drone fly from one point to another. The sensors used for this task are IMU and GPS sensors. Information from these sensors is used by the computer as inputs and the computer will apply sensor fusion using an extended Kalman filter (EKF) and then convert this information into thrust values. These values are then sent to the electronic speed controller (ESC), which has a PID controller, to apply the required voltages to the motors which in turn will apply the required forces to make the drone move as needed.

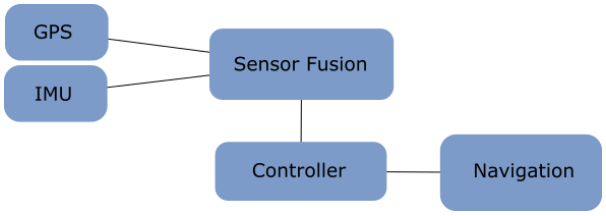

Figure II.4. Navigation task.

IMU sensors consist of gyroscopes and accelerometers. This allows the IMU alone to measure the position, velocity, and acceleration of the drone in both linear and rotational axes. IMU has several problems that can cause severe issues if used alone. For example, there is the problem of random drift, which is caused by measurement

errors. These small shifts can add up leading to a long-term drift that will cause the IMU sensor to give very inaccurate information. This problem can be solved by merging or fusing data measured from the GPS sensor and then applying EKF. The GPS is a satellite-based navigation system. GPS uses the method of triangulation from signals received from three or more satellites. A disadvantage to GPS sensors is that the receiver needs to maintain a line of sight with the satellites. This means that GPS is not of great use indoors or for example in a tunnel. Still, the fusing of data from both IMU and GPS sensors yields accurate results and is widely used in robotics. In this thesis, the simulation studies are performed by using Udacity Simulator using Unity. The simulator includes two different parts. One for free flying and to try different lines of code. The other one is for motion planning of the drone and trying different planning algorithms.

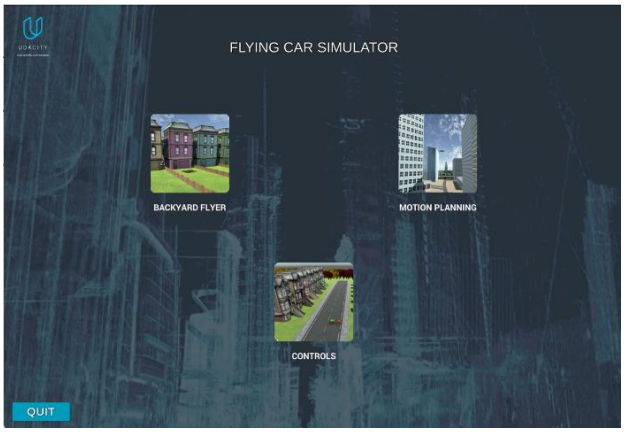

Figure II.5. Simulator interface [40].

Using this simulation, python code will be written to control the drone in the simulation. This same code, after testing it in the simulations, can be used on a real quadcopter and can be uploaded to a flight computer.

Figure II.6. Drone take-off.

A simple code is used to show how the simulator is working. The simulator and the library downloaded from *Udacity* contain some drone-related libraries that will help make the code easier. Here for example a line of code is used to start flying the drone. This code simply simulates applying a voltage to the motors so that a thrust is produced that will lift the drone to a height of five meters above the ground. In the Figure II.6, the drone is flying above the ground. The code applied is simply seen on the right side. A simple code is applied in which the height required for the drone to reach is added as a parameter to the function *drone.takeoff ()*. Then more lines of code will be tested. This time a trajectory will be planned and the drone in the simulator will be tested. The drone must arm, do a takeoff to an altitude of five meters, then fly in a square with specified coordinates that are specified in the code, then the drone must land back to its home point position and finally the drone should disarm. The transition from one phase to the other is done simply by using some rules in the line of codes in which each phase is transitioned when a certain rule is fulfilled. For example, to transition from landing phase to a disarm phase, the drone must be so close to the ground. This is written in the code as, if the altitude is less than 0.01 meters, the drone phase will change from a

landing phase to a disarm phase. Similar rules are made for the other phases as well based on the concept in mind. The lines of code used are shown as,

FLYING THE DRONE

```
1  Importing necessary libraries
2  Creating States class
3  Creating a Class with all functions required to fly the Drone
4  Starting a Mavlink Connection
5  Running functions to fly the Drone in a square trajectory
```

When running the code in the terminal, here the outputs are shown. The drone started with an arming phase in which the rotors are starting to run, then a takeoff phase where the drone takes off from the ground to an altitude of five meters, then the drone starts to move in the trajectory specified and it reaches the target points. After following the trajectory, the drone simply starts landing and finally disarms when it reaches the ground.

OUTPUT AFTER RUNNING THE CODE

```
1  Logs/TLog.txt
2  Starting connection // Mavlink connection
3  Arming transition // Drone propellers starts rotating
4  Takeoff transition // Drone starts flying
5  Setting home // Saving current position as home
6  Waypoint transition
7  Target position [10,0,3]
8  Waypoint transition
9  Target position [10,10,3]
10 Waypoint transition
11 Target position [0,10,3]
12 Waypoint transition
13 Target position [0,0,3]
14 Landing transition // Drone gets back to ground after the task is achieved
15 Disarm transition // Propellers stop rotating
```

III. PLANNING

The motion planning problem is the focus of this project. Planning here is answering the question of how the drone should reach the goal it is required to reach in the most efficient way possible without hitting or crashing into any obstacles [7]. This planning problem is just a step and is something computed by the flight computer. So, the planning problem output will be a set of trajectories which the drone should follow to reach its goal. The motion planning state machine can be shown in the diagram below. This diagram shows the many tasks a drone should do to have a successful flight. The planning task is just one step in this diagram.

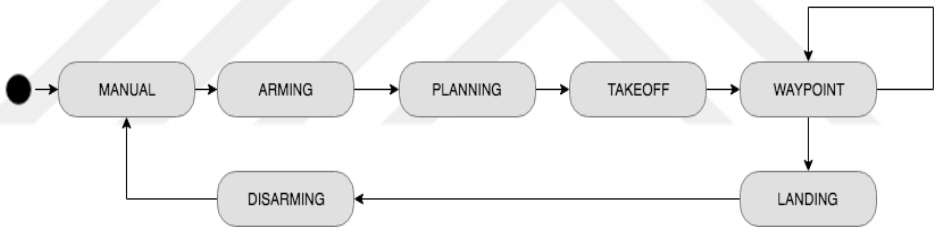

Figure III.1. Motion planning state machine.

3.1. Requirements to find a plan

Now, the next step to go, after trying the communication and the code, is to determine the planning method of the drone. A plan consists of a series of actions that will be done by the drone for it to reach the goal safely. The goal also is to try to make the drone reach the goal in the most efficient way. The plan is always restricted by some constraints like computational resources. To represent the planning problem and be able to define the search space in which the drone will be working, some points need to be present. These are a start state, a goal state, all possible actions that can be done, an assignment of some cost to every action, and finally a set of all possible states in which

the drone might find itself in the world. To start, the world can be represented as a grid or a group of squares. This planning method is known as *Graph Search Method* in which the workspace is divided into cells and then the cells are connected as a graph or a roadmap. The size of the grid (how large the squares are) determines the resolution of the representation. If the size of the squares is small, this means the grid will include more squares. This allows for a better representation of the world; however, the planning problem will require higher computational resources. Choosing a grid with bigger squares will require fewer computational resources but it means that the representation of the world will be of a lower resolution and some routes, which might be a better solution to the problem, might not be considered. So, the choice of square sizes is of great importance. After deciding on the design of the grid, now it's important to assign a cost to the actions that can be done by the drone. This cost can be used then to compare different plans. It simply will help in providing a measure of how good or bad a plan is. The cost can be taken as the Euclidean metric distance from the starting to goal point. This can be shown in Equation III.1.

$$Cost = \sum_{n=1}^{n-1} \sqrt{(x_{n+1} - x_n)^2 + (y_{n+1} - y_n)^2} \tag{III.1}$$

where the summation is for the cost of traveling between nodes to move from the starting point to the current point. Another important aspect that needs to be taken into mind is what's called heuristics [5]. Given the map of the world and the grid, now an estimate of how far the distance any given square on the grid to the goal can be calculated. Now, by using the cost function and the heuristics the planning problem can be solved.

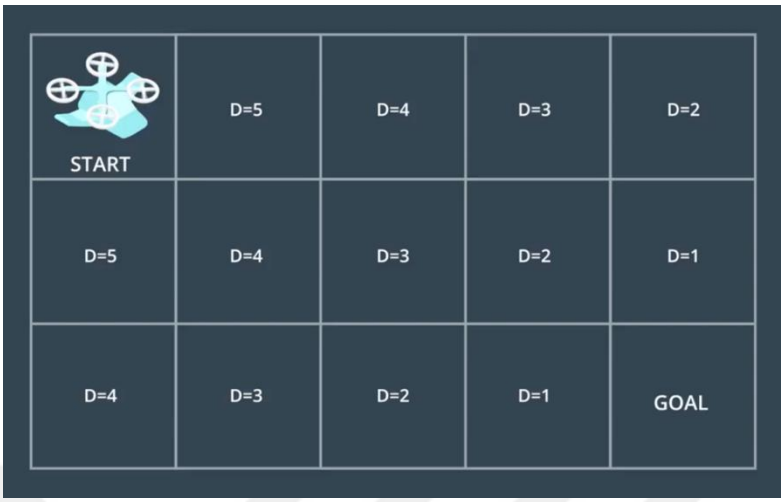

Figure III.2. Heuristic concept [28].

3.2. *A** Algorithm

In this algorithm, the cost function and heuristics are required. When planning, a set of different actions can be taken, and they might all lead to a correct solution in the end and the drone might be able to reach the goal with any of them. However, different paths will be better than other ones in terms of the resources required and the time that it will take the drone to reach the goal. What differs one path from the other will be the costs' function and the heuristics. Using them, different paths will have different total costs than the others. Searching all the paths would be so computationally complex and would take a lot of time. *A** algorithm helps with this problem of searching for the best paths without the need to look for all the paths. It allows finding the path with the lowest cost plan faster than other popular algorithms, however, it's depending on how good the heuristics are. A better heuristic means a better *A** implementation. Now all that's been discussed so far needs to be implemented as a code for the drone to understand. The idea of the world in grid representation can be made by coding in areas that are not

allowed by the drone with a value of one, while the free areas where the drone is allowed will be assigned the value of zero. An example can be seen as,

REPRESENTING THE GRID

```
1  Import numpy
2  Grid = numpy.array ([
       [0, 1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0],
       [0, 1, 0, 1, 0, 0],
       [0, 0, 0, 1, 1, 0],
       [0, 0, 0, 1, 0, 0],
   ])
```

In this grid, for example, the drone can be placed at cell (0, 0) which is the top left cell in the array. The drone is only allowed to go down, since going to the right would mean hitting an obstacle and the other directions are out of the map. The actions possible in this example are to move up, right, down, or left. So, this needs to be added in the code as well, for example, moving left would be (0, -1). So, if the drone is at a position of (3, 3) moving left would make its new position (3, 2). It's also important to keep track of the visited cells and to save any new visited cells to a list. Also, the action taken to reach that cell would be important to save. To add a cost function to the actions, each action would be made of a tuple with three elements. The first two elements represent the action itself, for example, to move up the elements will be (-1, 0). And to add a cost to the action we can add a one for example so the "up action" can be coded as (-1, 0, 1). Some actions can be given a cost higher than other actions. This can be related to real-life situations. Moving in a diagonal path might have a shorter distance than moving right then up for example. So, a lower cost can be given if the diagonal directions are to be considered. Below, is a simple code showing the four different possible actions that can be done by the drone in this example with each action having

a cost of one assigned to it. When considering the real code, different costs will be assigned to different actions. This can make the planning even more efficient. The cost can also change dynamically while the drone is in flight. If the wind speed and direction are to be considered as an example, flying against the wind would mean that more force is required by the motors which would mean less flight time. Having this information can be used along with the path information to find better results.

