**University of South Eastern Norway**
Faculty of Technology and Maritime Sciences
-
Master Thesis in Systems Engineering with Embedded Systems
Kongsberg Department of Engineering
June 10, 2021

Biplav Karna

# UAV path planning in search and rescue (SAR) missions

# *Abstract*

Unmanned Aerial Vehicle (UAV) and Search And Rescue (SAR) missions are hot topics of research these days. UAVs come in a wide range with different capabilities. Due to their applicability, they are being used for trivial as well as complex missions. The frequent mishaps and disasters have brought SAR into the limelight. SAR missions have unfavorable environments and short response times. SAR operations utilize various resources. Among them, UAV is one of the key resources, and it plays a vital role in such missions. This thesis explores the path planning of UAVs in SAR missions. Some of the area coverage algorithms have been considered and simulated in SITL with PX4 flight software stack in Gazebo. Quadcopter has been considered for the area coverage algorithms. Post-flight analysis of logs has been presented. Comparison between these algorithms has been deduced. Rapidly exploring Random Trees (RRT) has been considered as a target reaching algorithm in the SAR mission. RRT has been implemented in a 3D environment, modeling UAV as a 3D figure. Biased and unbiased flavors of Rapidly exploring Random Trees (RRT) have been implemented, and their comparison is discussed.

# *Acknowledgements*

I want to express my gratitude to my supervisor Professor Antonio L. L. Ramos at the University of South-Eastern Norway (USN) for providing me this thesis topic and for his guidance throughout the process.

I am grateful to my co-supervisor Professor Paulo Rosa from the Military Institute of Engineering (IME), Brazil, for his inputs and suggestions to improve this thesis.

The collaboration with Professor Paulo Rosa was made possible owing to the Branortech project (https://app.cristin.no/projects/show.jsf?id=2488728), a collaboration between USN and Norwegian University of Science and Technology (NTNU), plus IME and the Federal University of Rio de Janeiro (UFRJ) in Brazil. This project is co-funded by the Norwegian Agency for International Cooperation and Quality Enhancement in Higher Education (Diku) and the Coordination for the Improvement of Higher Education Personnel (CAPES), Brazil.

I am thankful to USN for the opportunity to pursue my masters. I am very thankful to the academic staff at USN who tried their best to share their experience and knowledge with the students.

I am grateful to my friends at USN, Sandeep Shivakoti, Leila Mozaffari, Victor Johan Hansen, and Deivydas Kazokas. Their presence has helped me in many subtle ways, which I can't express in words. I especially want to thank my friend Ramesh Timsina for all the support and motivation. I am indebted to my family for their love and support. I want to thank my cousin, Saurav Kantha, for the discussions.

Finally, a special thanks to the open-source community without which this work would not be possible.

Biplav Karna
Kongsberg, Norway, June 10, 2021

# Contents

# List of Figures

# List of Tables

# Acronyms

**2D** 2-Dimensions. 2, 9, 52, 53

**3D** 3-Dimensions. i, 2, 4, 9, 18, 19, 28, 45, 52, 53

**A*** A Star. 2, 8, 9

**API** Application Programming Interface. 31, 34

**CCW** Counter Clock-wise. 16, 25

**CG** Centre of Gravity. 14

**CW** Clock-wise. 14, 16, 25

**DOF** Degree of Freedom. 14

**DTAM** Dense Tracking And Mapping. 2

**EASA** European Union Aviation Safety Agency. 7

**EU** European Union. 7

**FastRTPS** Fast Real Time Publish Subscribe. 31

**GNC** Guidance Navigation and Control. v, 4–6, 31

**GPS** Global Positioning System. 5, 11, 31, 34, 37

**GUI** Graphical User Interface. 30, 34

**HITL** Hardware In The Loop. v, 31, 33, 34

**HTOL** Horizontal Take Off and Landing. 5

**IMU** Inertial Measurement Unit. 31, 37

**INS** Inertial Navigation System. 5

**LSD-SLAM** Large-Scale Direct Monocular SLAM. 2

**MAVLink** Micro Air Vehicle Communication Protocol. 34

**PID** Proportional Integral Derivative. 15

**PTAM** Parallel Tracking And Mapping. 2

**RAM** Random Access Memory. 30

**RC** Remote Control. 34

**ROS** Robot Operating System. 30

**RRT** Rapidly exploring Random Trees. i, v, 2, 3, 8, 28, 29, 45, 47, 49, 52

**SAR** Search And Rescue. i, 3, 4, 28, 49, 52, 53

**SITL** Software In The Loop. i, v, 3, 31, 33–36, 52

**SLAM** Simultaneous Localization And Mapping. 53

**TCP** Transmission Control Protocol. 31

**UART** Universal Asynchronous Receiver Transmitter. 31

**UAV** Unmanned Aerial Vehicle. i, iv, v, 2–8, 11, 14–16, 18, 21, 28, 31, 34, 36–38, 41, 45, 47, 49, 52, 53

**UDP** User Datagram Protocol. 31, 34

**USB** Universal Serial Bus. 31

**VTOL** Vertical Take Off and Landing. 5

# Chapter 1

# Introduction

FIGURE 1.1: A glimpse of Nepal earthquake 2015. *Picture Credit: Daniel Berehulak for New York's Time.*

Unmanned Aerial Vehicle (UAV), also popularly known as a drone, has become a hot topic in current days. Its application is being explored in a wide range of areas. The recent progress in technology has led to the development of UAVs with sizes from few inches to 80 feet in length. Various types of advanced sensors have enhanced the capabilities of UAVs for being autonomous and efficient. The advancement in battery technologies has made long flight time possible. In various ways, autonomous UAVs can benefit in search and rescue missions. Surveying an area, searching for persons and items of interest, localizing and mapping of the area, reaching out to the places where humans can't reach easily are some applications where UAVs can be handy in SAR missions.

## 1.1 Background and Motivation

The first UAV dates back to 1783, a hot air balloon which demonstrated unmanned aircraft [6]. In 1849, Austria used unmanned balloons to attack Venice. It was the first military use of UAV [7]. The first modern UAV was developed by UK Royal Air Force in 1935[8]. Now it's being used for multiple purposes like cinematography, agriculture, toys, advertisement, military and defense, search and rescue, recreation, and others[9]. The ability of the latest drone to maneuver efficiently and carry enough load makes it suitable for search and rescue missions. Today, autonomous UAVs and the collective work of UAVs are possible due to the achievements in artificial intelligence.

Search and rescue (SAR) operations are carried out after natural disasters, catastrophes, accidents, mishaps, and other incidents. Most of the countries which have access to the ocean have search and rescue squad. The Maritime sector exercises frequent search and rescue operations. The first SAR operation that was well documented was a maritime incident, where it was carried out after the wreck of the Dutch merchant ship Vergulde Draeck near Australia in 1656 [10]. Incidents in which SAR operations are required occur frequently. The search areas are usually large, having unfavorable terrain, and sometimes difficult to access.

UAVs are being used for SAR operations for surveying the area, tracking the lost person. UAVs are used alongside other machines and humans for the mission. UAV can also be used for creating a map of the area where the SAR operation takes days, like in earthquakes, tsunamis, etc. Planning the path for UAV is crucial in such situations.

## 1.2 Current State of the Art

A UAV has to avoid the obstacles and maneuver in the free space utilizing the least of resources for successful completion of a mission. The planning of this maneuverability of a UAV or multiple UAVs is path planning. Path planning determines the direction and length of path segments and their interconnection. The dynamics parameters like thrust, speed, acceleration, etc are not emphasized in path planning, unlike motion planning and trajectory planning. The path planning algorithms are designed for 2D and 3D environment. The many 2D algorithms cannot be extended for 3D environment. The path determination for 3D is NP-hard[11] [12]. This means no algorithm is efficient to solve the problem in polynomial time. There is great scope of research in path planning in 3D environment.

Till recent days, many path planning algorithms have been designed which are applicable to platform like UAV. The path planning algorithms are associated with the type of objective of the mission. The objectives are localizing and mapping the area [13], area coverage [14], and reaching the target point. Back and forth [15], sector search [15], spiral [15], barrier patrol [15], genetic algorithm [16], and chaotic ant colony optimization to coverage [17] [18] are algorithms for area coverage. Dijkstra[19], A* [20] and its variants, and RRT[21] and its variants are algorithms used to reach the target point. MonoSLAM[22], PTAM[23], DTAM[24], LSD-SLAM[25] are algorithms for simultaneous localization and mapping.

## 1.3 Objective

The main objective of this thesis is to study existing path planning algorithms that can be applicable to UAV in the SAR mission. The focus is on algorithms related to coverage of the region of interest and non-deterministic path planning to reach the target. The quadcopter model of UAV is taken into consideration to simulate and implement the algorithms. The aim is to simulate the algorithms in Software In The Loop (SITL) mode and provide the statistical analysis. The detection of obstacles and perception of environment with the sensors are not in the scope of this work. The tasks involved in this work are:-

- Literature review on current achievement in path planning algorithms
- Literature review on UAV and SAR
- Research on available simulation tools for UAV
- Implementation of algorithms relevant to SAR mission, using frameworks and tools
- Analysis of simulation

## 1.4 Contributions

Selected coverage path planning algorithms and target reaching algorithms were studied for applicability in SAR missions. Parallel line, creeping line, spiral long edge first, and spiral short edge first path planning algorithms were implemented with regard to region coverage. The implementations of these algorithms were simulated using the Gazebo simulator with a quadcopter and PX4 [4] software stack in SITL. The logs of these algorithms were analyzed after the completion of the missions. This thesis provides the comparison of these algorithms based on the logs. In this work, the RRT algorithm was implemented with regard to the target reaching algorithm. Two versions of RRT have been implemented, one without bias and the other with bias. Modeling of environment and UAV for the RRT is discussed. Comparison of biased and non-bias RRT is drawn.

## 1.5 Outline

The remainder of this document is formatted as follows. Chapter 2 provides the background of SAR, UAV and UAV path planning. Chapter 3 presents reference of frames, UAV maneuverability and graph theory. Chapter 4 provides the details of algorithms selected for simulation and evaluation. Chapter 5 documents the description of tools and the software used for implementing the algorithms. Chapter 6 provides the implementation and results of the simulation. Chapter 7 concludes the thesis with observation and future works.

# Chapter 2

# Background

This chapter provides the background of UAV, Search And Rescue (SAR), and path planning in general. It provides a description of the uses of UAVs, their classification, GNC, and regulations and safety. Various phases of SAR are mentioned. Various aspects and strategies of path planning are discussed here.

## 2.1 Unmanned Aerial Vehicle

An Unmanned Aerial Vehicle (UAV) is an aircraft that flies and carries out tasks without a human pilot inside it. UAVs are most commonly referred to as drones. Its operation is carried out by remotely based humans via the controller or autonomously via intelligent software. UAVs are used in agriculture, surveillance, cinematography, recreation, military and defense, search and rescue, advertisement, and others [9]. Based on the market, the sector of use can be broadly classified into categories *toy*, *hobby*, *professional*, *commercial*, and *military* [26].

Children are the primary target for *toy* drones. These are low-cost drones having a minimal number of sensors. These are used for recreational activities by kids.

In the *hobby* sector, hobbyists and enthusiasts use drones for capturing outdoor activities and sports. Videos of such activities shot from drones are posted by many professionals and amateurs athletes on multiple video streaming websites as well as on social websites. These drones have an emphasis on ease of control and image and video capturing capabilities.

Cinematography, surveillance, and agriculture are the *professional* sectors of the drone market. In cinematography, images from various angles are needed. UAV can take images from a wide range of angles. It can be used to take wide-angle image frames, where a large area needs to be in one frame. Using optical zoom of camera and moving drone far and near, provide a good way to zoom in and zoom out the scenes. Multiple drones are used in an array to create 3D videos and images. The use of drones in cinema has made the experience better and lively. When it comes to large area surveillance, drones are preferred. The forest, crop fields, and critical areas are monitored round the clock by drones. In recent days drones are used to monitor huge public gatherings, mobs, and public demonstrations. In the agriculture sector, apart from surveillance, drones are used for the dispersal of seeds, watering the field, and spraying insecticides and pesticides. DJI Agras-T16 [27] and DJI MG-1S [28] are drones made for agriculture by DJI.

In *commercial* space, custom-designed UAVs are manufactured for special purposes. It also includes the services associated with it. It includes UAVs for providing internet to remote areas, UAVs as telecommunication fronthaul, etc. Amazon's prime air[29] service delivers items by drones. The items are bought by customers online, and the items are picked from the warehouse and delivered to the customer's address by autonomous drones. The service was started on December 7, 2016. In 2016, Facebook experimented with a solar-powered drone as an atmospheric satellite to provide internet to remote places [30]. The drone named Aquilla acted like a relay for ground stations and communicated through laser beams.

UAVs used in the *military* sector are for combat and defense purposes, usually in a swarm of drones configuration. These are used for monitoring, supporting ground forces, and attack the targets. These carry ammunition like missiles and bombs with them. MQ-9 Reaper [31] used by U.S. Air Force and Bayraktar TB2 [32] used by Turkish Air Force are popular combat drones. These drones are partially autonomous, i.e., require a human operator for missions.

### 2.1.1 UAV Classification

UAVs come in various shapes and sizes, with different flying mechanisms and payload capacity. UAVs can operate with different levels of autonomy. The combination of these factors and capabilities are used to design UAVs for applicability in specific projects. UAVs can be categorized using different characteristics as a basis.

The classification of UAVs based on the flying mechanism is shown in Figure 2.1. *Fixed-wing* UAVs have large wings like an aircraft and need a runway to take off and land. This mechanism of flight is termed Horizontal Take Off and Landing (HTOL). These can glide in the air and have a higher speed than *rotorcraft*. Their performance in turning with different angles is not as par to *rotorcraft*.

*Rotorcraft* UAVs utilizes rotor blades for take-off and landing. These can take-off and land vertically and don't require a runway. This mechanism is known as Vertical Take Off and Landing (VTOL). They have very good maneuverability with different angles in 3-dimensional space. They can hover over a point with stability. These have less fuel/energy efficiency and have a short flight range. Their speed is comparably less than other types. They are further classified into *helicopter* and *multi-rotor* UAVs. *Helicopter*, also known as a *single-rotor* UAV, has a single large rotating blade for thrust. *Multi-rotor* UAVs have multiple rotors, and generally, they come with 4 rotors (quadrotor), 6 rotors(hexarotor), 8 rotors(ocatarotor), and 12 rotors (dodecarotor).

### 2.1.2 Guidance, Navigation and Control

Guidance, navigation, and control (GNC) are three aspects that every UAV posses for smooth functioning during flight. A high-level concept of operation is presented in Figure 2.2.

*Navigation* infers to moving in a stipulated path with stipulated motion parameters. UAV requires a three-dimensional co-ordinate system to navigate in an environment. The angles made by UAV with the x-axis, y-axis, and z-axis are roll, pitch, and yaw respectively. The time varying factors like speed and acceleration along the 3 axes are also required for navigation. The navigation system utilizes Global Positioning System (GPS), Inertial Navigation System (INS), and gyro sensors to determine these parameters.

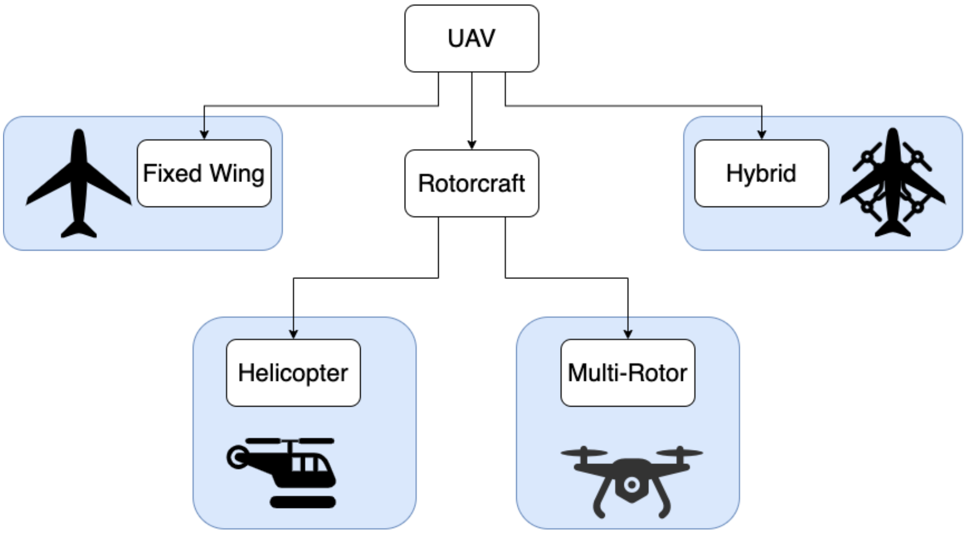



FIGURE 2.1: UAV classification based on flying mechanism.

*Guidance* ensures that the UAV is in the expected state and follows the planned path during the mission. The guidance system in UAV continuously monitors the navigational parameters against the expected path and motion during the course of the flight. If any deviation is detected, corrective control signals are forwarded to the control system.

*Control* means controlling the thrust, elevation, speed, angles, etc., to have desired motion. The Control system receives the information from the guidance system and translates to the signals to the actuators and sensors.