ADDING COST TO ACTIONS

```
1  Import Enum
2  Class Action (Enum):
       // Assign cost of each action as the third element in tuple
3  Left = (0, -1, 1)
4  Right = (0, 1, 1)
5  Up = (-1, 0, 1)
6  Down = (1, 0, 1)
```

Below are some lines of code for the simple example given before with the small map and the two-dimensional world.

SAVING START POSITION IS A QUEUE

```
1  Import Queue
2  Start = (1, 0)
3  q = Queue()
4  q.put(start)
```

ADDING CELLS VISITED TO A LIST

```
1  visited = set()
2  visited.add(start)
3  print(visited)
4  >>> {(1, 0)}
```

After giving a simple example for the *A** algorithm, now using the same approach, the real code used for solving the path planning problem can be seen in the figure below. All the possible paths are examined, and their costs are calculated. Then

the costs will be compared and the path with the lowest cost will be chosen. The last part of the code is used to retrace the steps of the path so that the drone's trajectory will start from its current position, and it will end with the goal position.

A* ALGORITHM CODE

```
def a_star(graph, start, goal):
    path = []
    queue = PriorityQueue()
    queue.put((0, start))
    visited = set(start)
    branch = {}
    found = False
while not queue.empty():
        item = queue.get()
        current_cost = item [0]
        current_node = item [1]
        if current_node == goal:
            print ('Found a path.')
            found = True
            break
        else:
            for next_node in graph[current_node]:
                cost = graph.edges[current_node, next_node]['weight']
                new_cost = current_cost + cost + heuristic (next_node, goal)
                if next_node not in visited:
                    visited.add(next_node)
                    queue.put((new_cost, next_node))
                    branch[next_node] = (new_cost, current_node)
    path = []
    path_cost = 0
    if found:
        path = []
        n = goal
        path_cost = branch[n][0]
        path.append(goal)
        while branch[n][1]! = start:
            path.append(branch[n][1])
            n = branch[n][1]
        path.append(branch[n][1])
    return path[::-1], path_cost
```

3.3. *D** Lite Algorithm

For planning a path in an environment that is well-known and static, the *A** algorithm would achieve greatly [6]. But this is not always the case. It is almost always that the environment is dynamic, obstacles show up in the drone's path and these obstacles are not considered. In this case, the *A** algorithm would be computationally complex and costly [1]. The problem is that every time an obstacle is detected, the *A** algorithm has to be run again to find a new path while avoiding the obstacle. The *D** lite algorithm is an adaptation of the *A** algorithm but in addition, it incorporates incremental search [3]. This means that the algorithm will reuse the information that is calculated from the previous searches instead of solving each search again from scratch. That makes the *D** lite a better algorithm for solving this replanning issue when detecting an obstacle [2]. This algorithm, in contrast to the *A** algorithm, runs from the goal point to the starting point. This is the main difference between both algorithms. The advantage here is that if the starting point changes, the existing tree paths can be used to quickly find a new path from the end node to the start node. Also with this advantage, the *D** lite algorithm is shown to give better results when used as the planning algorithm in case of flying the drone in an unknown environment. So, if the map is not already given as data and instead the drone has to figure out on its own the environment around it and find the best path to reach the goal, the *D** lite would be the better algorithm to use in such a situation. Now considering an example where the drone is flying in unknown terrain, and it is given a goal point where it should fly to. The drone should observe which adjacent cells are free and traversable and then it should move to one of them. From its starting position, the drone must reach its goal. While

doing that, the drone should always compute the shortest distance or else, the path with the lowest cost. The cells which have not been explored yet are taken as traversable, meaning that they can be included in the planned path of the drone. The drone will follow the planned path until it reaches its goal or until it explores a cell with an obstacle. If a cell with an obstacle is explored, the drone should recompute a new path from its new current position. The idea of avoiding obstacles when planning is done by adding cost to the nodes and edges where there are obstacles. So, if the path is planned already and then an obstacle is detected on the path, the node will be updated. If this node has a cost value of for example forty, after detecting the obstacle, this node will have a cost of infinity or a very large value. This will ensure that this node will no longer be taken into consideration when replanning. The next step is replanning. Now that some local nodes should not be considered anymore, they will be exchanged with other nodes having lower costs. If the other nodes have no obstacles in them, they will be added to the updated path. The advantage of using the *D** lite algorithm is that these local nodes are the only nodes that need to be updated. The rest of the path will be left unchanged, and this means lower computational costs and the new path will be planned so fast. A simple example that shows how the *D** lite algorithm works can be seen below. It's a simple two-dimensional grid with a starting node, colored in red, and a goal node, colored in green. The numbers show the heuristics to the goal and can be considered here as the cost directly. The thin square border moving with the red circle is the searching zone. This zone simulates for example a lidar sensor detecting obstacles around the drone. So, an obstacle can be detected if its inside this border, simulating the lidar range.

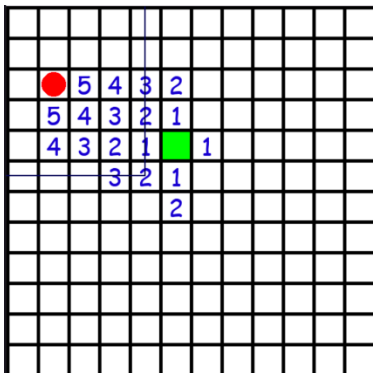

Figure III.3. Simple *D** lite algorithm problem [9].

The diagram shown in Figure III.4 shows how the cost is affected by adding some obstacles. The red circle must go around these obstacles to reach its goal. This quick simulation shows the idea of flying the drone in an unknown environment and how it would react.

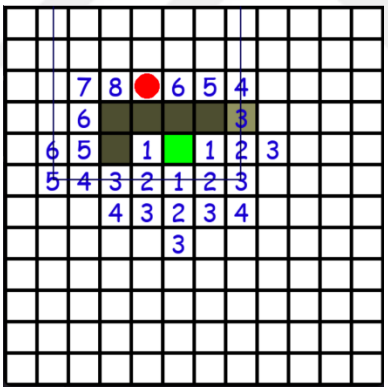

Figure III.4. Adding obstacles to the map [9].

As a code, the *D** lite algorithm would not differ too much from the *A** algorithm. As discussed before the main difference would be that the search will start from the goal node to the starting node. Then some functions will be added which are related to detecting obstacles and how the algorithm should deal with them. The function shown in the figure below is related to checking if an obstacle is detected about

the current node. If there is an obstacle detected it will just return a "*True*", while if there are no obstacles detected, the returned value will be "*False*".

DETECTING OBSTACLES

```
1  def scanForObstackes(graph, queue, s_current, scan_range, k_m):
2      states_to_update – {}
3      range_checked = 0
4      if scan_range >= 1:
5          for neigbor in graph.graph[s_current].children:
6              neighbor_coords = stateNameToCoords(neighbor)
7              states_to_update[neighbor] =
   graph.cells[neighbor_coords[1]][neighbor_coords[0]]
8          range_checked = 1
9      while range_checked < scan_range:
10             new_set = {}
11             for state in states_to_update:
12                 if neighbor not in new_set:
13                     neighbor_coors = stateNameToCoords(neighbor)
14                     new_set[neighbor] =
   graph.cells[neighbor_coords[1]][neighbor_coords[0]]
15             range_checked += 1
16             states_to_update = new_set
17 new_obstacle = False
18 for state in states_to_update:
19     if states_to_upodate[state] < 0:
20         for neighbor in graph[state].children:
21             if (graph.graph[state]children[neighbor] != float('inf')):
22                 neighbor_coors = stateNameToCoords(state)
23             graph.cells[neighbor_coords[1]][neighbor_coords[0]] = -2
24             graph.graph[neighbor].childern[state] = float('inf')
25             graph.graph[state].children[neighbor] = float('inf')
26             updateVertex(graph, queue, state, s_current, k_m)
27             new_obstacle = True
28 return new_obstacle
```

The value returned from the *scanForObstacles()* function will be used in this other function called *moveAndRescan()*. As the name suggests, this function will move a step and check if there are any obstacles in the neighboring nodes. As discussed

before, the neighboring nodes with obstacles will be assigned a cost of infinity and will not be included in the next path that will be planned.

RESCANNING FOR OBSTACLES

```
1  def moveAndRescan(graph, queue, s_current, scan_range, k_m):
2      if(s_current == graph.goal):
3          return 'goal', k_m
4      else:
5          s_last = s_current
6          s_new = nextInShortestPath(graph, s_current)
7          new_coords = stateNameToCoords(s_new)
8          if(graph.cells[new_coords[1]][new_coords[0]] == -1):
9              s_new = s_current
10         results = scanForObstacles(graph, queue, s_new, scan_range, k_m)
11         k_m += heuristic_from_s(graph, s_last, s_new)
12         computeShortestPath(graph, queue, s_current, k_m)
13 return s_new, k_m
```

IV. SIMPLIFYING THE PATH

4.1. Less waypoints

When planning the path, and after finding a feasible path to reach the goal, the drone is required to follow several waypoints to reach the goal point. By designing the world in a grid shape, each grid will be a waypoint for the drone. This means that each square on the grid on the calculated path is a temporary goal for the flight computer. The problem with this is that the drone will stop at each waypoint. This is not a good way to achieve the goal. Instead, it would be better to find a way to shorten the unnecessary waypoint in between.

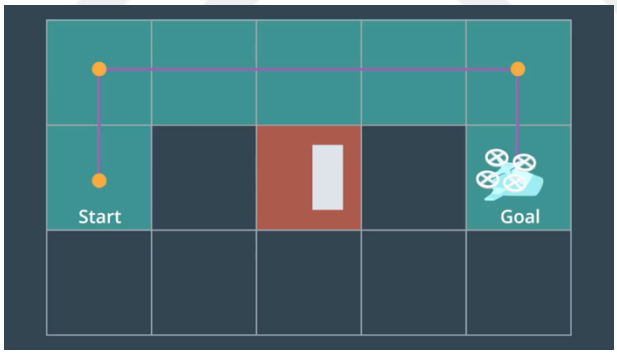

Figure IV.1. Original path [40].

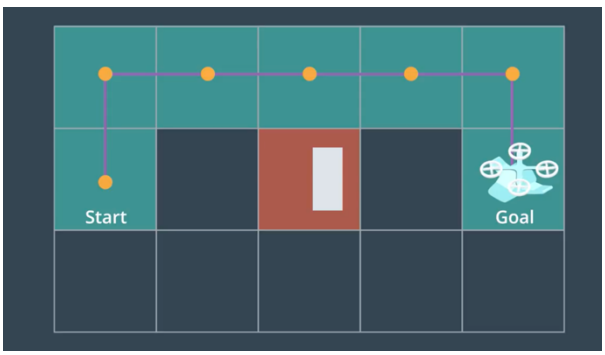

Figure IV.2. Applying collinearity [40].