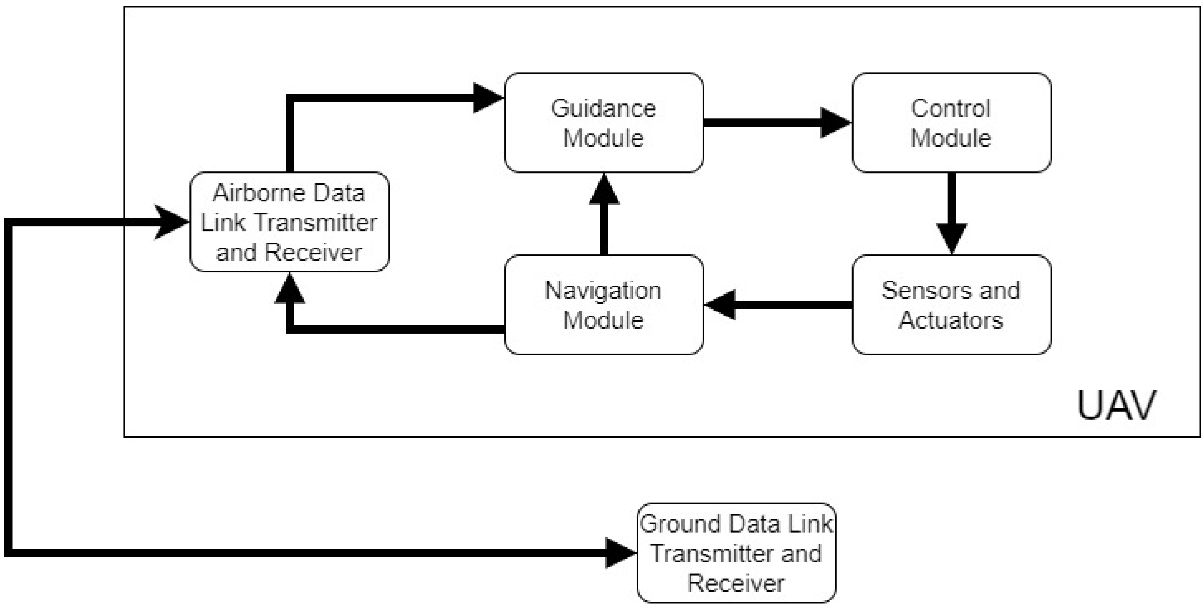



FIGURE 2.2: Concept of operation of GNC in UAV based on [1].

### 2.1.3 Regulation and Safety

UAVs are regulated by a government body, and may vary from country to country.

In Europe, European Union Aviation Safety Agency (EASA) drafts the regulation for UAVs. EASA categorizes UAVs into *open*, *specific*, and *certified* categories [33]. UAVs in the open category don't require any authorization or certification. These are mostly lightweight drones used for personal use and are operated within the area of sight. Specific category UAVs require operational authorization from the related authority before use. Certified category UAVs need to get certification according to the regulation framed in [33]. The minimum age of a remote pilot is 16 years [33]. Each member country in EU, decides the geographical zones and altitude of flight of UAVs[33].

## 2.2 Search and Rescue Missions

Search and rescue (SAR) refers to the operations which are carried out to search for, and provide aid to, persons, or things which are, or are feared to be, in affliction or imminent danger [34]. Such operations are generally carried out after catastrophes, disasters, accidents, mishaps, etc. Ground personals, vehicles, naval vessels, dogs, aircrafts, ground robots, and UAVs are the resources used for SAR. The area of operation can be underwater, underground (e.g. cave, tunnels), on water, and on the ground.

UAVs are used in SAR operations along with on-ground systems and personnel. Used UAVs are either autonomous or teleoperated. In most cases, UAVs' role in SAR is limited to search for the person and to provide supplements. UAV path determination is crucial in these tasks to ensure people are rescued as soon as possible. SAR operations are carried out in various environments like combat, maritime, low lands, cave, and mountain areas. Based on the areas of exploration, the challenges of the mission differ.

The International Maritime Organisation (IMO) categorized SAR operations in stages as follows [35]:

**Awareness Stage:** Local rescue body is informed about the incident in the awareness stage.

**Initial Action Stage:** Information about the incident is gathered, and the degree of emergency is evaluated.

**Planning Stage:** A comprehensive plan is laid out for the mission.

**Operations Stage:** The plan is executed in the operations stage.

**Conclusion Stage:** Mission is concluded with a report.

## 2.3 Path Planning

As mentioned in Section 1.2, path planning can be broadly categorized into three categories, i.e., localizing and mapping, coverage of the area, and reaching the target. The problem of path planning, in general can be put into hierarchical levels [2], as shown in Figure 2.3. A platform is any system that is used in a mission. It can be any type of UAV or other robot. The first level deals with the degree of freedom and dynamics of the platform.

The first level is divided into 3 types:

- **Holonomic**

  If all degrees of freedom are controllable, then the platformm is called holonomic. The holonomic constraints depend on the position parameters and time but not on time derivative parameters like speed.

- **Non-Holonomic**

  The platform for which the time derivative parameters are the constraints for movement is termed non-holonomic. Cars and fixed-wing UAVs are non-holonomic platforms.

- **Kinodyamics**

  The problem which has kinematics and dynamics constraints fall under this category.

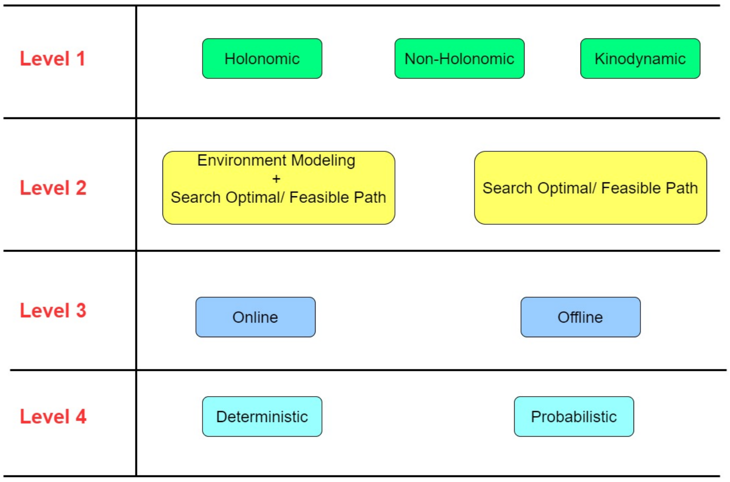



FIGURE 2.3: Path planning levels based on [2].

The second level addresses the approach of finding the path. Path planning algorithms like Dijkstra[19] and A* [20] require environment modeling to find solutions. This means the information of the environment is required prior to finding the path. Algorithms like RRT[21] don't need environment modeling priorly.

The third level is about the architecture of the path planning system. The computation of path planning can be online, i.e., remotely on a server or offline, i.e., on the platform. Some algorithms perform better online and some perform better offline.

The fourth level differentiates the nature of traversal by the platform in the environment. In the deterministic approach, the paths and waypoints are pre-determined for a platform. Dijkstra[19] and A* [20] are deterministic approach. In the probabilistic approach, the paths and waypoints are dynamic and are determined during the mission by sensing the environment and sharing the information. RRT[21] is one of the probabilistic algorithms.

### 2.3.1 Strategies

Solving path planning is a challenging task. Various strategies have been developed over a period of time to tackle it and find solutions that are feasible and have convergence. There are strategies for environment modeling, the number of platforms used, and modes of communication. The algorithms of path planning make use of one of the strategies or a combination of them.

The environment of the mission is limited by the region of interest. The mission is carried out within the boundaries of this region. The region of interest and obstacles are represented by enclosed planar shapes in 2D and enclosed 3D shapes and meshes in 3D. The strategies for environment modeling are:

- **No Decomposition**[14]

  The region of interest is usually non-complex 2D or 3D shapes. It is not further decomposed into smaller grids. This modeling is suitable for algorithms using one platform like Back and forth[15, 35] and A* [20].

- **Regular Grids**[2]

  The region of interest is partitioned into smaller sections having equal area/volume. In 2D, generally used grids are triangular, rectangular, square, and hexagon. Other polygons can be used as well. In 3D, the grids used are a cartesian grid and rectilinear grid.

- **Exact Cell Decomposition**[36]

  Parallel lines are drawn from the vertices of the obstacles to the boundary of the region. Each cell is given a number, and a connectivity graph is created which represents the adjacency of the cells. In the case of the region of the concave shape, the decomposition makes the region into convex shaped cells which are easier to deal with.

- **Approximate Cell Decomposition** [36]

  The region is divided recursively with varying dimensions until the cell is completely in free space or in obstacle space or until the limit of cell dimension is reached. In 2D, Quadtree[37] technique is used to form the irregular grids, while Octree[38] is used in 3D.

- **Roadmap Approach** [36]

  Each vertex of each obstacle is connected via lines to all other vertices of all obstacles without crossing the interior of any. The connected lines represent the set of the free path. Visibility Graph[39] and Voronoi Diagrams[40] are types of roadmap approach.

- **Occupancy Map**

  It is similar to the above decomposition strategies. Here as well region is divided into small cells, except the cells are associated with the probabilistic value of occupancy instead of deterministic.

A single platform is sufficient for some approaches, while some require numerous platforms. Based on the number of platforms, the strategies can be:

- **Mono-system**

  A single platform is used for the mission.

- **Homogenous system**

  Multiple platforms of the same type are used. The platforms' co-ordination is taken into consideration for designing algorithms.

- **Heterogenous system**

  Multiple platforms of various types are used. A particular type of platform may be able to do only some specific tasks. Coordination of multi-type platforms and task assignment to the platform according to capabilities are taken into consideration.

Communication plays an important role in path planning. The platform may operate autonomously without communicating to ground control. In a multi-platform system, the platforms may have to communicate with each other. The strategies of communication are:

- **No Communication**

  A platform would get a mission at the start point and have to complete it without communicating in between with other systems. The platform can't get any instructions like a change of plan or change of path during the execution of an ongoing mission. The environments where there is a lack of communication means require such strategies.

- **Communication with ground control**

  A platform periodically communicates with ground control during the execution of a mission. The updates of mission are sent to ground control, and ground control can send command to change the mission in between.

- **Inter platform communication**

  In the case of a multi-platform system, the platforms may communicate to each other either directly or via some relays. Communication is required to co-ordinate and efficiently carry out the mission.

# Chapter 3

# UAV Navigation and Path Planning

This chapter discusses the theoretical aspects involved with UAV and path planning. Different frame of references and co-ordinate systems are covered. Theory related to UAV maneuverability and path planning has been covered.

## 3.1 Frame of Reference

There are various co-orindate systems used in aircraft design and analysis[41]. The co-ordinate system can be categorized into the following [42].

- Geodetic co-ordinate system
- Earth-centered earth-fixed (ECEF) co-ordinate system
- Local north-east-down (NED) co-ordinate system
- Vehicle carried north-east-down (NED) co-ordinate system
- Body co-ordiante system

One or more co-ordinate systems are required for maneuverability of UAV.

### 3.1.1 Geodetic Co-ordinate System

The geodetic co-ordinate system is used in GPS widely. This system used latitude ($\phi$), longitude ($\lambda$), and height (h) (or altitude) to locate the point near the earth's surface. The longitude ranges from -180° to 180°, which is measured from Prime Meridian. The latitude ranges from -90° to 90°, which is measured from the equator of the earth. The latitude($\phi$) of any point in the geodetic system is the angle between the perpendicular line drawn to the surface of the earth from the point and the equatorial plane, which is different from geocentric latitude. In a geocentric system, the latitude($\phi'$)is the angle between the line passing through the point and centre of the earth, and the equatorial plane. Figure 3.1 shows the difference between ($\phi'$) and ($\phi$). The coordinate vector of the geodetic frame is expressed as

$$P = \begin{pmatrix} \phi \\ \lambda \\ h \end{pmatrix}.$$

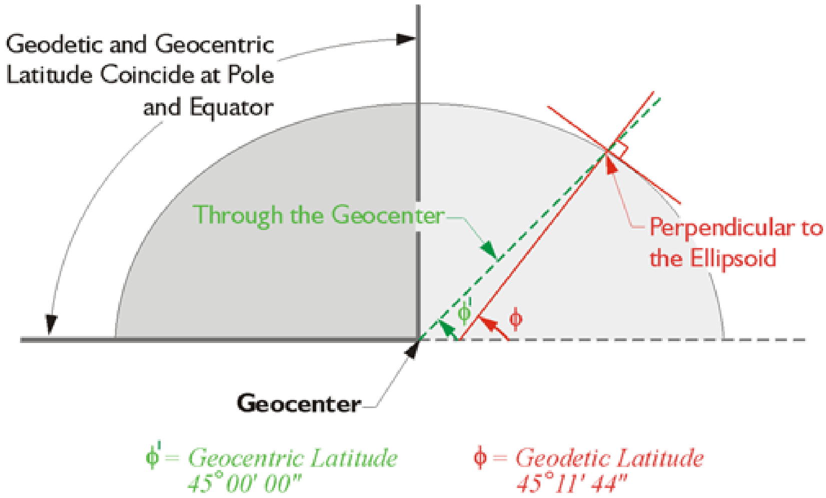



FIGURE 3.1: Geocentric and Geodetic co-ordinate system. *Source:GPS For Land Surveyors* [3].

### 3.1.2 Earth Centered Earth Fixed (ECEF) Co-ordinate System

ECEF is a geocentric co-ordinate system, and its origin is the earth's centre of mass. The x-axis passes through 0° latitude i.e., equator, and 0° longitude, i.e., Prime Meridian. The y-axis is perpendicular the to x-axis in the CCW direction. The z-axis is towards the true north direction. Figure 3.2 shows the x,y,z ECEF co-ordinate system. A vector in this frame is represented as

$$P_e = \begin{pmatrix} x_e \\ y_e \\ z_e \end{pmatrix}.$$

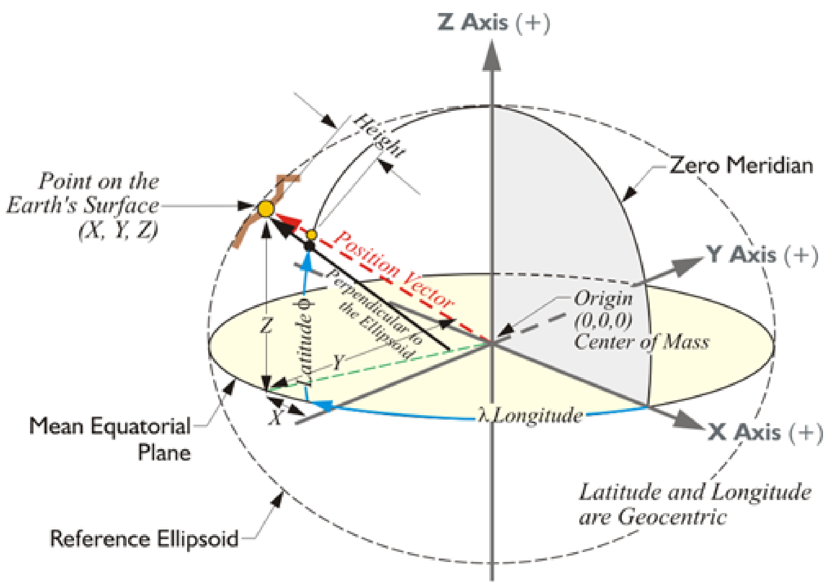



FIGURE 3.2: Earth-centered earth-fixed (ECEF) co-ordinate system. *Source:GPS For Land Surveyors [3].*

### 3.1.3 Local East, North, Up (ENU) Co-ordinate System

In this frame, east is the x-axis, north is the y-axis, and up away from the earth centre is the z-axis.

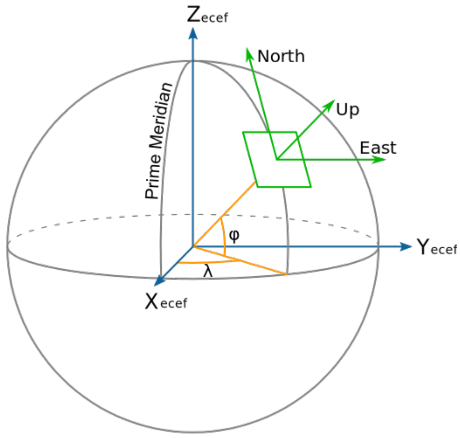



FIGURE 3.3: Local east, north, up (ENU) co-ordinate system. *Source:GPS For Land Surveyors [3].*

### 3.1.4 Local North, East, Down (NED) Co-ordinate System

Local NED frame is used in most aircraft systems. The north is the x-axis, east is the y-axis, and down towards the ellipsoid normal is the z-axis. The origin is fixed arbitrarily at one point on the surface of the earth. The x and y axes are on the tangent plane to the origin on the surface of the earth. A vector of a point in this frame is represented as

$$P_n = \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix},$$

whereas the height is represented as

$$h = -z_n.$$

### 3.1.5 Vehicle Carried North East Down (NED) Co-ordinate System

This frame is associated with the vehicle and carried by it. The origin is the center of gravity of the vehicle. The geodetic (ellipsoid) north is the x-axis, the geodetic (ellipsoid) east is the y-axis, and the z-axis is downwards along the ellipsoid normal. The vector of the point in this frame is denoted as

$$P_{nv} = \begin{pmatrix} x_{nv} \\ y_{nv} \\ z_{nv} \end{pmatrix}.$$

### 3.1.6 Body Co-ordinate System

The body co-ordinate system is on the body of the aircraft vehicle and is carried with it. The origin is located at CG of the aircraft vehicle. The x-axis is towards forward flying direction. The y-axis is perpendicular to x-axis in the CW direction. The z-axis is pointing downwards.

## 3.2 UAV Maneuverability

UAV maneuverability is a complex task. UAVs have 6 degrees of freedom (DOF) i.e.movement along x, y, and z axes, and rotation around those axes. If centre of gravity of UAV is considered as origin, the x-axis is along the forward movement direction, the y-axis is perpendicular to the x-axis towards the right or left direction, and the z-axis is perpendicular to the both towards up or down direction. Roll, pitch, and yaw are rotational angles around the x, y, and z axes, respectively. Figure 3.4 shows the axes and rotational angles. UAVs have some constraints on their degree of freedom, e.g., fixed wind UAVs can't take 90° turns.

A homogenous co-ordinate system is used for the maneuverability of UAVs. In this system, UAV has its own local co-ordinate system, and the environment has a different co-ordinate system, as shown in Figure 3.4. The relation between these co-ordinates is developed, which helps in the transformation from one co-ordinate system to other.

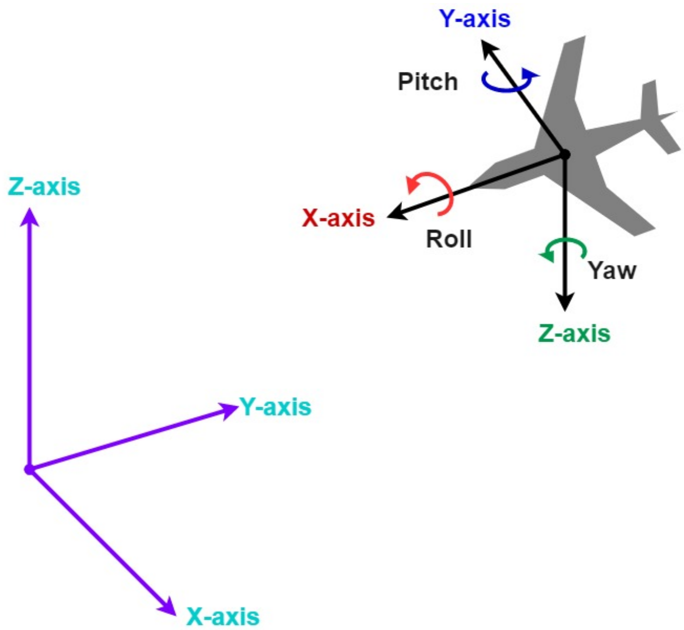



FIGURE 3.4: Co-ordinates system of UAV and ground.

Let $x, y, z$ be co-ordinates with reference to ground co-ordinate system, and $u, v, w$ be co-ordinates with reference to UAV. $P_g$ be any point represented w.r.t ground, $P_f$ be same point represented w.r.t. UAV, shown in Equation 3.1. $P_f$ point can be transformed to $P_g$ using transformation matrix ${}^gT_f$ as shown in Equation (3.2). ${}^gR_f$ is a $3x3$ rotational matrix, and ${}^gD_f$ is a $1x3$ translational matrix. The subscripts of $\theta$ in Equation (3.4), represents the angle between those axes. The translational matrix in Equation (3.5) has translational displacement between the corresponding axes of two co-ordinate system.

$$P_g = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, P_f = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} P_g \\ 1 \end{bmatrix} = {}^gT_f \begin{bmatrix} P_f \\ 1 \end{bmatrix} \tag{3.2}$$

$$\begin{bmatrix} P_g \\ 1 \end{bmatrix} = \begin{bmatrix} & {}^gR_f & & {}^gD_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_f \\ 1 \end{bmatrix} \tag{3.3}$$

$${}^gR_f = \begin{bmatrix} \cos\theta_{xu} & \cos\theta_{xv} & \cos\theta_{xw} \\ \cos\theta_{yu} & \cos\theta_{yv} & \cos\theta_{yw} \\ \cos\theta_{zu} & \cos\theta_{zv} & \cos\theta_{zw} \end{bmatrix} \tag{3.4}$$

$${}^fD_g = \begin{bmatrix} D_{xu} \\ D_{yv} \\ D_{zw} \end{bmatrix} \tag{3.5}$$

Combining equations (3.1), (3.2), (3.4), and (3.5) leads to Equation (3.6). $P_g$ point can be transformed to $P_f$ using the inverse of transformation matrix ${}^gT_f$ as shown in Equation (3.7).

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta_{xu} & \cos\theta_{xv} & \cos\theta_{xw} & D_{xu} \\ \cos\theta_{yu} & \cos\theta_{yv} & \cos\theta_{yw} & D_{yv} \\ \cos\theta_{zu} & \cos\theta_{zv} & \cos\theta_{zw} & D_{zw} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} \tag{3.6}$$

$$\begin{bmatrix} P_f \\ 1 \end{bmatrix} = {}^gT_f^{-1} \begin{bmatrix} P_g \\ 1 \end{bmatrix} \tag{3.7}$$

### 3.2.1 PID Control

Proportional Integral Derivative (PID) control is a technique of control theory used to achieve desired response in a system. It is widely used in numerous systems and in UAVs as well. It is used to drive the actuators with the desired value. It is a closed feedback loop control system, as shown in Figure 3.5. Proportional, integral, and derivative components are the three main components in it. This control system sits between the generated signal $r(t)$ and the actuator. The generated signal is converted to control signal $u(t)$ and fed to the actuator. The output of the actuator $y(t)$ is fed back to control system to determine the control signal. Equations 3.8 and 3.9 show the mathematical relations between the signals shown in Figure 3.5.

$$u(t) = K_p * e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt} \tag{3.8}$$

$$e(t) = r(t) - y(t) \tag{3.9}$$

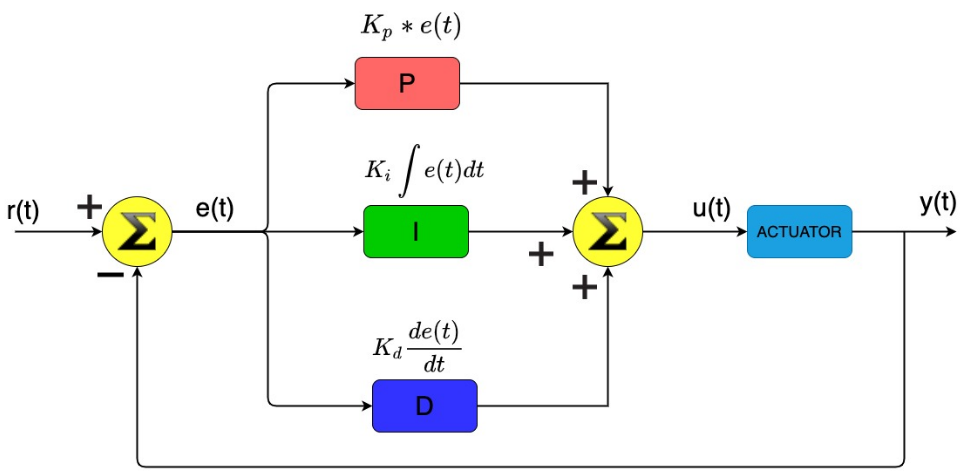



FIGURE 3.5: PID control Diagram.

- Proportional

  The proportional term $K_p * e(t)$ in Equation 3.8 is to minimize the error. The proportional gain $K_p$ determines the response to the error. If it is too high, the response will oscillate with high frequency. If it is too low, the response will change slowly to reduce the error.

- Integral

  The integral term $K_i \int e(t)dt$ in Equation 3.8 keeps the memory of errors and sums them. The term minimizes the steady-state error. A high value of integral gain, $K_i$, leads to oscillation [43].

- Derivative

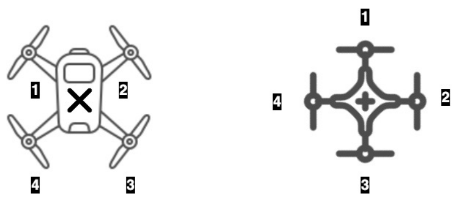
  The derivative term $K_d \frac{de(t)}{dt}$ in Equation 3.8, improves the transient response of the system [43]. The derivative gain $K_d$ is the dampening factor and prevents the overshooting of the signal.

### 3.2.2 Quadcopter Maneuverability

A quadcopter is a multi-rotor type of UAV which has 4 rotors on it. Figure 3.6a shows a quadcopter, a market product from DJI company. Quadcopters come in two orientations, "+" and "x" orientations as shown in Figure 3.6b. The rotors are attached to the four corners. The alternative rotors have opposite direction of spin. In Figure 3.6b, rotors are numbered from 1 to 4. If rotor 1 rotates CW direction for upward thrust, rotor 2 rotates in CCW, rotor 3 in CW, and rotor 4 in CCW directions. The opposite rotation of the alternative rotor makes the total angular momentum of the quadcopter zero when all rotors rotate with the same speed. This helps to hover at a constant height and maintains stability. In "+" orientation, the quadcopter's front has one rotor, while in "x" orientation, there are two rotors at front. Based on the orientation, the mechanism of maneuverability varies.

Figure 3.7 shows the speed and direction of rotors for various movements in a quadcopter with "+" orientation. The rotors are numbered from 1 to 4, where 1 is the front of the quadcopter. The speeds of the rotors are represented by color-coding. The red color represents high speed, blue the slow speed, and green the normal speed.

(a) DJI Phantom 4 Pro V2. : *Taken from DJI website[44].*




(b) Quadcopter Orientations.

*Left: "$x$", cross orientation; Right: "$+$", plus orientation.*

FIGURE 3.6: Quadcopter and its orientations.

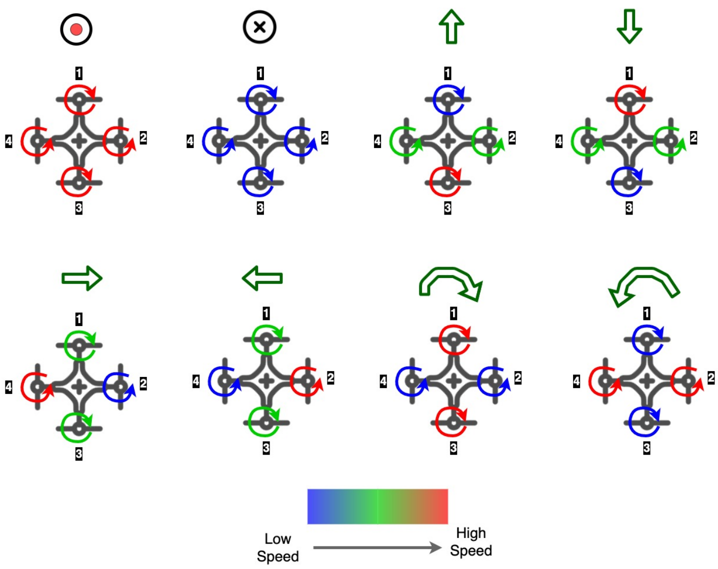



FIGURE 3.7: Variation of rotors' spins for movement of Quadcopter.

*From top left clockwise: lift, land, forward, backward, right, left, rotate right, rotate left.*

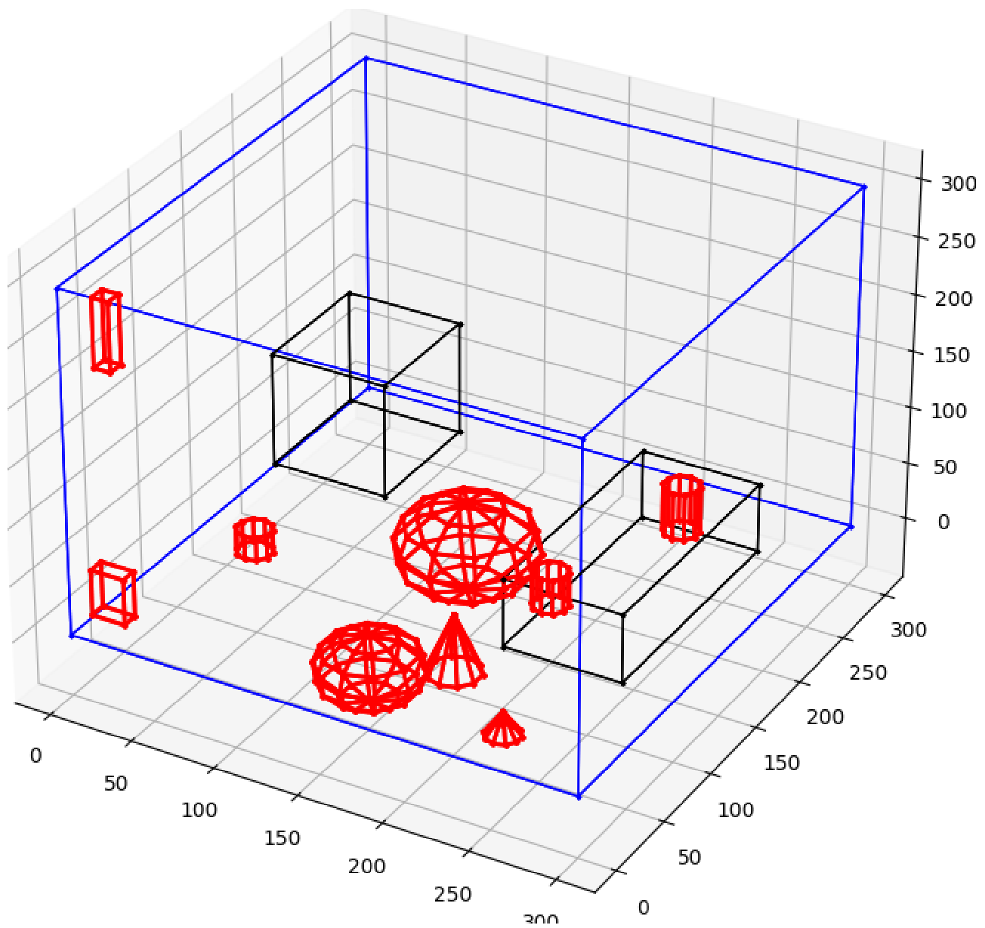



FIGURE 3.8: Region of interest with obstacles and no fly zones.

## 3.3 Path Planning

Path planning has a region of interest where the mission has to be carried out. A region of interest has a boundary, obstacles, access prohibited region, and free region. UAV access prohibited region is called the no-fly zone and free region as the fly zone. Figure 3.8 shows a region of interest in 3D, with boundary in blue, obstacles in red, and no-fly zone in black. 3D figures can be represented as a set of polygons, and polygons can be represented as a set of points.

$Polygon = \{pt_1, pt_2, pt_3, ...., pt_n\}$ where $pt$ is a point.

$Figures_{3D} = \{pl_1, pl_2, pl_3, ...., pl_n\}$ where $pl$ is a polygon.

Region of interest, obstacles, no-fly zones, and free space can be represented as follows:

Region of Interest $(Roi) = \{Bd_1, Bd_2, Bd_3, ...., Bd_{bn}\}$, where $Bd$ is boundary polygon, $bn$ is number of boundary polygons.

Obstacles $(Obs)$ $=\{Ob_1, Ob_2, Ob_3, ..., Ob_{on}\}$, where $Ob$ is 3D obstacle, $on$ number of obstacles.

No-fly zone $(Nfz) = \{Nf_1, Nf_2, Nf_3, Nf_4, ..., Nf_{nn}\}$, where $Nf$ is no-fly region, $nn$ number of no-fly region.

Fly zone $(Fz) = Roi - \{Obs \cup Nfz\}$

UAV fly in the fly zone between waypoints. Waypoints are the points where it stops or pauses, or takes a turn. The path consists of waypoints and routes between them.

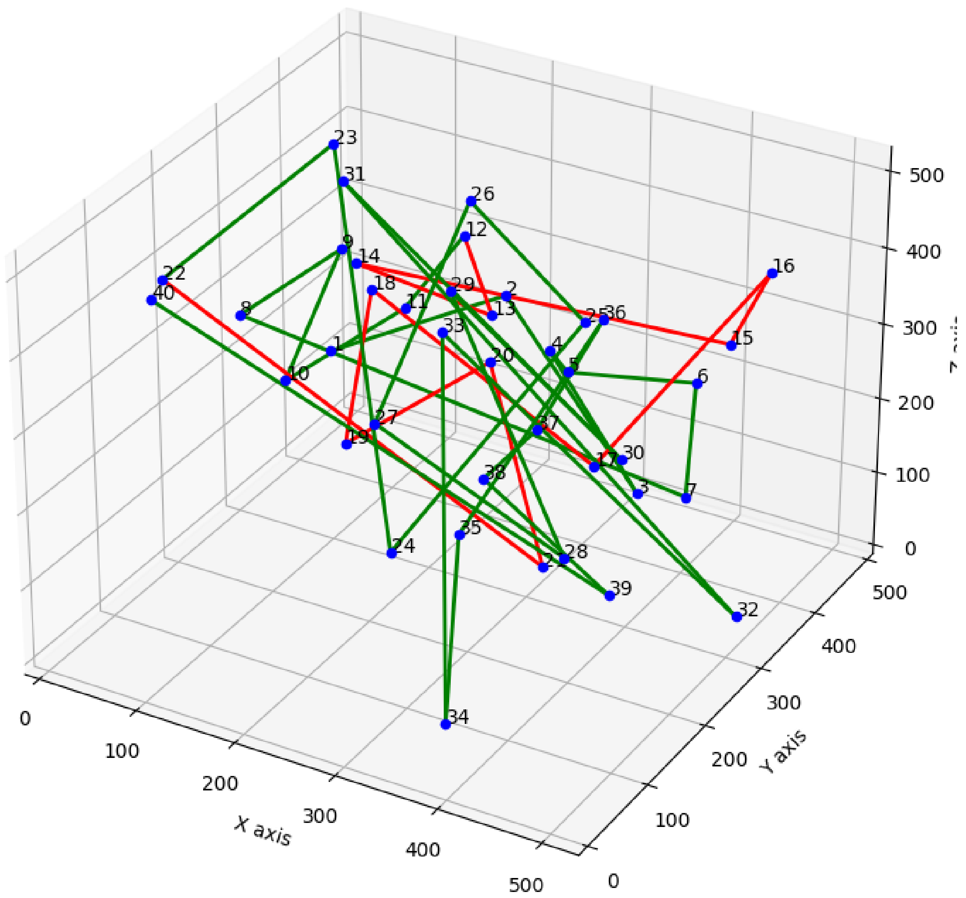
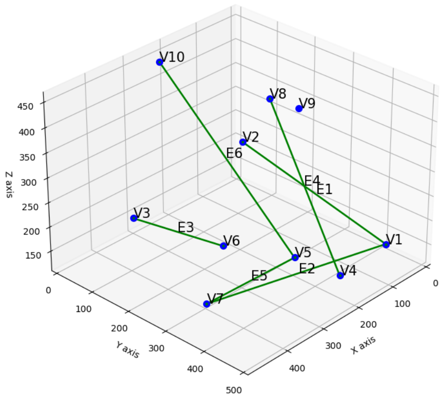



FIGURE 3.9: Example of UAV path.