This means that if for example the next set of waypoints are on the same line, and require no change in orientation for the drone, then the first and last waypoints on that line are only necessary. The other points in the middle are of no importance and can be removed. To do this, a method is needed to know the points on the same line. For this, a mathematical rule can be used, which is collinearity. This rule implies that if the area inside any three points is zero, then they are on the same line. This can be applied to each triplet of points on the path. For every three points that are proved to be collinear, the middle point can then be removed. By the end of this step, a much simpler path will be designed. Another important point that can lower the number of waypoints, and it can achieve a more optimal path is to find the diagonal paths instead of moving right and then up for example. To do so, first, it is required to determine the two waypoints. Then the line connecting them will be examined at several x values. If for any x value in that line there is an obstacle, then this means that the diagonal path is not possible. If otherwise, all possible x values on the line are examined until reaching the final point without crossing a square on the grid with an obstacle, then the path will be shortened by using the diagonal path examined. This method is called *Ray Tracing*, but it has a disadvantage. It's so computationally expensive since the numbers dealt with can be possibly float values. A solution for this is to use the Bresenham's method, which is a way that achieves the same task but deals with integer values instead of dealing with float values. To apply this method, the coordinates of the start and end points are noted. Then the following mathematical equations are applied,

$$\Delta x = x_n - x_0 \tag{IV.1}$$

$$\Delta y = y_n - y_0 \quad \text{(IV.2)}$$

Then a decision parameter P_k is calculated as,

$$P_k = 2\Delta y - \Delta x \quad \text{(IV.3)}$$

Then there are two cases that need to be considered,

Case 1: If $P_k < 0$

$$P_{k+1} = P_k + 2\Delta y \quad \text{(IV.4)}$$

$$x_{k+1} = x_k + 1 \quad \text{(IV.5)}$$

$$y_{k+1} = y_k \quad \text{(IV.6)}$$

Case 2: If $P_k \geq 0$

$$P_{k+1} = P_k + 2\Delta y - 2\Delta x \quad \text{(IV.7)}$$

$$x_{k+1} = x_k + 1 \quad \text{(IV.8)}$$

$$y_{k+1} = y_k + 1 \quad \text{(IV.9)}$$

These steps will be repeated until the endpoint is reached. This will form diagonal paths which will simplify the trajectory.

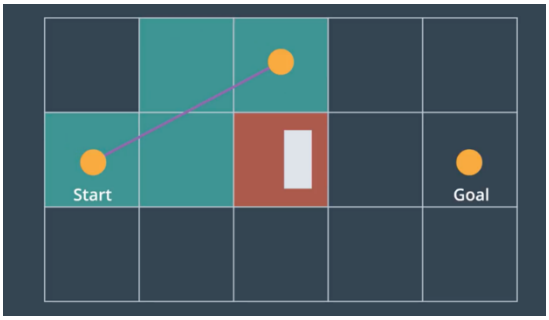

Figure IV.3. Path using diagonals [40].

4.2. Grids and Graphs

It is shown before that a grid can be a good representation of the world. Squares can be labeled as free, in which these squares can be used in the path, or they can be

labeled as infeasible, in which there might be an obstacle and the drone is not allowed to fly through them. However, as mentioned also before, the grid can have too many squares which can cause the path-planning step to be very computationally expensive. For that, the idea of designing a graph after the grid step can be useful. The idea here is, instead of looking at all possible paths by checking the cost and all of that, first, some paths will be not even considered. This step is done based on how far these paths are away from the obstacles. The concept here uses what's called the Medial axis. All the parts of the grid which are infeasible are stretched. This stretching will occur until the lines from different obstacles touch. This method ensures that the line found in the end is the one furthest away from all the obstacles. This ensures that the drone's path is the safest possible path from all the possible ones. After that, the path planning algorithm can be used on the paths left to find the shortest possible path. With the path chosen, the next point will be to add edges and waypoints on the path for the flight controller to use them. It is mentioned in Chapter 3 that the size of the grid will affect computation complexity. In the diagram below, the effect of increasing the grid size on both search algorithms the *A** and the *D** lite can be seen. As the size increases the node expansions when using the *A** algorithm is higher than that of the D* lite algorithm. This occurs because when the *A** algorithm is replanning, it has to search for a completely new path when an obstacle is detected. Also, this can be seen in the allocations. The *A** algorithm reallocates many nodes during replanning, while the *D** lite only searches and reallocates some nodes which are related to the obstacle detected, and then it uses the previously calculated path again.

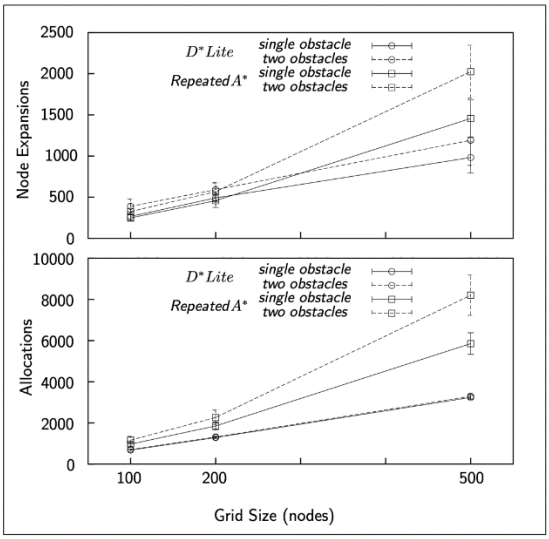

Figure IV.4. Effect of grid size on search algorithms [9].

4.3. Dead-bands

After finding the path and planning the trajectory that will be used by the drone to reach its goal. Now, the next step is to ensure the plan will be executed smoothly. For this to happen, the drone shouldn't stop at each waypoint, instead, it must move from a waypoint to the other continuously. The problem here is that the position of the drone is mostly received by a GPS signal which can be affected by a lot of things like the quality of the components or even atmospheric changes. For the drone, this will look like the path itself is moving around. This will cause the flight controller to send a lot of different signals that will affect the smoothness of the flight. To solve this issue, the idea of a dead band can be introduced to the flight controller. A dead-band is simply an area that will be around the waypoints. When the drone reaches this area of the first waypoint, the flight controller will start to give the drone commands to go to the next waypoint instead of trying to reach exactly the first waypoint.

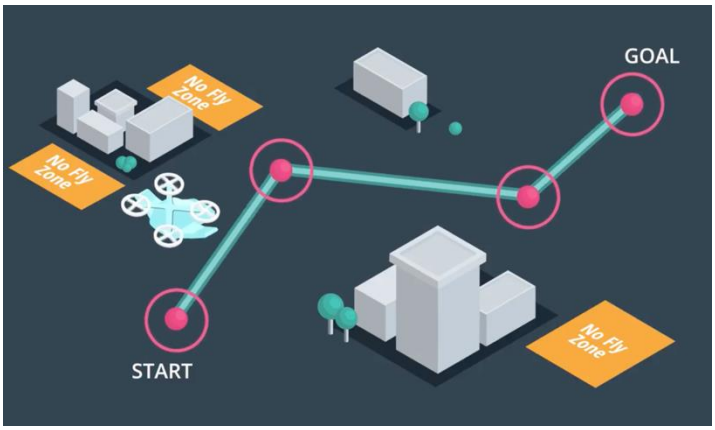

Figure IV.5. The idea of a dead-band [40].

The area of the dead-band can be chosen in a way that makes the plan executed faster and makes the drone reach the final waypoint precisely. In other words, the intermediate waypoints can have wider dead-bands since they are not so important while keeping the dead-band of the final waypoint tight to ensure the drone reaches the final point precisely. Also, another idea is to make the dead-band a function of velocity. If the drone is moving at a higher speed, the dead-band can be wider.

V. 2D TO 3D REPRESENTATION

Up to this point, the problem of planning and all what is done so far is being considered in two-dimensional space, including the idea of modeling the world as a grid or a graph. In real life, the drone will be flying in a three-dimensional world. In three-dimensional, the thing which is important again is to figure out which places in space are free and which places contain obstacles. In moving from two-dimensional to three-dimensional the planning problem has a higher computational cost. So, it must be decided what to optimize for, thus there will be some tradeoffs as well.

5.1. Three dimensional grids

The In three-dimensional, the same concepts discussed before in two-dimensional still apply, however in two-dimensional, the grid is made up of pixels as elements. These elements in three-dimensional are called voxels which is a volume element. So instead of having free and infeasible pixels, in three-dimensional, there will be free and full voxels. Once the obstacles are determined in the map, planning could be made using the same algorithm used before, which is the *D** lite algorithm, to find a path from the start point to the goal point. Even the code itself used in two-dimensional can be used in three-dimensional, however, a three-dimensional grid will surely require more memory to store, especially if the map is large and a high resolution is required. Because of this complexity, another alternative solution is what's called a 2.5d map.

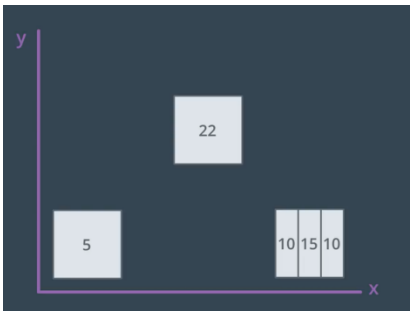

Figure V.1. 2.5D map representation [40].

5.2. 2.5D maps

In a 2.5d map, the idea is to model an obstacle as an extension from the ground plane to a certain height. Doing this, the planning map can still be stored as a 2d map. The only difference is that, instead of labeling the cells as free or infeasible, the cells will contain a number that is the height of the obstacle. So, the drone has to stay at an altitude higher than the numbers on the cells. A cell that has no obstacles, which means it's free, has no number, or in other words, has a zero. This means the drone can fly at any altitude over these cells. Using this method lowers the complexity of planning however it will still take a long time to plan than the two-dimensional planning time since the drone needs to take actions to change the altitude as well. This method of using 2.5d maps has a drawback, mainly, it is no longer possible to fly under tree branches or under bridges for example. This is not a big deal, since the drone has many options to reach its goal. In the diagram shown below, for example, there are three buildings. The values inside the obstacles show their height.

5.3. Graphs in three-dimensional

As discussed in two-dimensional space, working with a grid representation can be quite expensive. It is seen then that the idea of working with a graph is much more

efficient even though they might not capture the entire geometry of the world. In three-dimensional, a new representation, which is the 2.5d, is also seen to be better than a three-dimensional representation. So, again 2.5d graphs will be used instead of three-dimensional graphs. The idea here is that random states will be sampled, then they will be checked if they are inside obstacles or if they are in free space. The samples which will be inside obstacles will be removed while the ones in free space can then be used and connected to form a graph. Then from the graph created, a path can be chosen using any search algorithm as what is done in two-dimensional. Knowing whether a point is inside an obstacle or not is not hard in this case. A building for example is modeled in 2.5d representation as a simple two-dimensional square with a number. The number is just the height of the building. So, for example, if a random point is at the coordinates (1, 4, 8), which means it is placed at one point on the *x*-axis, four points on the *y*-axis, and eight points on the *z*-axis, if at that coordinate the square has a number of nine, then it means there is an obstacle of height nine, thus the random point is disregarded as it is inside an obstacle. If, however, the square has a number five for example, then it means that the point is at a height higher than the obstacle itself, therefore it can be used in the graph.

5.4. Random sampling in code

Obstacles can be made easily in Python using a package called Shapely. From this package, a polygon can be easily defined by giving a set of coordinates that describes the obstacle edges and height. Then using the same package, a point can be selected randomly and checked whether its inside the polygon or not. An example of a Python code can be shown as,

CREATING AN OBSTACLE

```
1  from shapely.geometry import Polygon
2  coords = [(0, 0), (1, 0), (1, 1), (0, 1)]
3  poly = Polygon(coords)
```

Here the Polygon shape or the obstacle in this case has been defined. Giving these coordinates creates a square of parameter four. To check that a polygon is created, using shapely package some attributes can be checked as well.