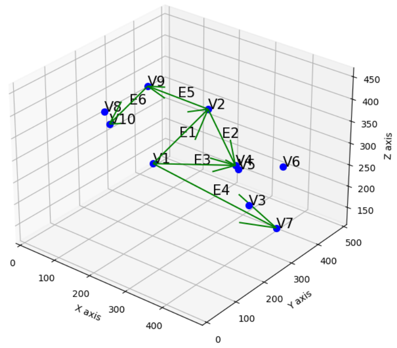
Each waypoint may not be connected to every other waypoint by the route. Figure 3.9 shows the path in 3D space. The lines represent the possible routes, the blue dots are waypoints, and the red lines are path traversed. Each route has a cost or multiple costs associated with it. The cost can be in terms of distance, time, energy, etc. The path planning problem is to find a path in free space that connects the start waypoint and the end waypoint, having minimal / optimal cost of traversal. Graph theory is used heavily in path planning, where waypoints are the nodes and routes are the edges in a graph.

### 3.3.1 Graph Theory

A graph is a collection of vertices and edges, where an edge connects two vertices. Mathematically graph can be expressed as $G = (V, E)$, where $E \subseteq [V]^2$, E represents edges, and V represents vertices. Graph theory is a study of graphs. Problems are modeled into mathematical graphs, and solutions are explored using graph theory. The graph and the graph theory are applicable in computer science, engineering, mathematics, natural science, networking, linguistics, etc. Figure 3.10 shows the directed and undirected graphs. In an undirected graph, the edge doesn't have direction and can be traversed in any direction. In a directed graph, the direction of traversal is fixed and is associated with the edge. The cost of traversal of between vertices is considered as weight and is associated to the edge. The cost of traversal is model specific and can hold combination of multiple parameters. The graph having such weights is called a weighted graph. The weights are used to find the ways to meet the criteria of the problem. The criteria can be to maximize or minimize weight, or to limit within range.




(a) Undirected Graph.




(b) Directed Graph.

FIGURE 3.10: Types of graphs.

### Graph Theory and Path Planning

Path in graph theory can be expressed as the edges connecting the start and the goal vertex of the graph. Mathematically path is $P = (V, E)$, where $V = \{V_1, V_2, V_3, ..., V_g\}$, $E = \{V_1V_2, V_2V_3, V_3V_4, ..., V_{g-1}V_g\}$, $V_1$ is start vertex and $V_g$ is goal vertex. The number of edges in a path is called path length. A path $P$ with length $l$ is denoted as $P^l$. There may be multiple paths to reach from the start vertex to the goal vertex. The set of possible paths $P_s$ is:

$$P_s = \{P_1^{l1}, P_2^{l3}, P_3^{l4}, ..., P_n^{ln}\}$$

The set of criteria for the feasible solutions $C_s$ is:

$$C_s = \{C_1, C_2, C_3, C_4, ..., C_m\}$$

The paths which match the criteria of the problem are the solution paths.

$$P_s \xrightarrow{\text{matching } C_s} \{P_{s1}^{ls1}, P_{s2}^{ls3}, P_{s3}^{ls4}, ..., P_p^{lsp}\}$$

This equation is a mathematical notation of path planning solutions. In real life, it may not be desired to find all the possible paths between the start and goal vertices. Various graph theory's theorems, graph properties, and algorithms can be used to find the feasible solution which meets the criteria.

# Chapter 4

# Path Planning Algorithms

This chapter provides the details of the algorithms that are considered for simulation. Selected coverage path planning and target reaching path planning algorithms are discussed with pseudo-codes.

## 4.1 Coverage Path Planning

In coverage path planning, an entire region of interest is covered. In this type of mission, the area is swept with a constant width on the ground, maintaining a constant height above ground. This width on the ground which a UAV covers at any instance is termed as sweep width. The width of the area on the ground is considered based on the capabilities of the sensors. The sensors can be a camera, microphone, thermal imaging, or any other sensors. The UAV can have multiple sensors as well. The height and angle of the sensor affect the sweep width and the quality of data. Figure 4.1 shows the width coverage of a camera at a constant height. For a camera sensor, the width coverage on the ground increases as the height increases. The resolution of the image decreases as the height increases, i.e., the image gets bloated as height increases, and it is difficult to identify an object or person. So an optimal height needs to be considered based on requirement and camera capabilities. Similarly, the angle of the camera also affects the width of the ground. At angle 90° with the horizontal axis, the width on the ground is minimum. As the angle decreases, the width increases, but after some value, the view shifts from the ground towards the atmosphere. An optimal angle has to be determined before the mission.

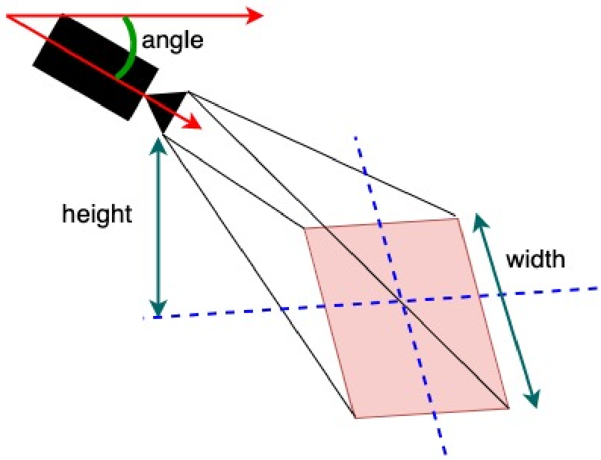



FIGURE 4.1: Sweep width and height for a single camera.

The patterns for coverage of the area, i.e., parallel search pattern, creeping search pattern, and spiral search pattern [35] are considered for study and simulation.

### 4.1.1 Parallel Line Search

Parallel line search is a back and forth pattern. It is the most common pattern and used as the default pattern by ground control softwares. In this pattern, back and forth is done parallel to the longer side of the region of interest, and turns are made parallel to the short side. The pattern is shown in Figure 4.2. The path has two legs, the longer leg termed as search leg and the short one termed as the cross leg. The search leg is parallel to the long side of the rectangular area of interest. The cross leg is 90° to the search leg and has a length of sweep width.

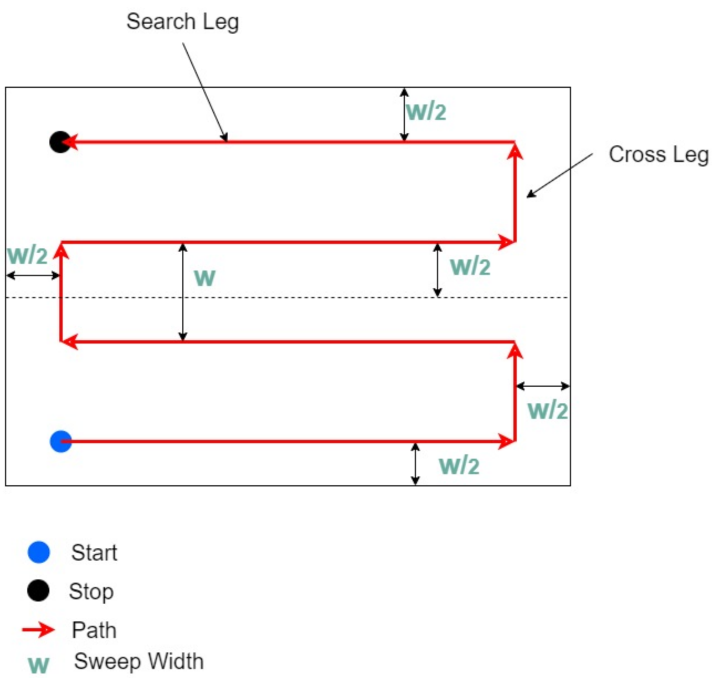



FIGURE 4.2: Parallel Line Search.

The pseudo-code for the parallel line search for a rectangular area of interest is presented in Algorithm 1. The sweep width should be relatively smaller than both the length and breadth of the rectangle. The direction of the search leg is determined in line 4 from starting point towards the opposite side of the rectangle along the long side. The direction of the cross leg is determined in line 5 towards the opposite side of the rectangle along the short side. Inside a while loop (line 9-24), waypoints are added (line 10) until the points are within the boundary of the region of interest. Starting waypoint is chosen towards the search leg with the dimension of the length of the rectangle minus sweep width (line 12 ). The next waypoint is chosen towards the cross leg with the dimension of sweep width (line 20). The cross leg dimension will be the width left to cover, in case it is less than sweep width (line 18). The next waypoint is chosen opposite to the search leg direction with the same dimension that of the search leg (line 14 ). This continues in a loop until the area is covered.

**Algorithm 1** Algorithm for Parallel Line Search.

**Ensure:**
sweep width << length of rectangle
sweep width << breadth of rectangle
1: **procedure** PARELLELLINEPATTERN($rectangle$,$sweepWidth$, $startPoint$,$height$)
2: $bounds \leftarrow$ boundary of $rectangle$
3: $length \leftarrow$ long side length of $rectangle$
4: $searchLegDir \leftarrow$ direction from $startPoint$ towards opposite side of $rectangle$ along long side
5: $crossLegDir \leftarrow$ direction from $startPoint$ towards opposite side of $rectangle$ along short side
6: $currentPoint \leftarrow startPoint$
7: $currentDir \leftarrow longEdgeDir$
8: $pathCnt \leftarrow 0$
9: **while** $currentPoint$ within $bounds$ **do**
10: $wayPoint[pathCnt] \leftarrow (currentPoint.x, currentPoint.y, height)$
11: **if** $pathCnt$ % 4 == 0 **then**
12: $currentPoint \leftarrow$ translate $currentPoint$ towards $searchLegDir$ with $length - sweepWidth$
13: **else if** $pathCnt$ % 4 == 2 **then**
14: $currentPoint \leftarrow$ translate $currentPoint$ opposite of $searchLegDir$ with $length - sweepWidth$
15: **else**
16: $widthLeft \leftarrow$ from $currentPoint$ to boundary towards $crossLegDir$
17: **if** $widthLeft > sweepWidth/2$ and $widthLeft < sweepWidth$ **then**
18: $currentPoint \leftarrow$ translate $currentPoint$ towards $crossLegDir$ with $widthLeft$
19: **else**
20: $currentPoint \leftarrow$ translate $currentPoint$ towards $crossLegDir$ with $sweepWidth$
21: **end if**
22: **end if**
23: $pathCnt \leftarrow pathCnt + 1$
24: **end while**
25: **end procedure**

### 4.1.2 Creeping Line Search

The creeping line search pattern is similar to the parallel line search 4.1.1, except the search leg is parallel to the short side of the rectangular area, and the cross leg is parallel to the long side. Figure 4.3 shows the path pattern of the creeping line search.

The creeping line search algorithm's pseudo-code is presented in Algorithm 2. Here as well, the sweep width should be relatively small compared to the length and breadth of the area. Algorithm 2 resembles Algorithm 1 in most aspects except the selection of search leg and cross leg direction. Here the search leg direction is selected from the start point towards the opposite side of the rectangle along the short side of the rectangle (line 4). The cross leg direction is selected from the start point towards the opposite side of the rectangle along the long side of the rectangle (line 5).

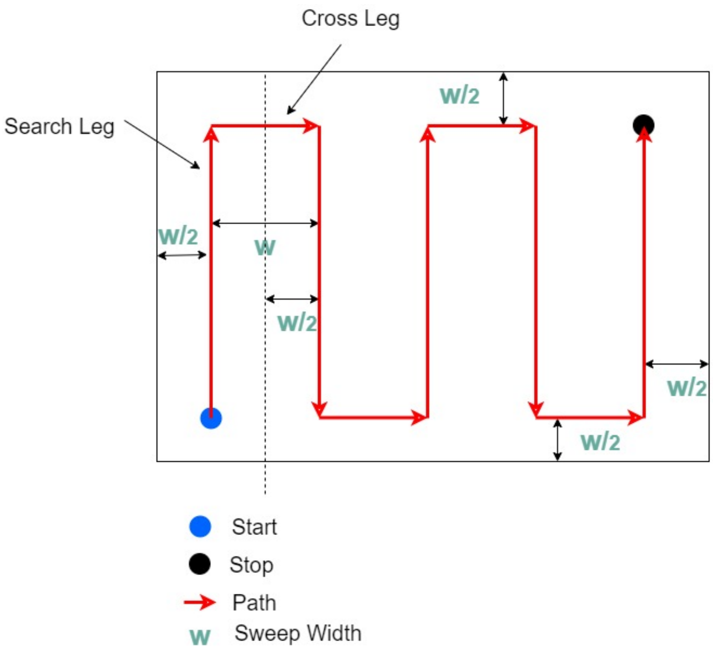



FIGURE 4.3: Creeping Line Search.

### 4.1.3 Spiral Search

The spiral pattern has a path with expanding concentric squares, as shown in Figure 4.4. This pattern is also known as square search. The path can be expanding squares starting from the center to outwards or contracting squares from outward to the center. The expanding squares are useful in cases when the region of interest is not fixed prior to the mission. The area of search can be rectangular as well. In such a case, the path will be concentric rectangles.

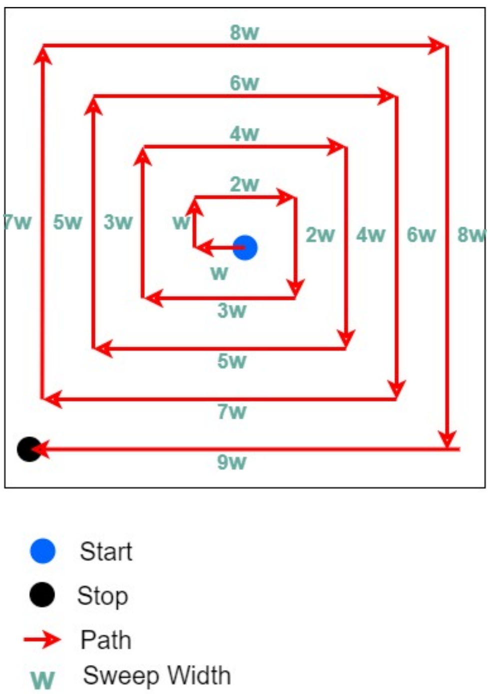



FIGURE 4.4: Spiral Search.

**Algorithm 2** Algorithm for Creepling Line Search.

```
Ensure:
    sweep width << length of rectangle
    sweep width << breadth of rectangle
procedure CreepingLinePattern(rectangle, sweepWidth, startPoint, height)
    bounds ← boundary of rectangle
    breadth ← short side length of rectangle
    searchLegDir ← direction from startPoint towards opposite side of rectangle along short side
    crossLegDir ← direction from startPoint towards opposite side of rectangle along long side
    currentPoint ← startPoint
    pathCnt ← 0
    while currentPoint within bounds do
        wayPoint[pathCnt] ← (currentPoint.x, currentPoint.y, height)
        if pathCnt % 4 == 0 then
            currentPoint ← translate currentPoint towards searchLegDir with breadth − sweepWidth
        else if pathCnt % 4 == 2 then
            currentPoint ← translate currentPoint opposite of searchLegDir with breadth − sweepWidth
        else
            widthLeft ← from currentPoint to boundary towards crossLegDir
            if widthLeft > sweepWidth/2 and widthLeft < sweepWidth then
                currentPoint ← translate currentPoint towards crossLegDir with widthLeft
            else
                currentPoint ← translate currentPoint towards crossLegDir with sweepWidth
            end if
        end if
        pathCnt ← pathCnt + 1
    end while
end procedure
```

Algorithm 3 presents the pseudo-code of the spiral search in an expanding manner. In an infinite loop (lines 6 - 14), "wayPoints" are added with increasing "sweepWidth" after every two waypoints (line 9). The direction is rotated by 90° in a specified direction ( CCW or CW) after every waypoint (line 12).

In Algorithm 4, an inward spiraling version of spiral search is presented with the pseudo-code. The algorithm is applicable to the rectangular and the square area. The "longEdge" argument in the procedure $SpiralSearchInward$ (line1), determines either the starting would be along long side or short side of the rectangular area. Same argument is used to decide if the direction of rotation will be CW or CCW (lines 5 - 11). The length of the path will vary according to length and breadth of the area, so "lengthEdge" and "widthEdge" variables are assigned with initial values, which are length minus sweep width and breadth minus sweep width, respectively (line 12 and

**Algorithm 3** Algorithm for Expanding Spiral Search.

```
procedure ExpandingSpiralSearch(sweepWidth, startPoint, height, startDirection, rotationDirection)
    currentPoint ← startPoint
    currentWidth ← sweepWidth
    currentDirection ← startDirection
    pathCnt ← 0
    while True do
        wayPoint[pathCnt] ← (currentPoint.x, currentPoint.y, height)
        if pathCnt % 2 == 0 then
            currentWidth ← currentWidth + sweepWidth
        end if
        currentPoint ← translate currentPoint towards currentDirection with currentWidth
        currentDirection ← currentDirection rotated by 90° in rotationDirection
        pathCnt ← pathCnt + 1
    end while
end procedure
```

13). These variables decay with sweep width one after another in a while loop (lines 16 - 41). The while loop stops when "widthEdge" becomes zero or less. The direction is rotated after every waypoint.

**Algorithm 4** Algorithm for Inward Spiral Search.

```
Ensure: :
    sweep width << length of rectangle
    sweep width << width of rectangle
procedure SpiralSearchInward(rectangle, sweepWidth, startPoint, height, longEdge)
    bounds ← boundary of rectangle
    length ← length of rectangle
    width ← width of rectangle
    if longEdge == True then
        currentDirection ← direction along long side
        rotateDirection ← rotation direction from long side to short side
    else
        currentDirection ← direction along short side
        rotateDirection ← rotation direction from short side to long side
    end if
    lengthEdge ← length − sweepWidth
    widthEdge ← width − sweepWidth
    currentPoint ← startPoint
    pathCnt ← 0
    while widthEdge > 0 do
        wayPoint[pathCnt] ← (currentPoint.x, currentPoint.y, height)
        if longEdge then
            if pathCnt % 2 == 0 then
                currentPoint ← translate currentPoint towards currentDirection with lengthEdge
                if pathCnt !=0 then
                    lengthEdge ← lengthEdge − sweepWidth
                end if
            else
                currentPoint ← translate currentPoint towards currentDirection with widthEdge
                widthEdge ← widthEdge − sweepWidth
            end if
        else
            if pathCnt % 2 == 0 then
                currentPoint ← translate currentPoint towards currentDirection with widthEdge
                if pathCnt !=0 then
                    widthEdge ← widthEdge − sweepWidth
                end if
            else
                currentPoint ← translate currentPoint towards currentDirection with lengthEdge
                lengthEdge ← lengthEdge − sweepWidth
            end if
        end if
        currentDirection ← currentDirection rotated by 90° in rotationDirection
        pathCnt ← pathCnt + 1
    end while
end procedure
```

## 4.2 Target Reaching Path Planning

In target reaching path planning, a destination point is given in prior, and the UAV has to reach it, avoiding the obstacles. In SAR, the region of interest doesn't have a well-defined map to maneuver. Fixed waypoints or fixed nodes and pre-determined possible paths are not known. An approach that is non-deterministic and can make decisions on the fly is required in this case. RRT[21] is a non-deterministic algorithm, and it doesn't require environment modeling. This algorithm suits SAR missions and is considered for this thesis.

### 4.2.1 Rapidly exploring Random Trees (RRT)

RRT[21] algorithm works well where information of environment is not known in prior. It depends on sensors' data to sense the surroundings and determine the next step. It creates a tree by randomly selecting a node in the environment in each step. The tree eventually fills up the free spaces in the environment. RRT explores the environment, creates a graph of free spaces and finds a path from one point to other. The path may not be optimal. This algorithm can address the non-holonomic and dynamics constraint. Figure 4.5 shows the RRT in a 3D environment.

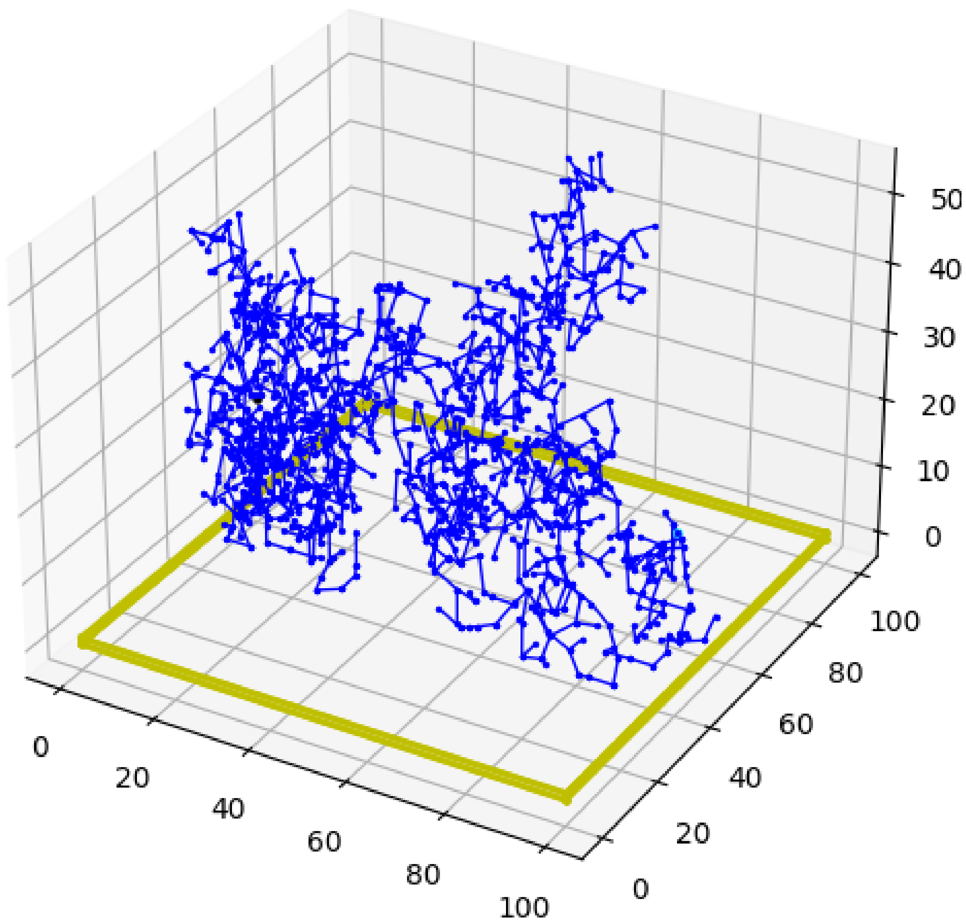



FIGURE 4.5: Rapidly exploring Random Trees (RRT).

The pseudo-code of RRT is presented in Algorithm 5. The input arguments are the start point , goal point, and the maximum number of iterations (line 1). The start point is the root of the tree. When the tree grows and able to find the goal point, the algorithm stops (line 15). The algorithm explores the whole free space when the number of iterations tends to infinity. It may happen the goal point is not found after many iterations. The $maxIteration$ argument is used to limit the iteration if the goal point is not found. The algorithm continues to find random point 6 in a while loop (lines 5 - 17). The core of RRT is the determination of random points. If the random point is chosen without bias, then the algorithm may converge after infinite iteration. A random generator with good estimation bias can converge the algorithm quickly.

**Algorithm 5** RRT Algorithm.

1: **procedure** RRT($startPoint$, $goalPoint$, $maxIteration$)<br>
2:     intialize empty graph $rrtGraph(v, e)$<br>
3:     $rrtGraph.add(startPoint, 0)$<br>
4:     $count \leftarrow 0$<br>
5:     **while** $count < maxIteration$ **do**<br>
6:         $randPoint \leftarrow RandomPoint()$<br>
7:         $count \leftarrow count + 1$<br>
8:         **if** $randPoint$ is in obstacle space **then**<br>
9:             continue<br>
10:         **end if**<br>
11:         $nearestPoint \leftarrow$ node in rrtGraph nearest to $randPoint$<br>
12:         $randPoint.parent = nearestPoint$<br>
13:         $rrtGraph.add(randPoint$, distance($randPoint$, $nearestPoint$))<br>
14:         **if** $randPoint$ is very near to $goalPoint$ **then**<br>
15:             break<br>
16:         **end if**<br>
17:     **end while**<br>
18:     return $rrtGraph$<br>
19: **end procedure**

# Chapter 5

# Tools and Software

This chapter covers the details of tools and software used for implementation. It also presents the methodologies for using the tools to implement the path planning algorithms. Available tools and frameworks were reviewed from the survey papers [45, 46].

## 5.1 Gazebo

Gazebo[47] is an open-source robot simulation tool maintained and developed Open Source Robotics Foundation. It has distributed architecture with a server and a client. The server is used for simulating world physics, rendering and, sensors. The client is used for providing a graphical interface for visualization and interaction with the simulation. It supports Robot Operating System (ROS) and flight simulators interaction with it. Gazebo is supported only on Linux and Mac operating systems. It is very lightweight and can run on general computers with adequate RAM and no graphics card. Figure 5.1 shows the gazebo client, which is a GUI interface.

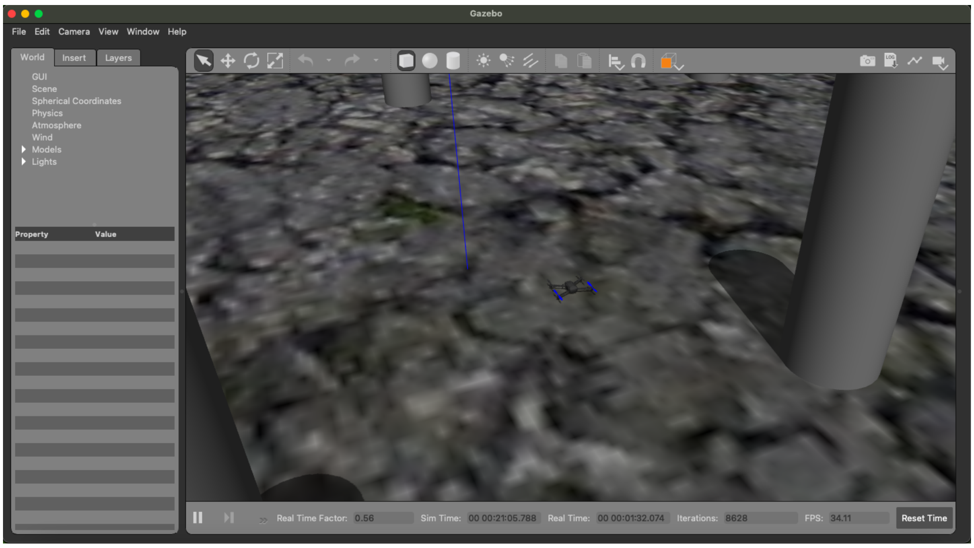



FIGURE 5.1: Gazebo client window.

## 5.2 Px4

PX4[4] is an open-source professional autopilot software stack. It is developed and maintained by Dronecode Project. It supports various ground vehicles, drones, and submersibles systems. All the types of UAVs shown in Figure 2.1 are supported by it. The Px4 software supports many flight controller hardware like Pixhawk 4 [48], Pixhawk 3 Mini[48], Pixracer[48], and others. There are many commercial UAVs that run on Px4. Px4 stack also runs in Hardware In The Loop (HITL) and Software In The Loop (SITL) modes.

### 5.2.1 Px4 Architecture

The software architecture of Px4 is shown in Figure 5.2. It has four major blocks flight control, drivers, storage, and external connectivity. External connectivity block represents the support for two protocols, MAVLink[49] and FastRTPS, via UART and UDP interface. Drivers block has drivers software for sensors like GPS, Camera, IMU Drivers, etc. Storage block consists of database, parameters, and runtime logs. The main logic resides in the flight control block. This block controls the actuators, gets input from sensors, receives commands via MAVLink/FastRTPS, and retrieves/saves data into the storage device.

The high-level block diagram of Px4 working is shown in Figure 5.3. It has guidance, navigation, and control (GNC), functionality, which makes the autonomy of UAV feasible. Navigator is responsible for navigation, position controller provides the guidance, and attitude and rate controller acts like control for actuators. The estimator fetches data from various sensors, computes the state and other parameters. It then feeds that information to the navigator and controllers.

### 5.2.2 Px4 Simulation

Both HITL and SITL simulation is possible with Px4. In HITL the stack runs on a hardware flight controller. The flight controller is connected to simulation software via USB. The sensor data is fetched from the simulator and processed on the hardware controller, and the control signal is forwarded to the simulator. The simulator can be connected to ground control and offboard API via UDP ports. Figure 5.4 shows the overview of HITL of Px4.

In SITL the software stack runs on a general computer. Figure 5.5 shows the setup for Px4 on SITL. Px4 on SITL is connected to API and ground control via default UDP ports 14540 and 14550, respectively. It is connected to the simulator via TCP port 4560. Each of the blocks in Figure 5.5 can be run on different computers and communicate over a network. The sensor data is passed on Px4 by the simulator, and the control data is sent to the simulator after processing. The supported simulator are Gazebo[47], flightgear[50], JSBSim[51], jMAVSim[52], and Airsim[53]. Simulators Gazebo[47] and jMAVSim[52] run in lockstep mode with Px4 on SITL. In lockstep simulation, the Px4 stack waits for sensor data from the simulator, and the simulator waits for actuator data from the Px4 stack.

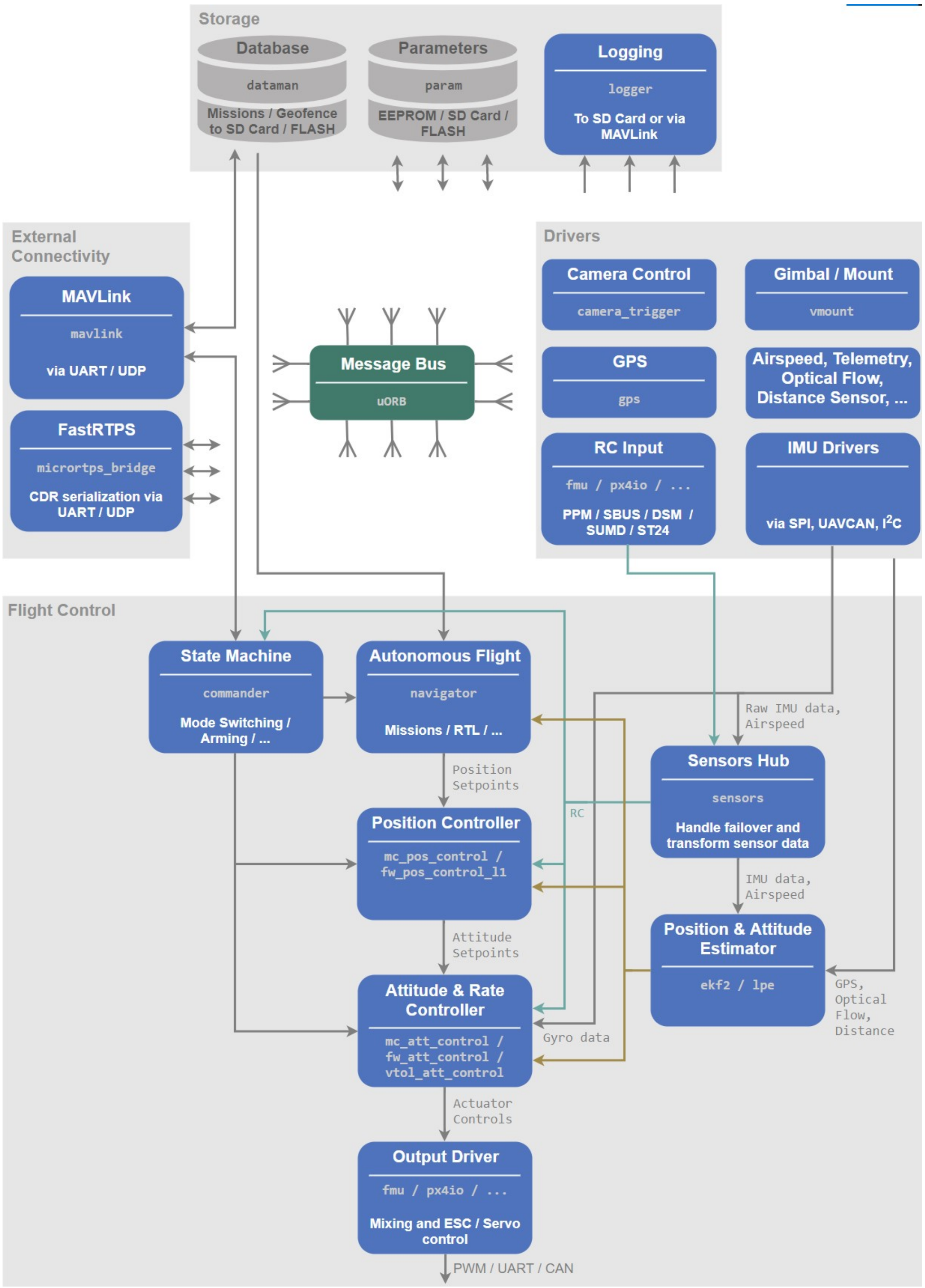



FIGURE 5.2: Px4 Architecture. *Courtsey:Dronecode[4].*

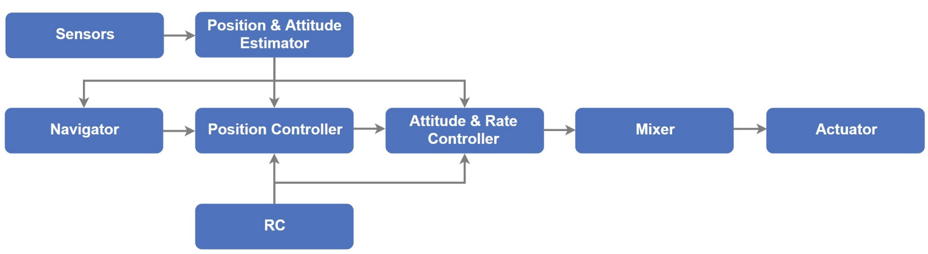



FIGURE 5.3: Px4 High Level Flight Stack. *Courtsey: Dronecode[4].*

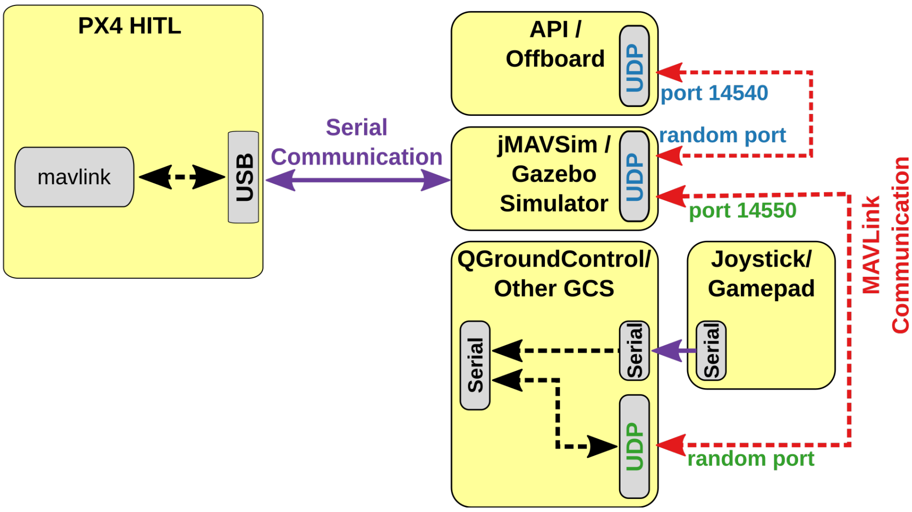



FIGURE 5.4: Px4 HITL. *Courtsey: Dronecode[5].*

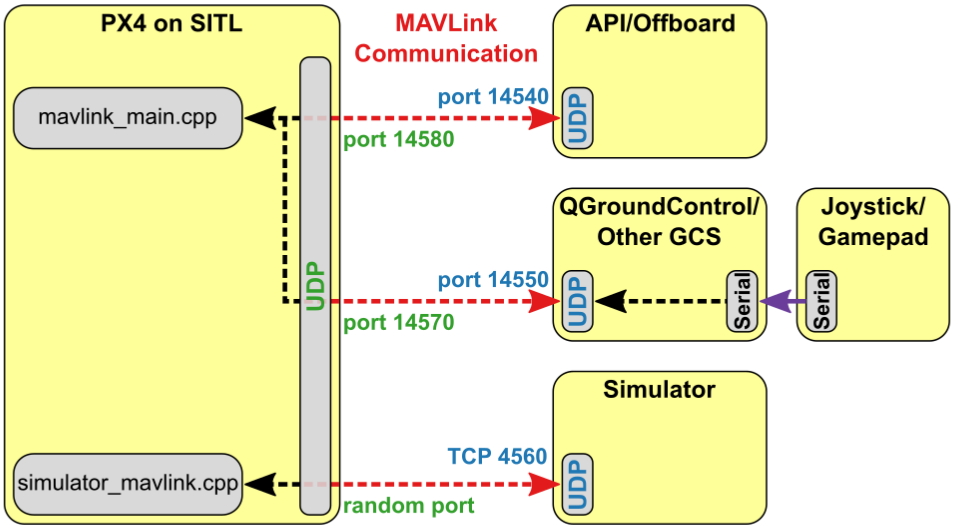



FIGURE 5.5: Px4 SITL. *Courtsey: Dronecode[5].*

## 5.3 QGround Control

QGroundControl[54] is a GUI-based ground control software for Px4 and ArduPilot powered platforms. It provides full control, setup, and monitoring of platforms via the GUI interface. It runs on Windows, macOS, Linux, iOS, and Android devices. It can be interfaced with a RC joystick and communicate with platform via MAVLink protocol. It connects with simulators in HITL and with the platform stack in SITL via UDP ports. Figure 5.6 shows the interface of QGround Control.