CHECKING OBSTACLE ATTRIBUTES

```
1  print(poly.area)
2  print(poly.length)
3  print(poly.bounds)
4  > 1.0
5  > 4.0
6  > (0.0, 0.0, 1.0, 1.0)
```

As can be seen, the polygon is defined with an area of one and a length of four. The edges or bounds of the square are also shown. For the purpose this package is used, two points will be defined. One will be inside the polygon bounds while the other is defined outside the bounds. This is exactly what is required in defining the 2.5d graph. The lines of code are shown below and the results of true and false can be seen as well. True meaning that the point lies inside the obstacle while false means it is not:

CHECKING IF POINTS ARE INSIDE AN OBSTACLE

```
1  from shapely.geometry import Point
2  P1 = Point(0.5, 0.5)
3  P2 = Point(1.5, 1.5)
4  print(poly.contains(P1)
5  print(poly.contains(P2)
6  > True
7  > False
```

The example given above dealt with an obstacle in the ground plane. The only difference now is to add the height to the polygon and examine the point with three

dimensions instead of two dimensions as in the example given. The world used in the simulation has buildings in them that can be used as obstacles. So, this can be a great example to show the random sampling idea. In this next part, random sampling will be implemented on the map used in the simulation. The full code will be added below with some description,

RANDOM SAMPLING IN CODE

Importing necessary libraries

```
import time
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from shapely.geometry import Polygon, Point
```

Defining some attributes

```
def extract_polygons(data):
    polygons = []
    for i in range(data.shape[0]):
        north, east, alt, d_north, d_east, d_alt = data[i, :]
        obstacle = [north-d_north, east-d_east, north+d_north, east+d_east]
        corners = [(obstacle[0], obstacle[1]), (obstacle[0], obstacle[3]),
                   (obstacle[2], obstacle[3]), (obstacle[2], obstacle[1])]
        height = alt + d_alt
        p = Polygon(corners)
        polygons.append((p, height))
    return polygons
```

Sampling random 3D points

```
def sample(data, k):
    xmin = np.min(data[:, 0] - data[:, 3])
    xmax = np.max(data[:, 0] + data[:, 3])
    ymin = np.min(data[:, 1] - data[:, 4])
    ymax = np.max(data[:, 1] + data[:, 4])
    zmin = 0
    zmax = np.max(data[:,2] + data[:,5])
    num_samples = k
    xvals = np.random.uniform(xmin, xmax, num_samples)
    yvals = np.random.uniform(ymin, ymax, num_samples)
    zvals = np.random.uniform(zmin, zmax, num_samples)
    samples = list(zip(xvals, yvals, zvals))
```

```
29      return samples
Checking if points collides with any obstacles
31  def collides(polygons, point):
32              for (p, height) in polygons:
33          if p.contains(Point(point)) and height >= point[2]:
34              return True
35      return False
36  def to_keep(samples, polygons):
37      to_keep = []
38      for point in samples:
39          if not collides(polygons, point):
40              to_keep.append(point)
41      return to_keep
```

The beginning of the code is importing the important libraries that will be used in the code. The file containing the obstacles of the map in the simulation is added and the data is read from the file. In [6], an attribute is created in which polygons are created from the data read from the file. So, the 4 corners are extracted first, then the heights of the obstacles are appended or added to the polygons. Now that the polygons are extracted, random samples will be created. So first, the limits of the figure are recognized and saved in x_{min}, x_{max}, y_{min}, y_{max}, z_{min} and z_{max}. In [25], the samples are created randomly using the *np.random.uniform()* function. The samples are created, however, as discussed before, only the ones which are not colliding with obstacles need to be considered. For this, the colliding samples are removed by creating a function called collides, which will check whether a sample point is inside the obstacles or not. If yes, the point will be removed, if not, the sample point will be saved inside a list called *to_keep*. The last step now is to visualize the obstacles and the sample points on a figure. For that, a grid is created with the max and min points. And the obstacles, as well as the randomly sampled points are both plotted in the figure. The results can be

seen in the Figure V.2, in which the obstacles are colored in black, while the sample points are colored in red,

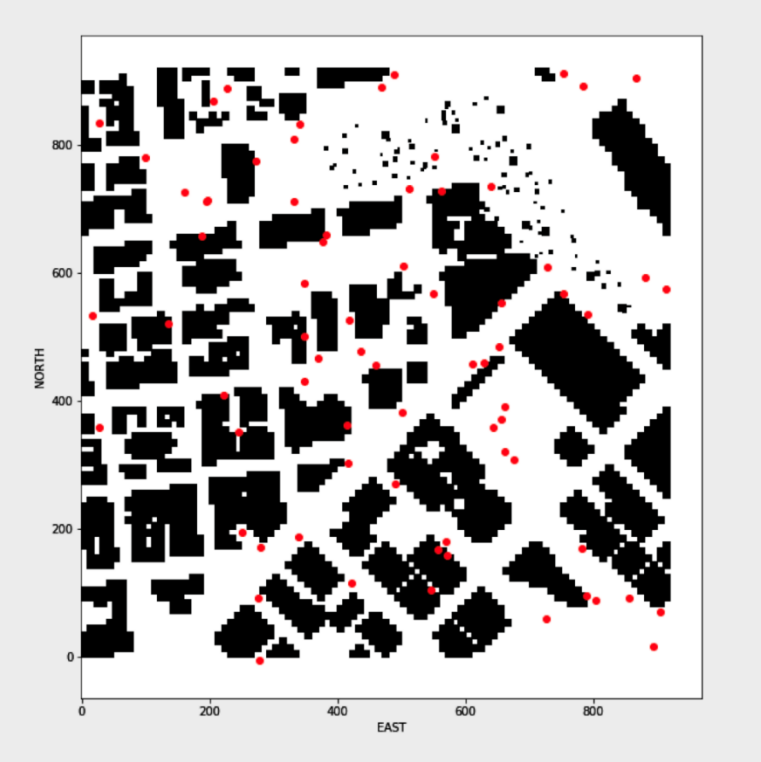

Figure V.2. Result of random sampling.

The next step now is to build a graph and run a search algorithm to find a path that can reach the goal without colliding with an obstacle. These three steps discussed so far are what is named a probabilistic roadmap. Starting with scattering random samples, then rejecting the colliding ones, and finally connecting the feasible samples together with edges. Again, like what is made in 2d, lines connecting samples will be discretized and checked. If while building the line an obstacle is on the way, this edge will be disregarded. Doing this with all points will create a graph showing all the possible edges that can be used to plan a path. This graph might not contain the most optimal solution in terms of minimum cost, minimum distance, or even minimum time;

however, it can be used to efficiently find a trajectory through the environment. Design decisions can be made here, for example, if more random samples are created, a more optimal solution can be found. In the end, an algorithm, like the *A** algorithm or the *D** lite algorithm discussed before, can be used to find a plan. Running the random sampling code again and implementing the path planning algorithm gives the following graph.

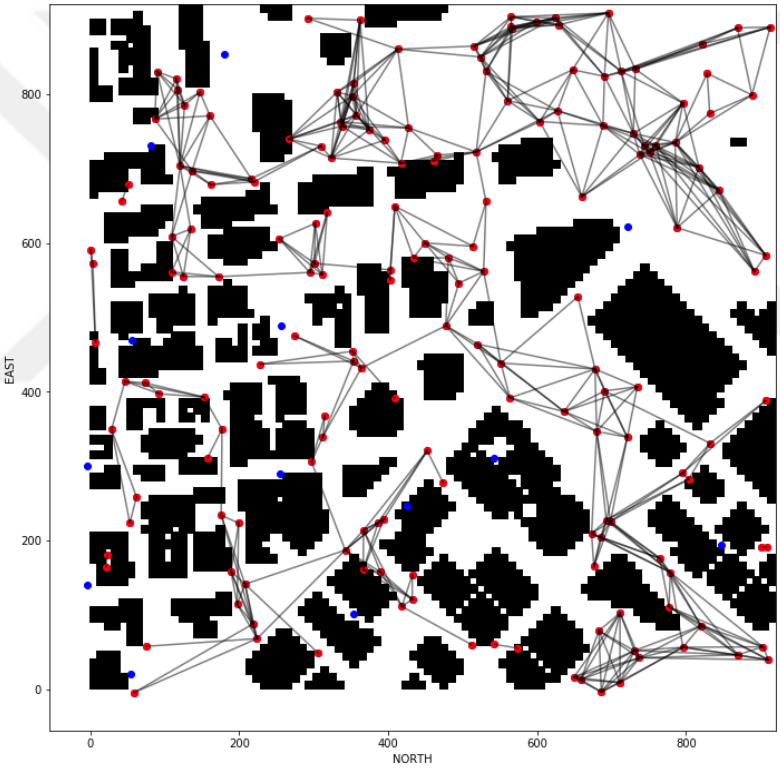

Figure V.3. Graph of possible paths.

VI. CONTROLLER DESIGN RESULTS

The control problem is discussed earlier. After defining a path for the drone to follow, it's important to make sure that it can be able to follow these trajectory points smoothly. A good plan will avoid hitting obstacles but if the control problem is not well designed, then the drone might not follow the trajectory well and it can still hit obstacles that are already avoided in the plan. In this section, some simulation results will be shown that will show how the drone reaches a certain node and how it will follow the trajectories planned. Some parameters will be changed as well to show how they can affect the drone's behavior.

6.1. Some results for the drone control

In the following section, some results will be shown while running the simulation. The results will show how the drone can reach its target trajectory smoothly and with high robustness as well. The model of the drone itself might be uncertain. Also, the world around drones is unpredictable. In other words, it is impossible to predict for example the wind which can exert a force on the drone. All these things need to be taken care of with control. In the first part, three drones with different masses will be examined with the same control scheme. They started from the same z or altitude, and they are required to reach a point in the x direction.

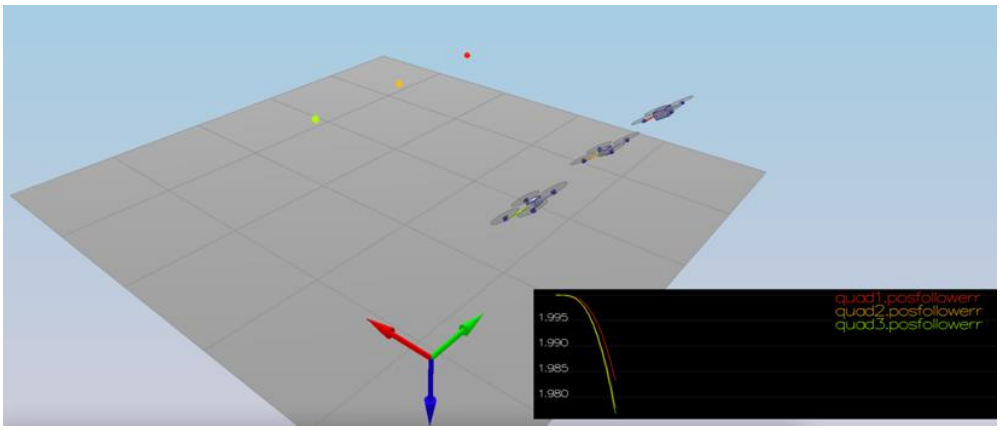

Figure VI.1.1. Three drones reaching a goal point (1).

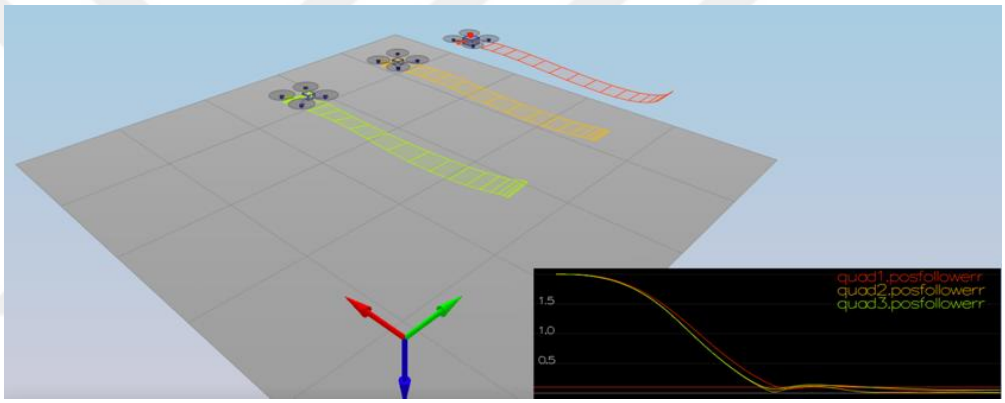

Figure VI.2. Three drones reaching a goal point (2).

They managed to reach the goal point stably. the error between their current position and target position can be seen in Figure VI.1. The error is reduced to zero as well, showing that the target point is reached successfully. So, it can be said that the control used is robust to parameter uncertainty in the model. Next, using the same simulation with the three drones, an external force can be added that can simulate wind for example in real life. This disturbance needs to be overcome as well with the control scheme designed since in real life it is never known what disturbances can be exerted on the drone.