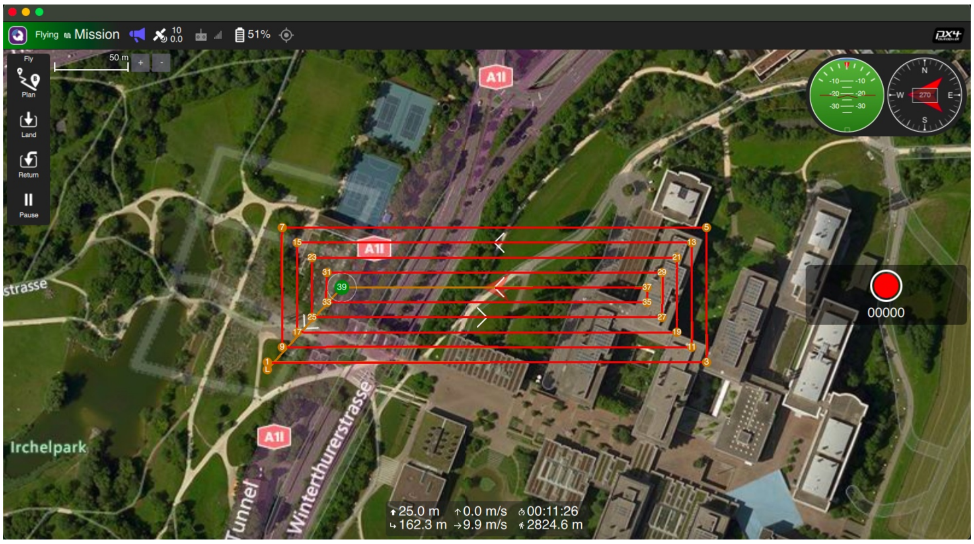



FIGURE 5.6: QGroundControl GUI.

## 5.4 MAVLink and MAVSDK

MAVLink[49] is the de-facto messaging protocol for UAVs. It is used for communication with UAV and between components on it. It is a combination of publish-subscribe and point-to-point models. It has two major versions v1.0 and v2.0. MAVLink can be used with MAVSDK. It is a collection of libraries and exposes APIs for various programming languages. The supported languages are C, C++, Python, Java, Go, Javascript, CSharp, and Rust.

## 5.5 Flight Review

A log file is created in PX4 software from the time of take-off till landing. The log file is used for post-flight analysis. The Flight Review is an online PX4 log analyzer provided by the PX4 community. It is available at `https://review.px4.io/`. It provides graphical charts for path, altitude, angles with axes, rate of change of angles, velocities along axes, accelerations along axes, GPS data, resource utilization, and axes positions w.r.t time. It also extracts and displays the log messages generated during the flight.

# Chapter 6

# Implementation and Results

This chapter provides details about the implementation and findings. It has a description of the system, tools and library, and the outcome of the simulation. The analysis and comparison of different algorithms are presented here.

## 6.1 System Description

The details of the computer on which the implementation was carried out are provided in Table 6.1. The versions of various tools used are listed in Table 6.2. Python programming language was used for interacting with Px4 on SITL, modeling environment, and implementing the algorithms. The python libraries used in the process are listed in Table 6.3.

| **Computer Description** | |
|---|---|
| Model | MacBook Air |
| OS | macOs Big Sur Version 11.1 |
| Processor | 1.6 GHz Dual-Core Inter Core i5 |
| RAM | 8 GB 1600 MHz DDR3 |
| Graphics | Inter HD Graphics 6600 1536 MB |

TABLE 6.1: Details of components of computer used.

| **Tools Description** | |
|---|---|
| Gazebo | Version 11.3.0 |
| PX4 SITL | Version 1.11 |
| MAVLink Protocol | Version 2 |
| Flight Review | Online Service |
| Anaconda | Version 4.9.2 |
| Jupyter Notebook | Version 6.2.0 |
| Python | Version 3.7 |

TABLE 6.2: List of tools and their version.

| Python Libraries | |
|---|---|
| mavsdk | Version 0.15.0 |
| pygeodesy | Version 21.2.12 |
| shapely | Version 1.7.1 |
| geopandas | Version 0.8.2 |
| matplotlib | Version 3.3.4 |
| Geometry3D | Version 0.2.2 |

TABLE 6.3: Used python libraries and their versions.

## 6.2 Coverage Area Implementation

The implementations of coverage area algorithms parallel line, creeping line, and spiral search mentioned in 4.1, are described here. This section covers the setup for the implementation, model of UAV, results, and analysis.

### 6.2.1 Setup

Gazebo with Px4 on SITL is used for simulation. Figure 6.1 shows the setup with the flow of data and commands among the blocks. The setup was run on one computer, and the flight logs were collected after the completion of the mission. The log was uploaded to an online flight log analyzer to get an analysis of the mission. Python MAVSDK was used on Jupyter Notebook to control the UAV. QGroundControl was used to monitor the progress of the mission.

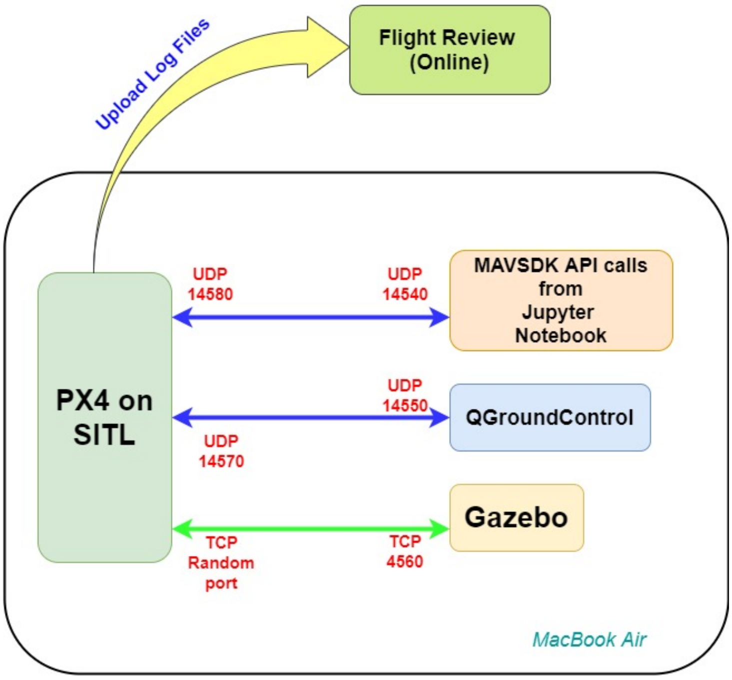



FIGURE 6.1: Setup for coverage algorithm simulation.

### 6.2.2 UAV Modeling

A quadcopter model Iris 3DS was used. Figure 6.2 shows the model in Gazebo along with its components. It has a body with links, joints, and plugins. At the joints, links of the same object or other objects can be attached. At the joint "rotor_0_joint", link "rotor_0" is attached. The link "rotor_0" is a component of the same object, "iris". At the joint "gps0_joint", "iris::gps0" is attached, which is another object. The plugins consist of a middleware program that simulates various components of the model. The plugins for motors, magnetometer, barometer, mavlink, IMU, and groundtruth were attached to the quadcopter. The GPS model was attached to the quadcopter as a different object. The camera was not attached to the model.

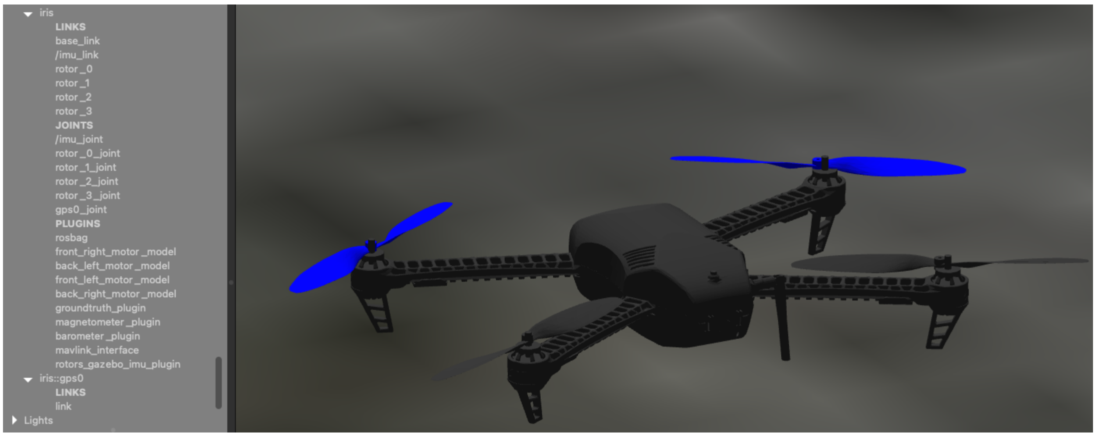



FIGURE 6.2: Iris 3DS Quadcopter Model.

### 6.2.3 Environment Modeling

A custom environment was created for simulation. A flat ground of 1114.49 m long, 905.077 m wide, and 0.1 m thick was created. A random number of solid cylinders were distributed over the ground. The height of all cylinders was less than the height set for the flight mission. All the objects had mesh and physics body, i.e., had solid shapes and collision enabled. The wind in the environment was set to zero. Figure 6.3 shows the environment of simulation in the Gazebo.

### 6.2.4 Region of Interest and Height

Four algorithms are taken into consideration for simulation. Region of interest and height was fixed for all the algorithms so that a statistical analysis could be deduced among them. The considered region of interest was an area of 300 m long and 100 m wide. The considered height of the mission was 25 m.

### 6.2.5 Parallel Line Search

The algorithm mentioned in Subsection 4.1.1 was implemented. The pseudo-code in Algorithm 1 was translated into python function ( Appendix B, function getParallelLineMissionItem, starting at line 178). The function requires polygon as an argument, either square or rectangle. The polygon's co-ordinates have to be in longitudes

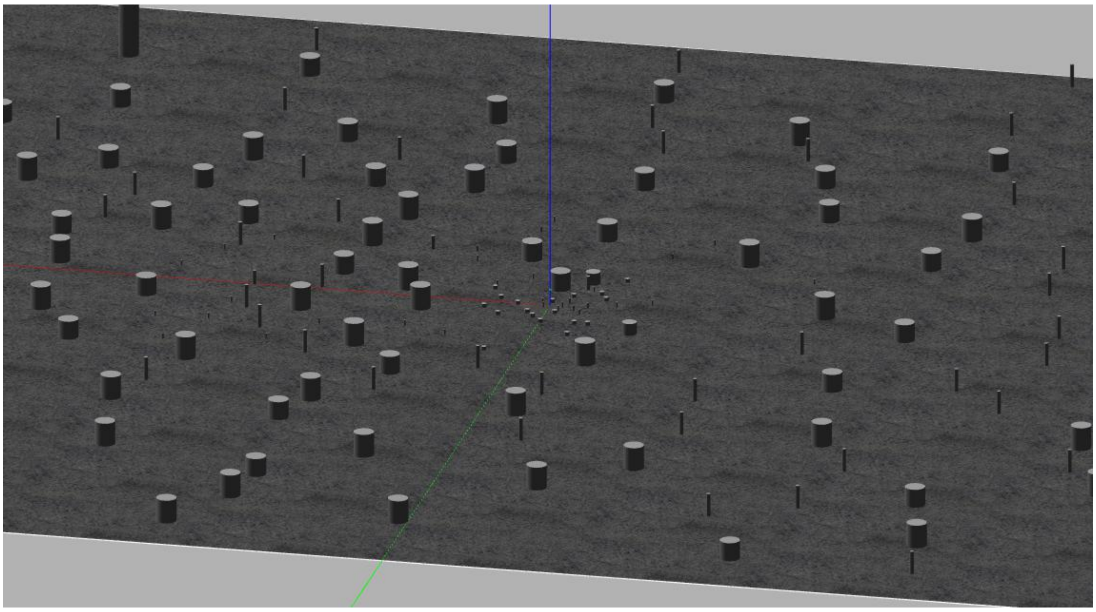


FIGURE 6.3: Environment for simulation.

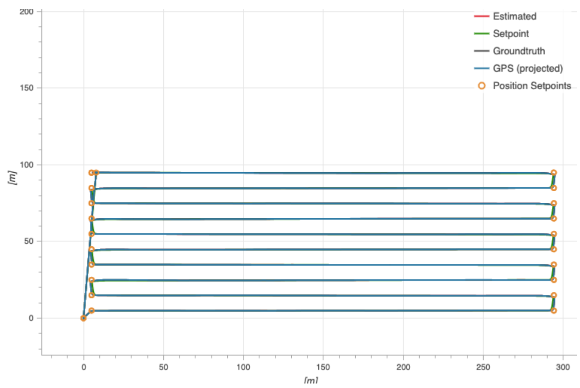
and latitudes. It is the region of interest of the mission. The first co-ordinate of the polygon is considered as the starting corner. The other arguments are sweep width, height, and speed. These are optional and take default values if not specified. The function returns an array of MAVSDK MissionItem, i.e., the waypoints of the path. The returned array is used to create Mission (Appendix A,line 34).

Figure 6.4 shows the graphical analysis of the mission. Sub-figure 6.4a shows the completed mission path. It can be seen the back and forth path, with longer legs of path parallel to the longer side of the region of interest. The mission starting point is the base and returns back to the same point after completion of the mission. Sub-figure 6.4b shows the height of the UAV during the flight. Z-axis points are negative as the graph show NED frame analysis. The mission was carried out at a constant height of 25 m and can be seen on graph 6.4b. After the mission, the UAV reaches the default height of 30 m to return to the start point. Sub-figure 6.4c and 6.4d, respectively, show accelerations and velocities along the 3 axes during the mission.

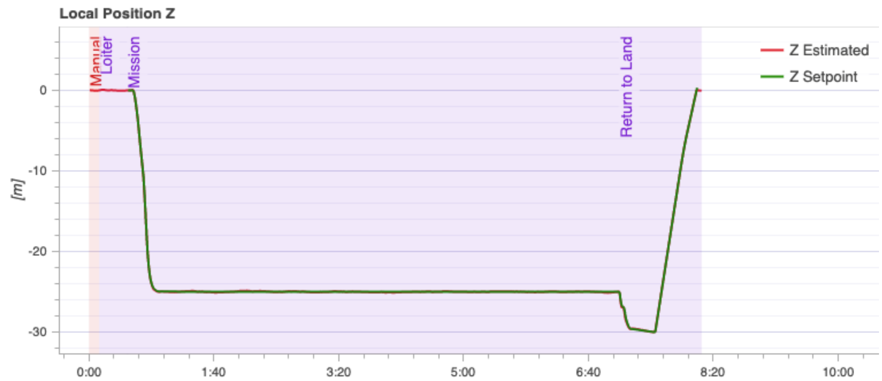
### 6.2.6 Creeping Line Search

The algorithm mentioned in Subsection 4.1.2 was implemented. The pseudo-code in Algorithm 2 was translated into python function "getCreepingLineMissionItem" ( Appendix B, function , lines 235). The arguments and return type of this function are similar to that of "getParallelLineMissionItem", as mentioned in Subsection 6.2.5. The underlying logic is different. It returns the path whose long leg is parallel to short side of region of interest.