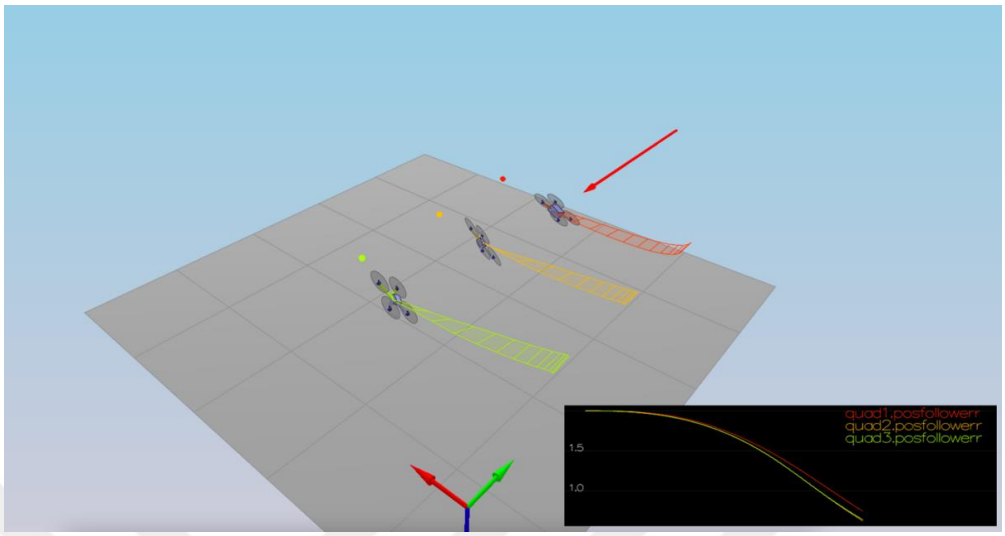

Figure VI.3. Three drones reaching a goal point adding disturbance (1).

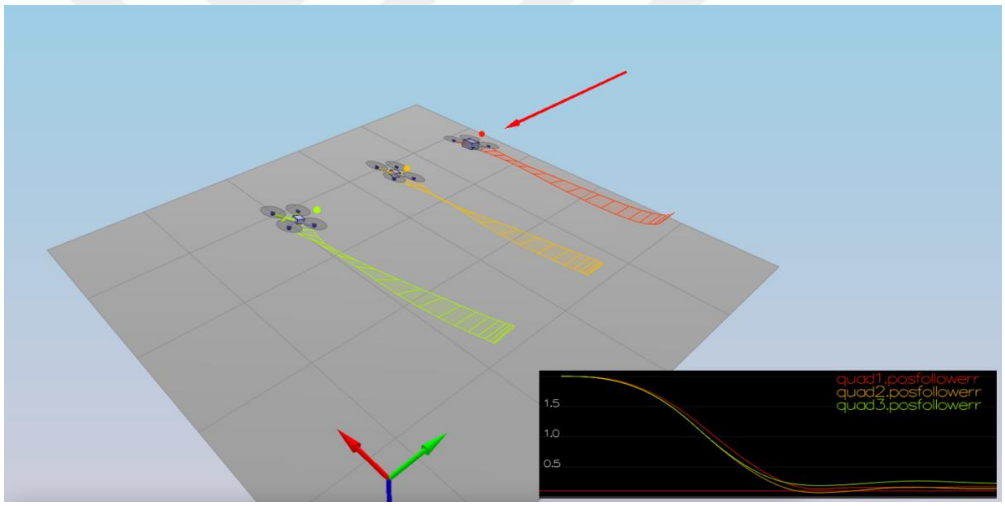

Figure VI.4. Three drones reaching a goal point adding disturbance (2).

Even though a force is exerted on the drones, still they managed to remain stable and reach close to the goal point. The force exerted is equally opposing the drone's thrust force that's why the drones didn't exactly reach the goal point. But as soon as the disturbance is removed, the drones successfully reach the goal point and remain there. The idea of having a robust controller is of great importance especially when dealing with a system like a drone, since the environments in which drones fly are almost always uncertain. Different forces can act on the drone from any direction at any time

and the drone should be able to remain stable under all these different uncertain situations.

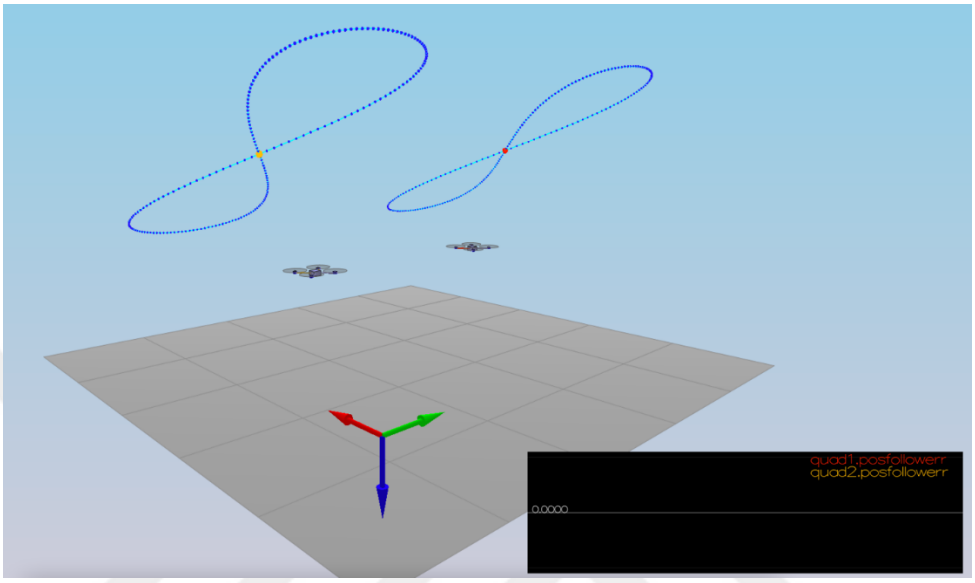

Figure VI.5. Trajectory following.

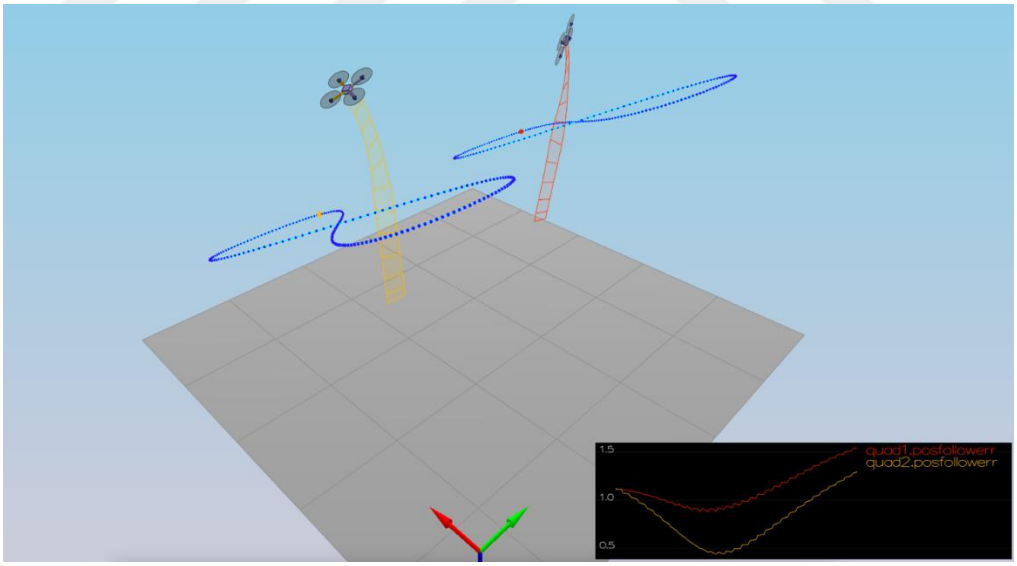

Figure VI.6. Drone overshoot.

Another so important simulation will be discussed now. It is already shown how a trajectory can be created and the algorithm is shown to successfully find a path while excluding the waypoints that cross an obstacle. However, it is not shown yet how the drone can follow these waypoints smoothly. So, in this section, a trajectory will be

designed, and the drone must be able to follow it. First, the trajectory plan with the shape of the number eight is shown. The drone started somewhere away from the path, so in the beginning, it will try to reach the waypoint quickly, which will lead to overshoot. However, after reaching the path, the drone can be seen to follow the path smoothly. From the graphs, the error between the current state and the desired state can be seen. So, the drone with the orange path has better path tracking than the drone with the red path. The drone with the red path is still following the path so the task is done, but its mass is higher than the other drone, that's why it has higher inertia which makes it harder for it to reach the desired state given all the maneuvers in the path.

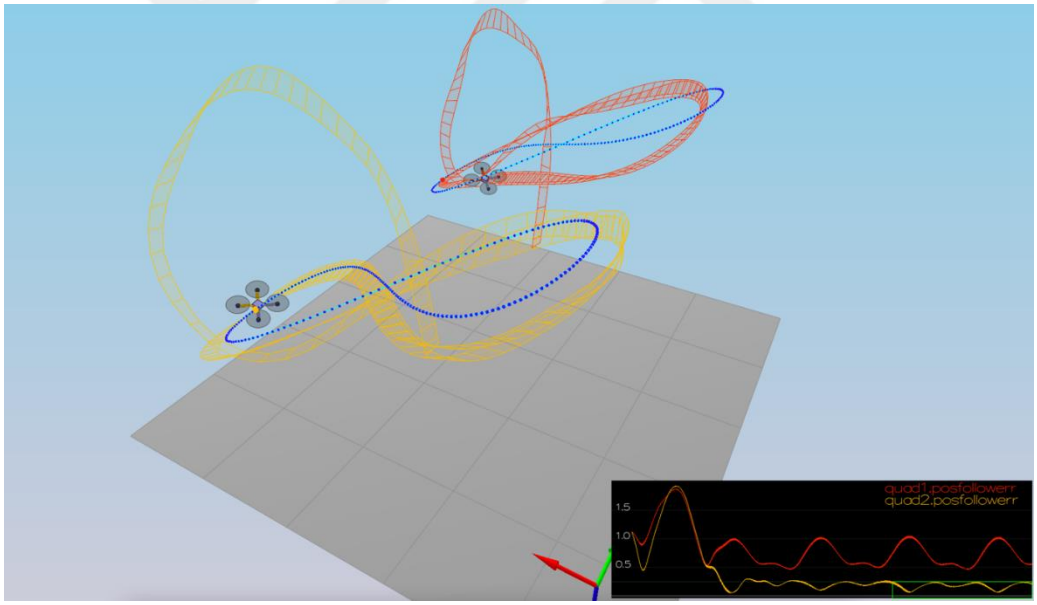

Figure VI.7. Drones following the path.

VII. KALMAN FILTER FOR SENSOR FUSION

Different sensors are used in navigation as mentioned in Chapter 1. The measurements from those sensors are merged to achieve more accurate results. For merging, EKF is used. A Kalman filter is an optimal estimator which is used to predict measurements from inaccurate and uncertain measurements [8]. Kalman filters are widely used because of their online real-time processing and the simplicity of formulation as well as their optimality. The simple Kalman filter can be used for linear systems which is not the case in this thesis. The drone is a nonlinear system and requires a nonlinear filter. For that, the EKF is used instead which can be used for nonlinear systems. Kalman filters can be divided into two steps: The prediction step, which results from using the dynamic model, and the correction step, which results from the sensor model. An EKF can be applied by first linearizing the nonlinear equations using Taylor expansion. This expansion will help approximate the nonlinear equations using simpler linearized equations. Then Kalman filter can be used. These steps explain how an extended Kalman filter differs from the basic one. In the case of the drone, the EKF can be used to estimate position, angular and linear velocities, and accelerations. This can be done by using measurements got from accelerometers, gyroscopes, and GPS sensors. The basic Kalman filter cannot be used to solve nonlinear problems because it assumes that the motion and measurement models can be in the Gaussian world. This is not the case when dealing with nonlinear functions. The linearization method makes use of the Taylor approximation. A Jacobian matrix will be used to linearize the prediction and

the correction parts. The prediction contains two parts: prediction of the state and prediction of the covariance matrix of the noise. These can be written as the following,

$$\bar{\mu_t} = g(u_t, \mu_{t-1}) \tag{VII.1}$$

$$\bar{\Sigma_t} = G_t \Sigma_{t-1} G_t^T + R_t \tag{VII.2}$$

The next step is to correct these estimations. The correction part occurs based on the measurement units which are the sensors. In the drone case, the measurement is got from the GPS and the IMU sensors. The correction part also includes an important weighting factor which shows how much the measurements from the sensors can be trusted over the prediction. For example, when the drone is in a tunnel, the GPS signal cannot be trusted much because GPS needs to be considering sight with the satellites. In this case, IMU sensor can be trusted more as well as the prediction calculated from using the system model. The correction part can be written as follows,

$$K_t = \bar{\Sigma_t} H_t^T (H_t \bar{\Sigma_t} H_t^T + Q_t)^{-1} \tag{VII.3}$$

$$\mu_t = \bar{\mu_t} + K_t(z_t - h(\bar{\mu_t})) \tag{VII.4}$$

$$\Sigma_t = (I - K_t H_t)\bar{\Sigma_t} \tag{VII.5}$$

The H_t and G_t are the jacobian matrices linearizing the function at specified μ_t. The K_t matrix shows how much the measurement can be trusted over the prediction.

VIII. SIMULATION RESULTS

The results shown and discussed before show the control part of the drone. So, it is seen how the drone will go from one point to the next one in a stable and smooth manner. In this section, a full simulation result will be shown. This will include everything discussed so far. The drone will have an initial position and altitude. Then a final goal position will be sent to the flight computer. A plan will be made, and a set of trajectories or waypoints will be designed which the drone has to follow in order to reach the goal in an efficient way. In the simulation, the waypoints will also be visualized as nodes and edges. In the simulator, the map is given, and the obstacles are already known.

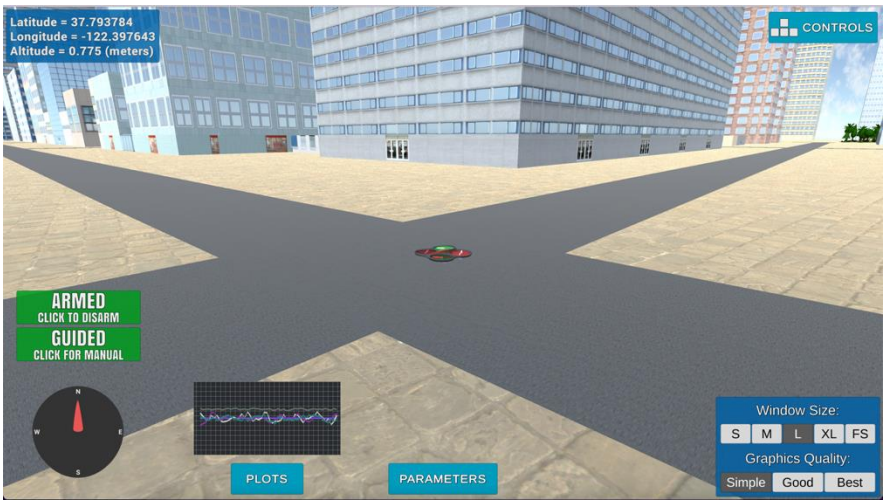

Figure VIII.1. Drone and its environment.

The simulation will be started with the quadrotor at a random rest point in the map. Then a final goal point will be given that the drone should reach. The diagram below shows the drone at rest and the environment surrounding it. The latitude, longitude and altitude information can be seen on the top left part. When running the code, the drone's

mode is switched to "*Guided*" and it's "*Armed*". The propellers start rotating when the drone is armed. It is also worth mentioning that on the bottom left there can be seen some plots. These plots include a lot of important information that can be used to check the drone's performance and behavior. The computation of the trajectory takes a little time, between 20-60 seconds, based on how far the initial and final positions are from each other and based on the number of obstacles in the way. The algorithm is trying to find the best optimal path as discussed before. For both discussed algorithms, *A** and *D** lite, the first run for a given known map will take almost the same time to find a path, however, this simulation will be run with the *D** lite algorithm. First, the final goal will be chosen close to the initial position to see how long the algorithm will need to find a path.

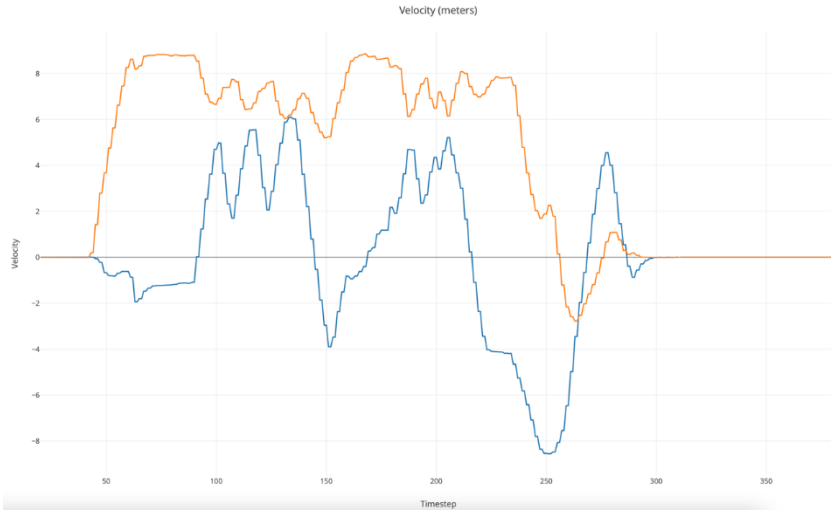

Figure VIII.2. Velocity in roll and pitch.

It can be seen in the graph above that the drone took around 35 seconds before it started moving. The algorithm starts looking for a plan once the code starts running at time 0. The drone took around 265 seconds after finding the plan to reach its goal point. The next two graphs are for the altitude and the position of the drone from its starting

position to its goal position. The altitude graph shows that the drone starts rising once the code is started. It stays at an altitude of around 2.8 meters during flight time then when it reaches the end goal it lands. In the simulation, there is no sensor for landing so the drone just switches off its propellers that is why there is an overshoot that can be seen at around 280 seconds.

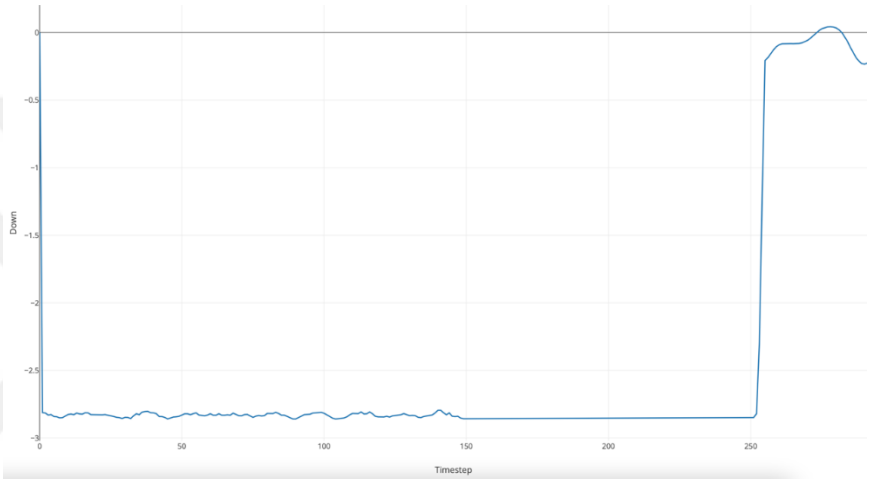

Figure VIII.3. Altitude of the drone.

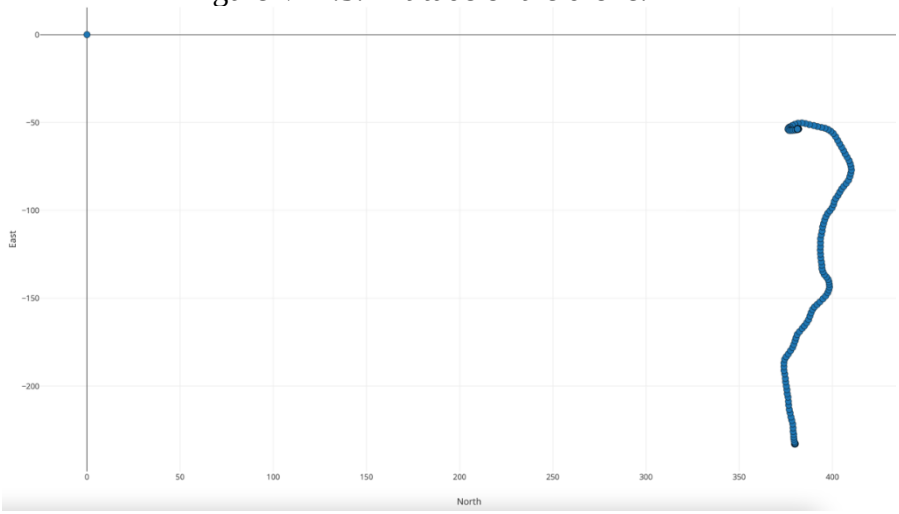

Figure VIII.4. Position of the drone.

This diagram above shows the position of the drone as seen from above (bird's view). The drone starts from the coordinates (370, -240) and the goal position is set to the coordinates (370, -52). It can be seen that there are not many turnings since the way

is almost straight and not many obstacles are faced during the flight time. In the simulation, the waypoints can be visualized as seen in the diagram below.

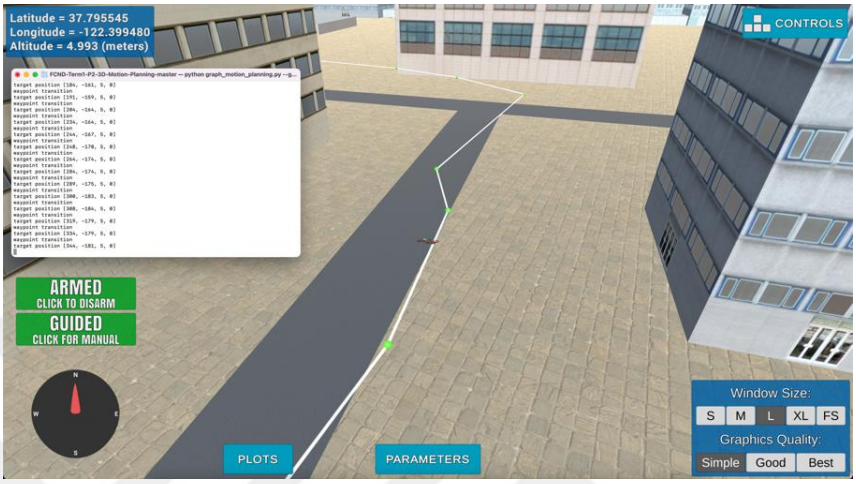

Figure VIII.5. Path visualization and waypoints information

Now the goal position will be set further away from the drone's initial position. It is predicted that the algorithm will take longer until finding a path.