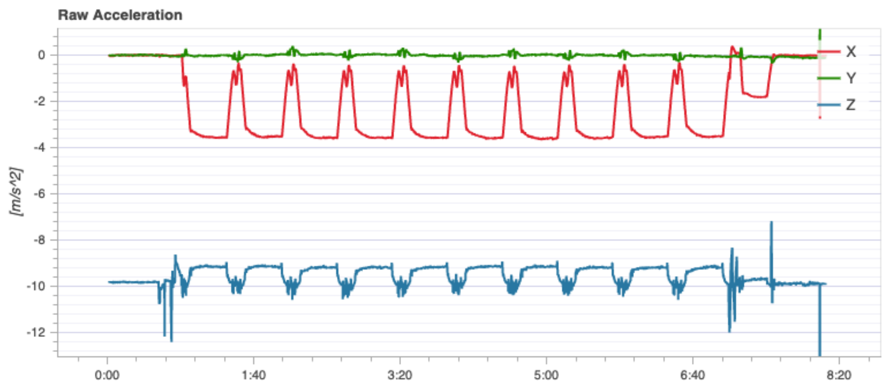
Figure 6.5 has a graphical analysis of the mission with this algorithm. Sub-figure 6.5a shows the completed mission path, Sub-figure 6.5b shows the variation of height, Sub-figure 6.5d shows the variation of velocities along 3 axes, and Sub-figure 6.5c shows the variation of accelerations along 3 axes.



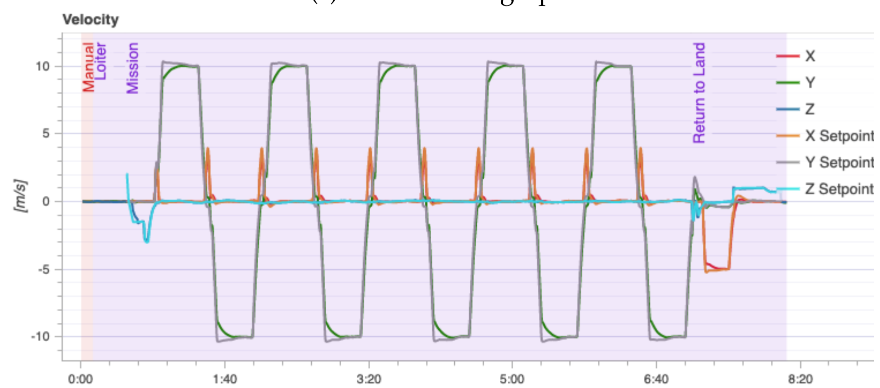

(a) Path traversed.




(b) Height graph.

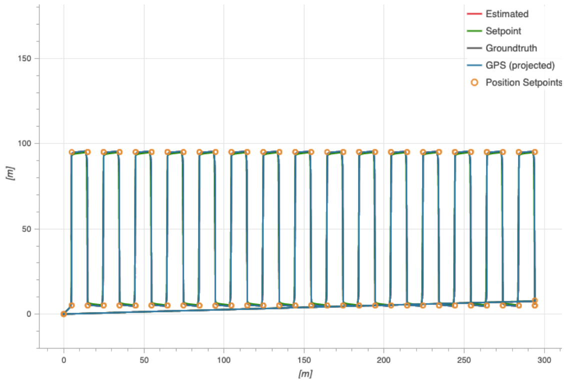



(c) Acceleration graph.

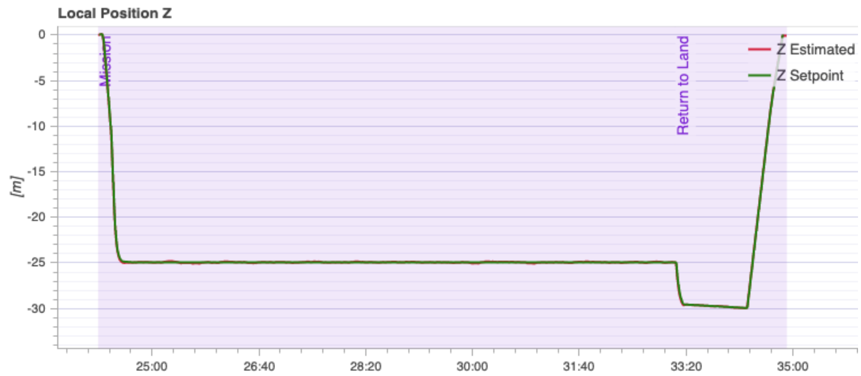



(d) Velocity graph.

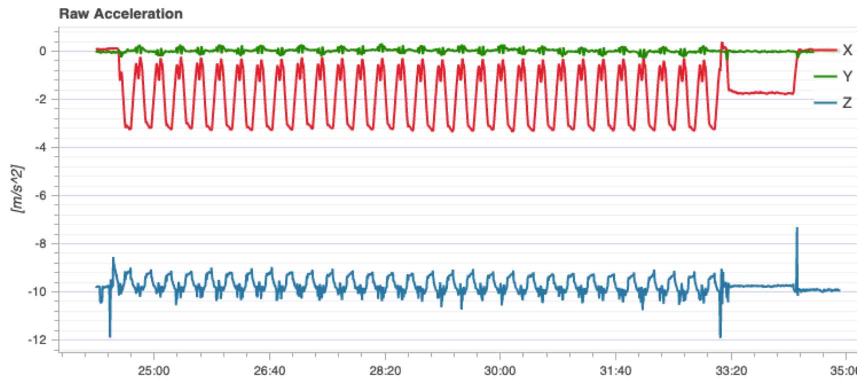
FIGURE 6.4: Parallel line search mission analysis.



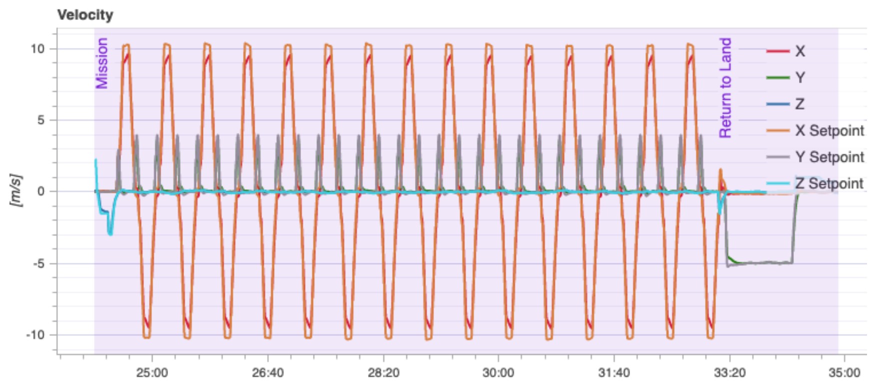

(a) Path traversed.




(b) Height graph.




(c) Acceleration graph.




(d) Velocity graph.

FIGURE 6.5: Creeping line search mission analysis.

### 6.2.7 Spiral Search (Long Edge First)

Subsection 4.1.3 elaborated on the spiral search and mentioned its applicabilities in non-square region. The algorithm described in Subsection 4.1.3, started from the center and kept on growing spirally in an outward fashion. The implementation is done in a spirally inward way. Here the path starts with a path segment parallel to the longer side of the region of interest. Python function "getLongEdgeSpiralMissionItem" ( Appendix B, function , starting line 293) has the implementation of spiral inward for square/rectangle region of interest. Its arguments and return variable are similar to that of "getParallelLineMissionItem", mentioned in Subsection 6.2.5.

The mission analysis of this algorithm is shown in graphical form in Figure 6.6. The path traversed is shown in Sub-figure 6.6a. The fluctuation of height is shown in Sub-figure 6.6b. The velocities and accelerations variations during the mission is captured in graphs 6.6d and 6.6c, respectively.

### 6.2.8 Spiral Search (Short Edge First)

This algorithm is almost similar to Long Edge First Spiral Search mentioned in Subsection 6.2.7, except the first path segment is parallel to the short side of the region of interest. Python function "getShortEdgeSpiralMissionItem" ( Appendix B, function , starting line 375) has the implementation of this algorithm. The function's arguments and return variables are similar to that of "getParallelLineMissionItem", mentioned in Subsection 6.2.5.

The graphical mission analysis is shown in Figure 6.7. The four sub-figures 6.7a, 6.7b, 6.7d, and 6.7c show the path, height, velocities, and accelerations, respectively.

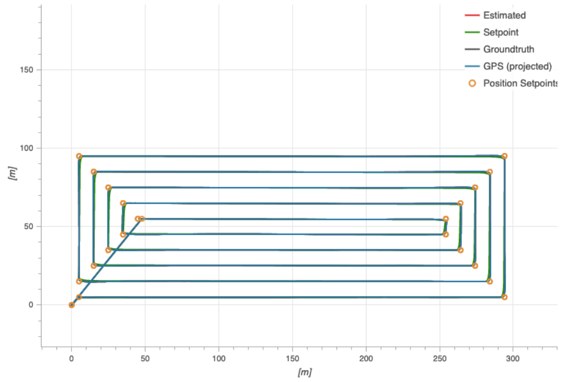
### 6.2.9 Comparison of the Algorithms

Comparison among the algorithms was deduced based on the analysis of the algorithms in subsections 6.2.5, 6.2.6, 6.2.7, and 6.2.8. Table 6.4 shows the comparison and its parameter.

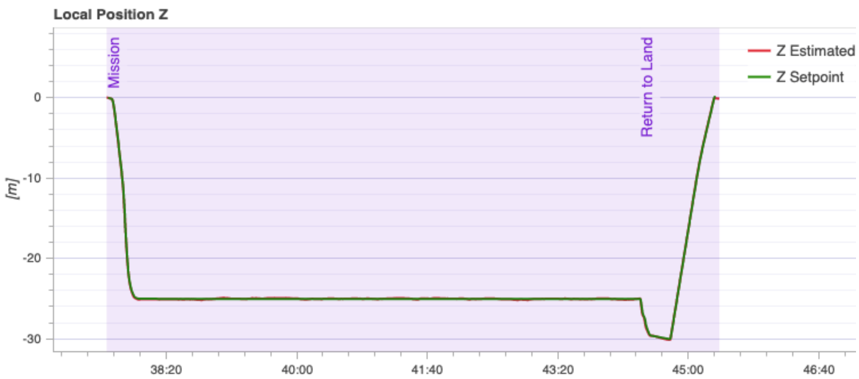
The parallel Line algorithm stands out as the fastest algorithm to cover the region. It has the minimum number of turns in the path, among others. The time to return to the base depends on the last waypoint of the mission. The last waypoint for the parallel line would be near to either diagonally opposite corner or adjacent corner along with the breadth of the region. The last waypoint for the simulation was near the adjacent corner, so the return time to the base was short.

The creeping Line algorithm is the slowest among the considered algorithms. It has the highest number of turns in the mission path, which added additional delays. At each waypoint, the UAV has to de-accelerate to take the turn, it adds to the delay. The number of turns can be considered as the cause of the slowest performance. This algorithm has its end waypoint near to either diagonally opposite corner or adjacent corner along the length of the region. The time to return to base would be either equal or greater than the time to return of Parallel Line. This algorithm, in general, would have the longest time to return compared to others.

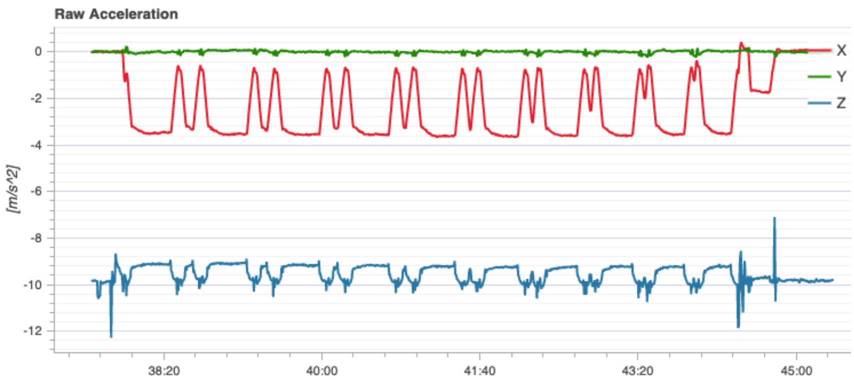
The spiral long edge first and the spiral short edge first algorithms performances are relatively similar. The spiral long edge first algorithm takes a little less time to complete the mission than the later algorithm. The difference in mission time is due



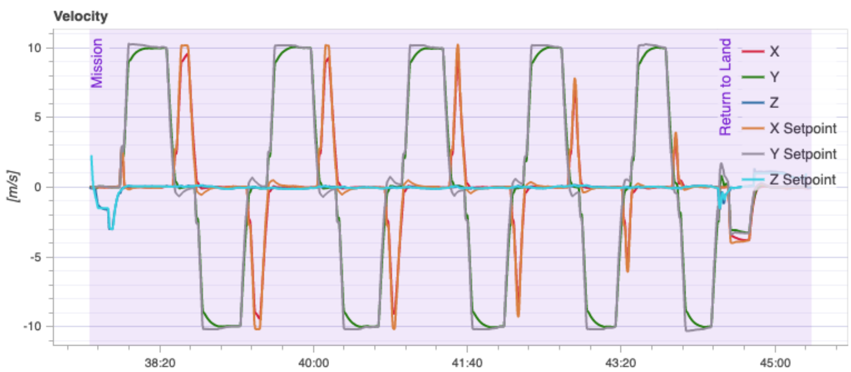

(a) Path traversed.




(b) Height graph.

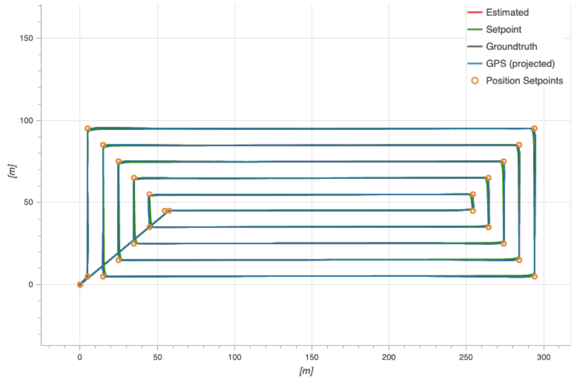



(c) Acceleration graph.

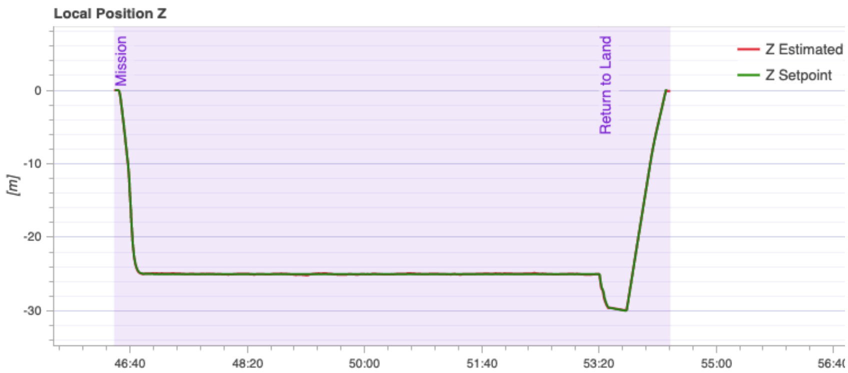



(d) Velocity graph.

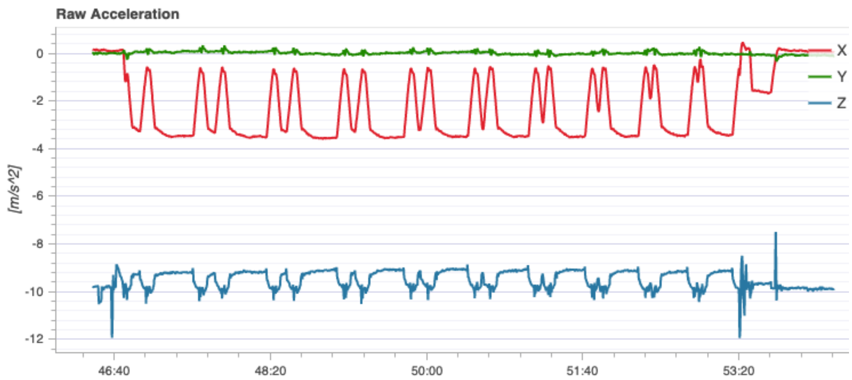
FIGURE 6.6: Spiral long edge first mission analysis.



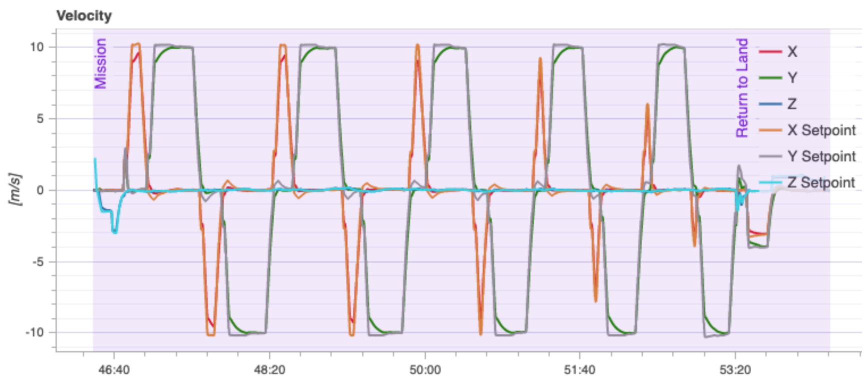

(a) Path traversed.




(b) Height graph.




(c) Acceleration graph.




(d) Velocity graph.

FIGURE 6.7: Spiral short edge first mission analysis.

to the difference in the number of turns in the mission path. The spiral short edge first algorithm had one turn more than the other in the selected simulation. Both the algorithm ends the mission near the center of the region of interest. This is the reason they have nearly equal time to return to base. These two algorithms have a shorter time to return than the other two.

| **Algorithm** | **Mission Time (sec)** | **Time to return (sec)** | **Number of Turns** |
|---|---|---|---|
| Parallel Line | 394 | 63 | 18 |
| Creeping Line | 541 | 101 | 58 |
| Spiral Long Edge First | 408 | 58 | 18 |
| Spiral Short Edge First | 413 | 59 | 19 |

TABLE 6.4: Comparison of algorithms.