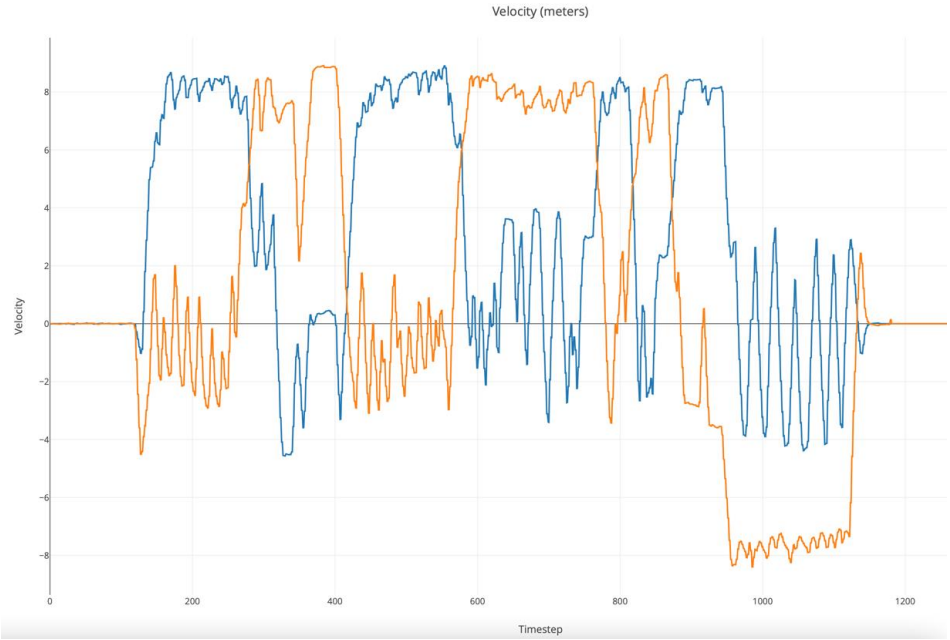

Figure VIII.6. The velocity of the drone in roll and pitch.

It can be seen that the drone this time took around 85 seconds to find a path. That is due to searching on a bigger map with more obstacles. The amplitude graph shown below shows an increase in the amplitude at the middle of the flight time. This increase resulted because there is a short building that the drone can fly above it while keeping the cost low. Other buildings are too tall in the simulation, and it would increase the cost if the drone planned a path from above them. The diagram after shows the path the drone took to reach its goal position. The plan this time is more complex, and the drone had to avoid much more obstacles on its way. Now, the error between the target point and the real position of the drone will be compared. It is important to remember the concept of dead-band as mentioned in chapter 5.6. It is introduced to make the drone move smoother. This will cause the error between the target and the real value to be slightly higher than zero because the dead-band allows the drone to reach an area around the target point and not exactly on it. The diagram below shows the target points in yellow and the drone's positions in blue.

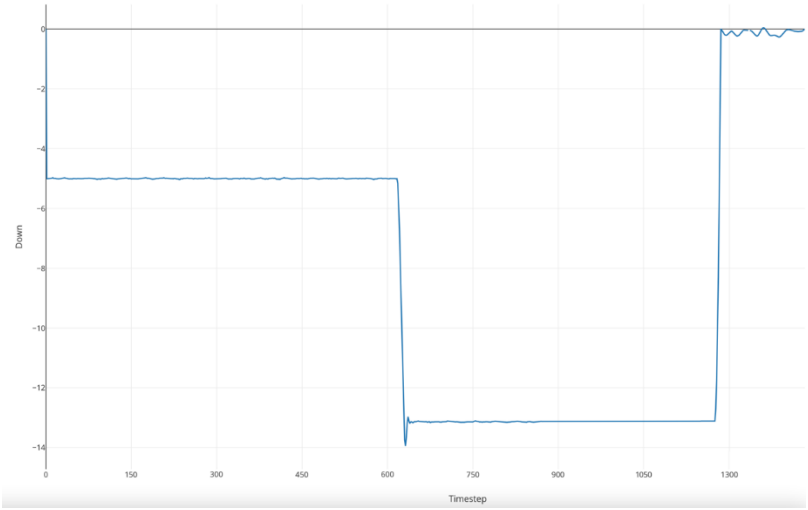

Figure VIII.7. The amplitude of the drone for longer path.

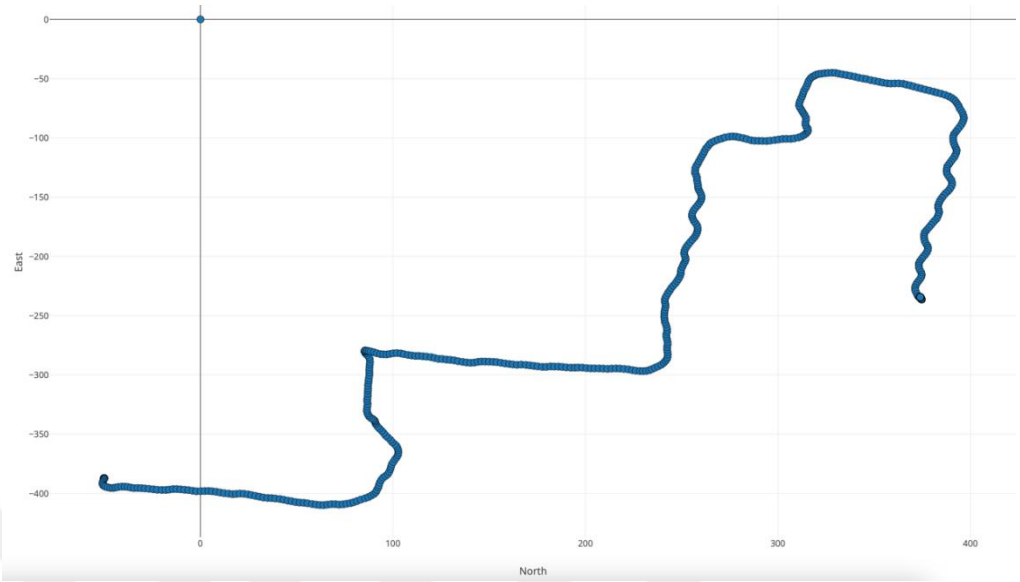

Figure VIII.8. The position of the drone for longer path.

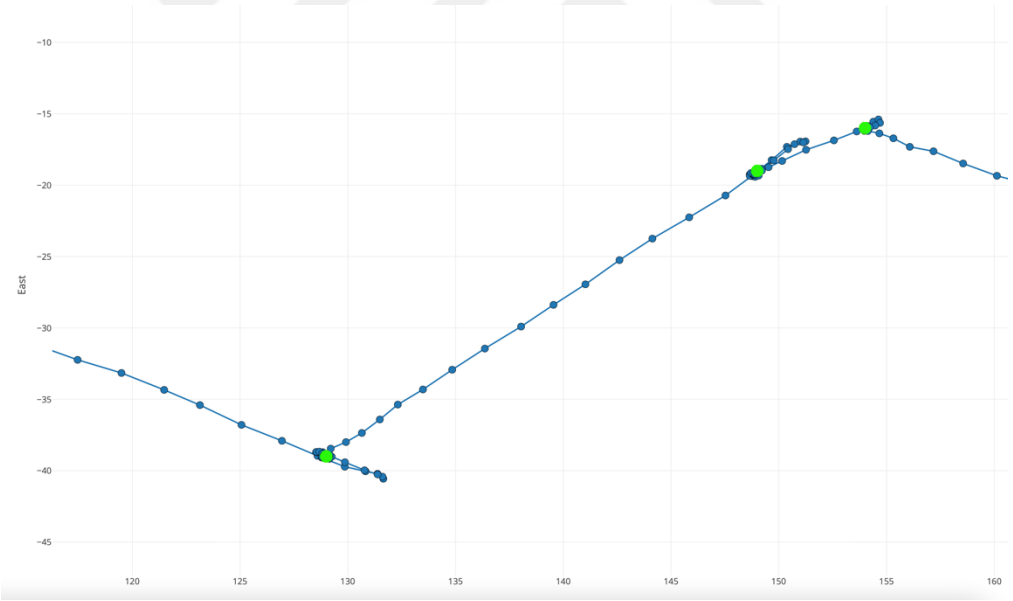

Figure VIII.9. Target point and drone's real position.

The error between the target and the current drone's position can be seen in the figure below. The drone receives a target point that it has to go to. Once the drone enters the accepted range, which is determined by the dead-band, it will directly receive a new target point. This can be seen in the figure, as the error decreases to almost zero at time

47, the error suddenly increases again. That occurs when a new target point is sent to the drone. The error reaches a minimum value of 1.44. The only point the error gets very low is at the final point. The error reaches a value of 0.2 meters which is not high taking into consideration GPS and IMU uncertainties. The reason why the error reaches the lowest value at the final position is that there is no dead-band introduced at the end position. It is assumed that the drone must reach the final point exactly and that it is critical. So, no dead-band is added at the final position.

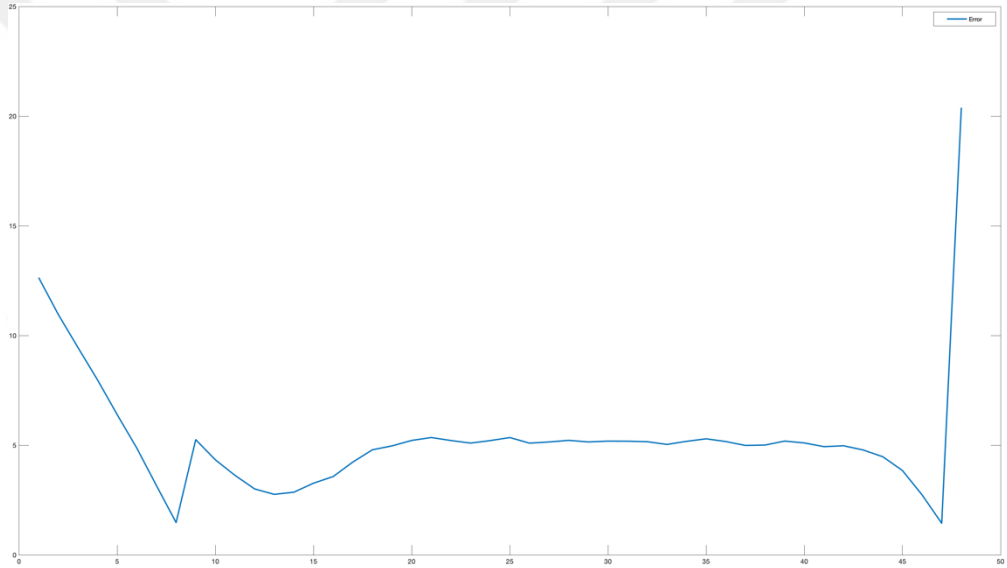

Figure VIII.10. The error between target and real drone's position using *D** lite.

This simulation is run with the *D** lite algorithm. From time 20 until time 40, the drone is hovering in place, this can be seen from the error that it is not increasing or decreasing. This occurs for the fact that there are some buildings that are removed from the map file that is fed to the algorithm. So, they will be added during the simulation to be dealt with as discovered obstacles on the way. So, the time taken by the *D** lite algorithm to find a new path is around 20 seconds.

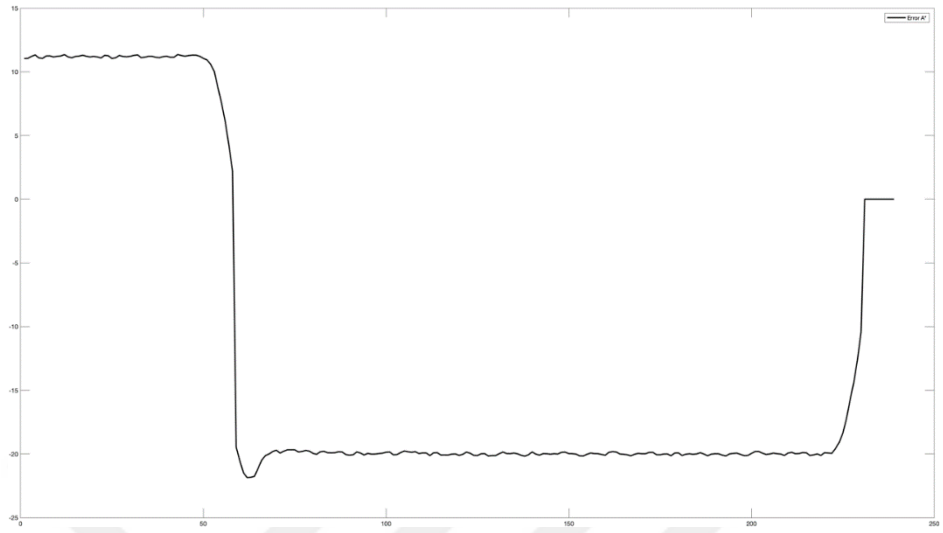

Figure VIII.11. The error between target and real drone's position using *A**.

The same is applied now using the *A** algorithm to see how long it will take to find a new path after discovering the new obstacles. The diagram below shows the error again between the target point and the real drone's position while using the *A** algorithm for path planning. The drone hovered in place for around 180 seconds this time. This shows how fast the *D** lite algorithm is compared to the *A** algorithm when used in a changing environment. This simulation is repeated several times while changing the number of obstacles added during the flight. A comparison between the *A** and the *D** lite algorithms is made to check the effect of discovering obstacles that are unknown before. This means that after the algorithms found a path, and during the flight, the drone will discover new obstacles on its way. This requires the algorithms to search for a new path. This is done for both algorithms by increasing the number of obstacles. The new allocations and the node expansions are checked in comparison to the number of obstacles discovered. As can be seen from the figure above, for both algorithms the number of node expansions increased linearly with the number of obstacles discovered. However, the rate of increase with the *A** algorithm shows double

the increase of the *D** lite algorithm. The result for the number of node allocations shows almost a constant value for the *D** lite algorithm, while there is again a linear increase for the *A** algorithm. These results show that the *D** lite algorithm works better and faster than the *A** algorithm when dealing with an unknown environment which is exactly the case with the drone. This simulation and results proved that the control and planning tasks have been fulfilled successfully. This means that the dynamics of the drone are modeled correctly and that the controller could be used to achieve the goal. The results also show that the planning algorithms also work. So, this means that the code can be used on a real drone in a real environment and similar results can be achieved. However, if the map is unknown and the buildings are not already given and fed into the code, the *D** lite algorithm is shown to achieve better. Better here means faster results since it updates its path with every obstacle detection. While in the case of the *A** algorithm, the algorithm is run from the beginning every time an obstacle is detected.

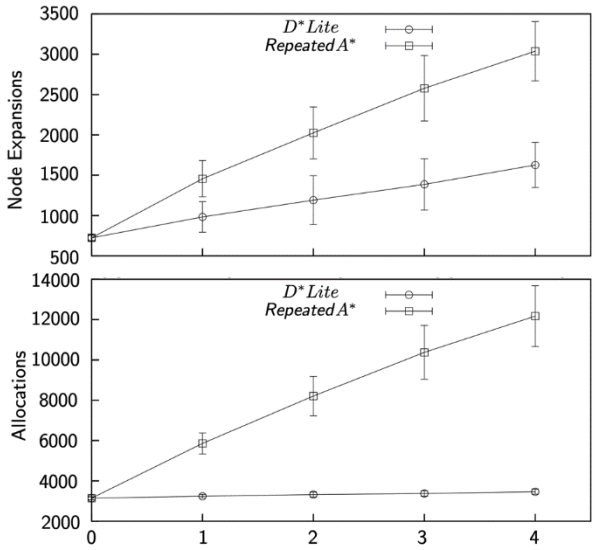

Figure VIII.12. Effect of obstacles discovered on *A** and *D** lite [9].

IX. CONCLUSION

In this thesis, a drone simulation is performed. Various aspects are looked at including the sensors used and why they are selected. Then the dynamics and control aspects are also discussed. After that the focus of the project, which is solving the path planning problem, is examined and looked at. It is seen how the two algorithms, the *A** and the *D** lite, are different. The main difference between these algorithms is that the *D** lite algorithm searches from the goal node toward the starting node, while the *A** algorithm goes the other way. The *D** lite algorithm is seen to perform better, especially in the case where the environment is unknown. Even when the environment is known but there are obstacles that can show up in the environment, the *D** lite algorithm would achieve better results.

The issue that can be further improved is the simplifying of the path. It is seen from the trajectories formed that still the path has some points which can be made straight. This can improve the drone's performance since there will be fewer waypoints in the path and this means that the drone will not consider stopping at these points which will save some time. Another interesting point that can further improve the planning problem would be optimizing the safety distance and dead-bands around the waypoints. A good method could be implementing dynamic safety distance and dead-bands. This means that both will be variables that can be relative to the speed of the drone. So, if the speed is high the safety distance should be high as well since it will require the drone more time to be able to come to rest and avoid hitting the obstacle. While for the dead-bands if the velocity is high the dead-bands can be made larger.

REFERENCES

[1] Koenig, S. and Likhachev, M., "Incremental a. Advances in neural information processing systems," 14, 2001.

[2] Koenig, S. and Likhachev, M., "Improved fast replanning for robot navigation in unknown terrain," in *Proceedings 2002 IEEE international conference on robotics and automation (Cat. No. 02CH37292) (Vol. 1, pp. 968-975). IEEE.* , 2002, May.

[3] Stentz, A., "The focussed d* algorithm for real-time replanning," in *IJCAI (Vol. 95, pp. 1652- 1659)*, 1995.

[4] Hart, P.E., Nilsson, N.J. and Raphael, B., "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, 4(2), pp.100-107, 1968.

[5] Pearl, J., "Heuristics: intelligent search strategies for computer problem solving," *Addison-Wesley Longman Publishing Co., Inc*., 1984.

[6] Likhachev, M. a. K. S., "Lifelong Planning A*and Dynamic A* Lite," *The proofs. Technical report, College of Computing, Georgia Institute of Technology,Atlanta*, 2001.

[7] Maw, A.A., Tyan, M. and Lee, J.W., "Development of Flight Mission Planner using Intelligent Anytime Planning and Replanning Algorithm for UAV Operation," *(Doctoral dissertation, MERAL Portal)*, 2019.

[8] Ribeiro, M.I., "Kalman and extended kalman filters: Concept, derivation and properties," *Institute for Systems and Robotics*, 43, p.46, 2004.

[9] Mackay, D., "Path planning with d*-lite," *DRDC Suffield TM*, 242, 2005.

[10] Edelkamp, S., "Updating shortest paths," In *Proceedings of the European Conference on Artificial Intelligence, 655–659.* 1998.

[11] Frigioni, D.; Marchetti-Spaccamela, A.; and Nanni, U, "Fully dynamic algorithms for maintaining shortest paths trees," *Journal of Algorithms 34(2):251–281*, 2000.

[12] Thayer, S.; Digney, B.; Diaz, M.; Stentz, A.; Nabbe, B.; and Hebert, M, "Distributed robotic mapping of extreme environments," *In Proceedings of the SPIE: Mobile Robots XV and Telemanipulator and Telepresence Technologies VII, volume 4195*, 2000.

[13] Yahja, A.; Stentz, A.; Brumitt, B.; and Singh, S., "Framed quadtree path planning for mobile robots operating in sparse environments," *In International Conference on Robotics and Automation*, 1998.

[14] Moore, A., and Atkeson, C., "The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces," *Machine Learning* 21(3):199–233., 1995.

[15] Nilsson, N., "Problem-Solving Methods in Artificial Intelligence," *McGraw-Hill*, 1971.

[16] Stentz, A., and Hebert, M., "A complete navigation sys- tem for goal acquisition in unknown environments," *Autonomous Robots 2(2):127–145*, 1995.

[17] L. Lei, "Application research of intelligent optimization algorithm in UAV track planning," *Ship Electron. Eng., vol. 5, pp. 34-37,* 2017.

[18] A. Stentz, "The focused algorithm for real-time re-planning," *Proc. Int. Joint Conf. Artif. Intell., pp. 3310-3317,* 2000.

[19] S. Koenig, M. Likhachev and D. Furcy, "Lifelong planning A*," *Artif. Intell.*, vol. 155, no. 1, pp. 93-146, 2004.

[20] W. Zhangqi, Z. Xiaoguang and H. Qingyao, "Mobile robot path planning based on parameter optimization ant colony algorithm," *Procedia Eng., vol*. 15, pp. 2738-2741, 2011.

[21] T. Oral and F. Polat, "Lite: An incremental path planning algorithm taking care of multiple objectives," IEEE Trans. Cybern., vol. 46, no. 1, pp. 245-257, FEB. 2015.

[22] D. Ferguson and A. Stentz, "Algorithm for improved path planning and re-planning in uniform and non-uniform cost environments," Jun. 2005.

[23] X. Chen and D. Liu, "Three-dimensional trajectory planning of unmanned aerial vehicle during target movement using," Electron. Control, vol. 20, no. 7, pp. 1-5, 2013.

[24] Y. Lian, Y. Fan, H. Yu and Z. Yang, "Application of improved lite algorithm in virtual soldier path planning," *Modern Electron. Technol*., vol. 46, no. 6, pp. 23-27, 2018.

[25] S. Wang, L. Hu and Y. Wang, "Path planning of indoor mobile robot based on improved algorithm," *Comput. Eng. Des*., vol. 41, no. 4, pp. 1118-1124, 2020.

[26] H. Pan, "Research on autonomous path planning algorithm of mobile robot in complex environment", 2018.

[27] Aine, S. and Likhachev, M., "Truncated incremental search," *Artificial Intelligence* 234: 49–77, 2016.

[28] Hart, P.E., Nilsson, N.J. and Raphael, B., "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics* 4(2): 100–107, 1968.

[29] Hernández, C., Asín, R. and Baier, J.A., "Reusing previously found A* paths for fast goal-directed navigation in dynamic terrain," *19th AAAI Conference on Artificial Intelligence, Austin, TX, USA*, pp. 1158–1164, 2015.

[30] Hernández, C., Baier, J.A. and Asín, R., "Making A* run faster than D*-Lite for path-planning in partially known terrain," *Proceedings of the 24th International Conferenceon Automated Planning and Scheduling, Portsmouth, NH,USA*, pp. 504–508, 2014.

[31] Hernández, C., Meseguer, P., Sun, X. and Koenig, S., "Path-Adaptive A* for incremental heuristic search in unknown terrain," Proceedings of the 19th International Conference on Automated Planning and Scheduling, Thessaloniki, Greece, 2009.

[32] Hernández, C., Sun, X., Koenig, S. and Meseguer, P., "Tree Adaptive A*," *10th International Conference on Autonomous Agents and Multiagent Systems, Taipei, Taiwan*,Vol. 1, pp. 123–130, 2011.

[33] Likhachev, M., Ferguson, D.I., Gordon, G.J., Stentz, A. and Thrun, S., "Anytime Dynamic A*: An anytime, replanning algorithm," *Proceedings of the 15th International Conference on Automated Planning and Scheduling, Monterey, CA, USA*, pp. 262–271, 2005.

[34] Podsedkowski, L., "Path planner for nonholonomic mobile robot with fast replanning procedure," *1998 IEEE International Conference on Robotics and Automation, Lueven, Belgium, Vol*. 4, pp. 3588–3593, 1998.

[35] Podsedkowski, L., Nowakowski, J., Idzikowski, M. and Vizvary, "A new solution for path planning in partially known or unknown environment for nonholonomic mobile robots," Robotics and Autonomous Systems 34(2): 145–152, 2001.

[36] Trovato, K.I., "Differential A*: An adaptive search method illustrated with robot path planning for moving obstacles and goals, and an uncertain environment," *International Journal of Pattern Recognition and Artificial Intelligence* 4(2): 245–268, 1990.

[37] Trovato, K.I. and Dorst, L., "Differential A*," *IEEE Transactions on Knowledge and Data Engineering* 14(6): 1218–1229, 2002.

[38] Van Toll, W. and Geraerts, R., "Dynamically Pruned A* for replanning in navigation meshes," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Hamburg, Germany*, pp. 2051–2057 , 2015.

[39] Sturtevant, N.R., "Benchmarks for grid-based pathfinding," *IEEE Transactions on Computational Intelligence and AI in Games* 4(2): 144–148, 2012.

[40] WEB_1, (2018), Udacity, Udacity repository in Github, https://github.com/udacity/FCND-Controls-CPP/blob/master, 15/05/2018.