## 6.3 Target Reaching Path Planning

The implementation of the RRT algorithm described in Section 4.2, is discussed here. The implementation is done entirely in python using the Geometry3D library.

### 6.3.1 UAV Modeling

UAV is modeled as a sphere that is a mesh structure of 10 longitudes and 4 latitudes. This can be seen in Figure 6.8. This creates 40 complex polygon meshes on the surface of the sphere. The normal vector passes through the centroid of the polygon and perpendicular to the surface. Each polygon has a normal vector, as shown in Figure 6.8. The directions a UAV can move are considered to be along these normal vectors as well as along the x, y, and z axes.

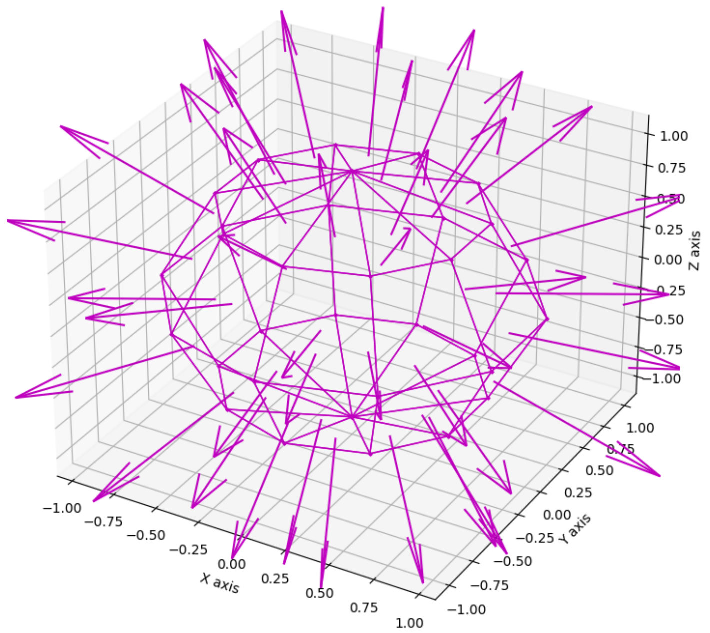



FIGURE 6.8: Spherical model of an UAV.

### 6.3.2 Environment Modeling

The environment is modeled with 3D shapes, obstacles, and ground. The ground is considered a parallelepiped of dimension 100 x 100 x 1 ( length x breadth x height) units. Its base is on the positive x-y plane with one corner as the origin. Obstacles are modeled as spheres, cones, and cylinders of different sizes. These are distributed randomly over the ground object. The ground and the obstacles' can be seen in Figure 6.9. The environment limits are set to 100 units on all three axes.

### 6.3.3 Obstacle Detection

The intersection method of the Geometry3D library is used to detect the obstacle. This method provides the intersection of the points of any two 3D Geometry3D objects.

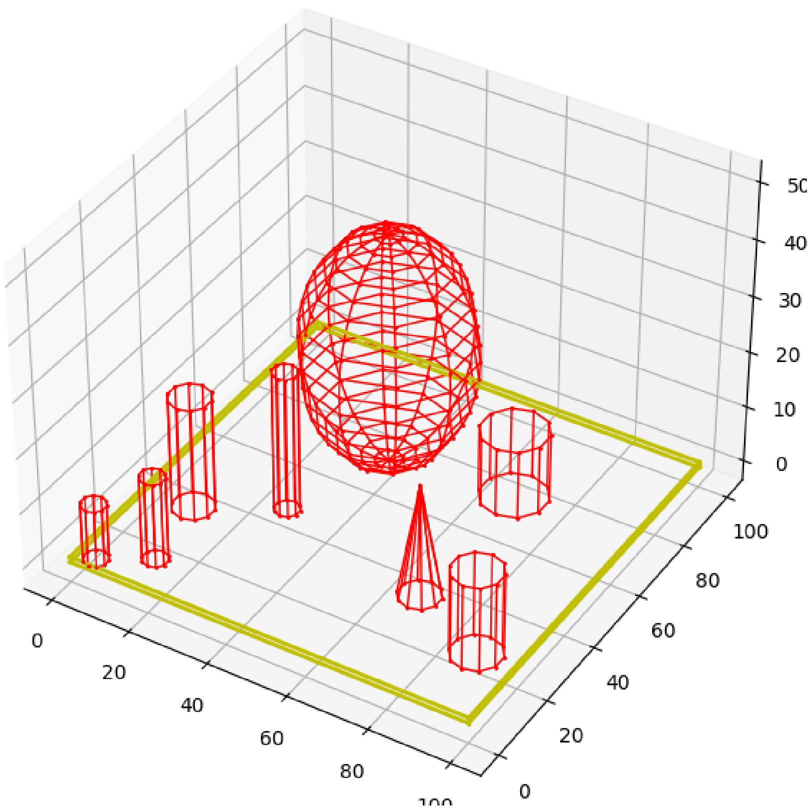



FIGURE 6.9: Environment model.

When a random point is selected in the environment, the path segment is the line joining the current point and the random point. Taking the path segment as axis, a cylinder with a unit radius is created, and the intersection is checked with all the obstacle objects. If no intersection is found with the obstacles, the path segment is in free space. In Figure 6.10, the cylinder in cyan color shows the cylinder of path segment, and the section in blue color shows the intersection with the obstacle.

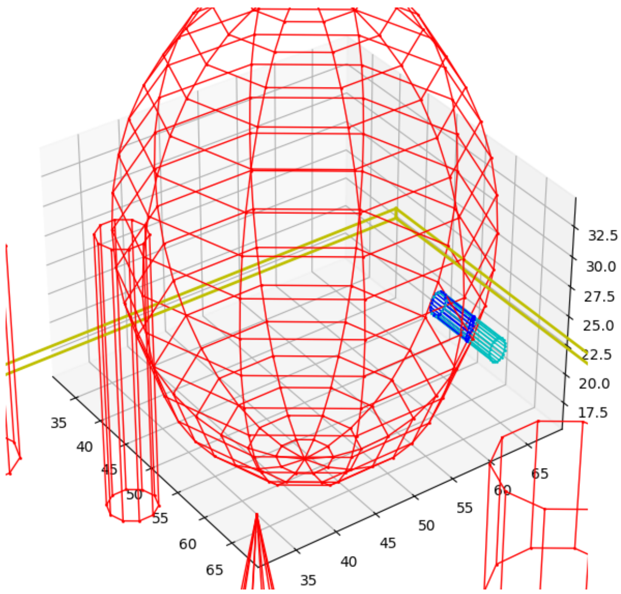



FIGURE 6.10: Obstacle detection with intersection.

### 6.3.4 Unbiased RRT

RRT pseudo-code presented in Algorithm 5 is implemented with the environment and UAV modeling. Random point is chosen along the vectors of UAV movement mentioned in Subsection 6.3.2. The vector is chosen randomly without any bias. A point at an incremental distance from the current point along the chosen vector is the random point. The implementation of the RRT without bias is done in function *run*, listed in Appendix C, line 413.

The algorithm was run with the iteration of 2000. The chosen start and goal points are (80,78,5) and (7,50, 20) respectively. The algorithm found the goal in the $1642^{nd}$ iteration. Figure 6.11c shows the tree created by the algorithm. Figure 6.11a displays the actual path covered by the UAV. The paths which encountered obstacles are shown in Figure 6.11b.

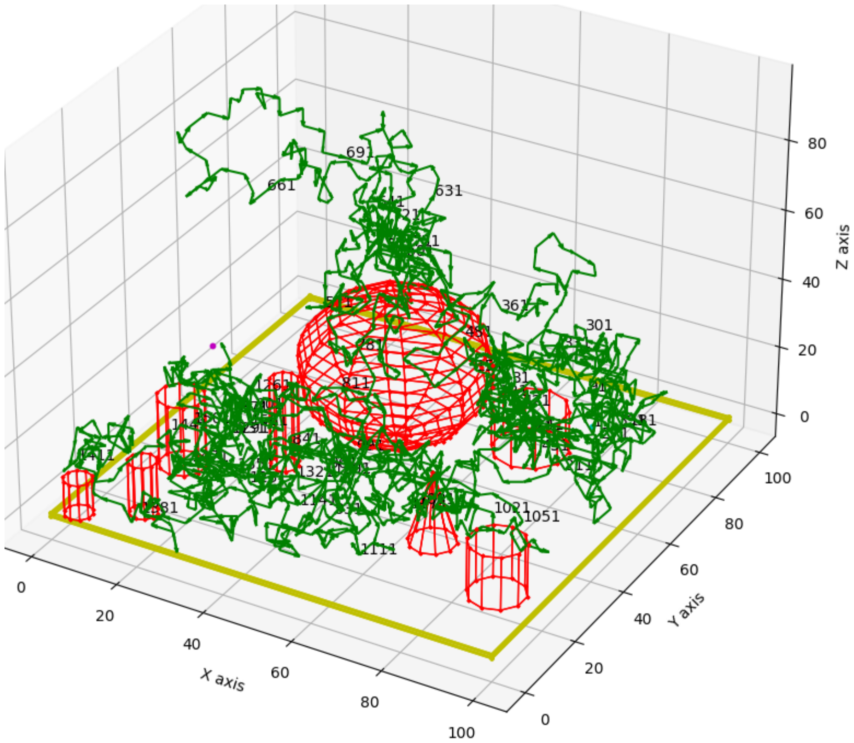
### 6.3.5 Biased RRT

Similar implementation of RRT is done here, except the random points are chosen with a bias. The logic chosen for bias of random point is captured in Algorithm 6. The vector of movement of UAV which makes the least angle with the vector to goal from the current point, is chosen as bias. The point distant along with this vector is chosen as the random point. In case of a collision with the returned point, $collisionCnt$ variable is increased. If the returned point is in the collision-free region, then $collisionCnt$ is set to -1. In the case of collision, the *RandomPointWithBias* returns random point along any vector of movement. After 10 collisions with a random point, $collisionCnt$ is reset to -1. The implementation of the logic can be found in function *runToGoalAlongTheVector* listed in Appendix C 473.

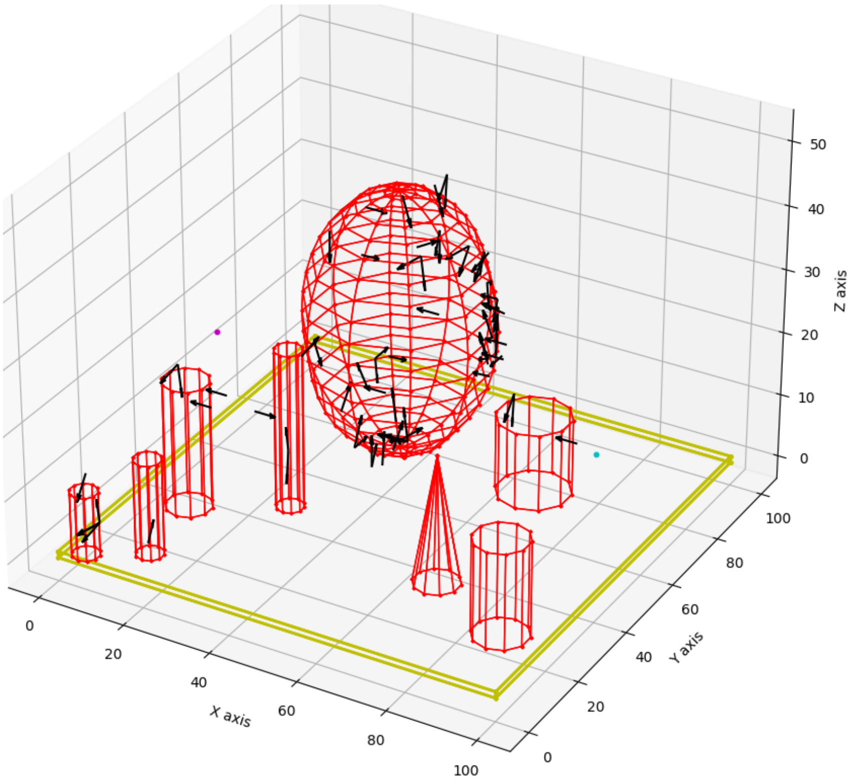
**Algorithm 6** Chosing random point with bias for RRT Algorithm.

**procedure** RANDOMPOINTWITHBIAS($collisionCnt$, $currentPoint$, $goalPoint$, $distance$, $vectorsOfMovement$)
    **if** $collisionCnt >= 0$ **then**
        $Point \leftarrow RandomPoint(distance, vectorsOfMovement)$
        return $Point$
    **end if**
    intialize $vectorAlongGoal \leftarrow Vector(currentPoint, goalPoint)$
    $minAngle \leftarrow \infty$
    $minVector \leftarrow vectorsOfMovement[0]$
    **for all** $vector$ in $vectorsOfMovement$ **do**
        $angle \leftarrow vector.angle(vectorAlongGoal)$
        **if** $angle < minAngle$ **then**
            $minAngle = angle$
            $minVector = vector$
        **end if**
    **end for**
    $Point \leftarrow$ point $distance$ along $minVector$ from $currentPoint$
    return $Point$
**end procedure**

The algorithm was run with 200 iteration limit. The start point was (80, 78, 5) and the goal point was (7,50, 20). It was able to reach the goal point in the $60_{th}$ iteration. Figure 6.12c shows the created tree by the algorithm. In Figure 6.12a, the actual path

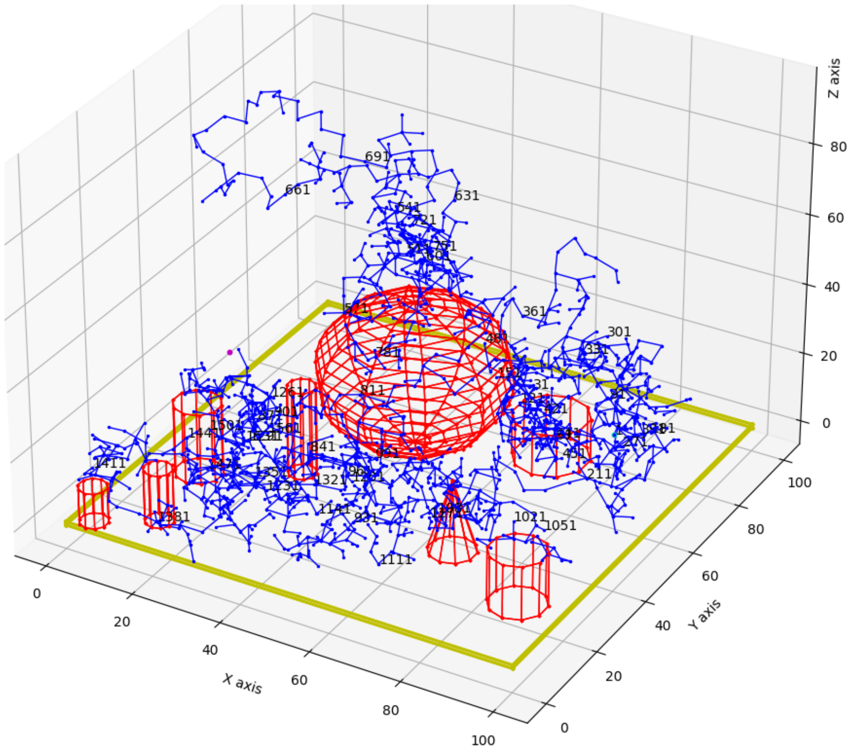



(a) Actual path travelled by UAV.




(b) Paths encountered with obstacles.

FIGURE 6.11: Unbiased RRT algorithm.



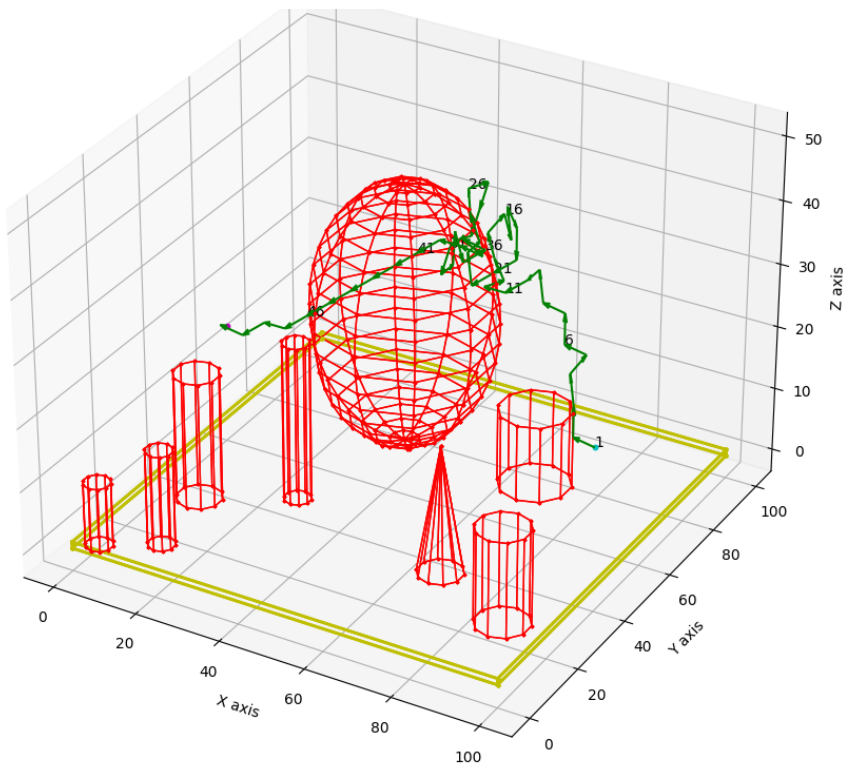

(c) Tree created by the RRT algorithm.

FIGURE 6.11: Unbiased RRT algorithm (continued).

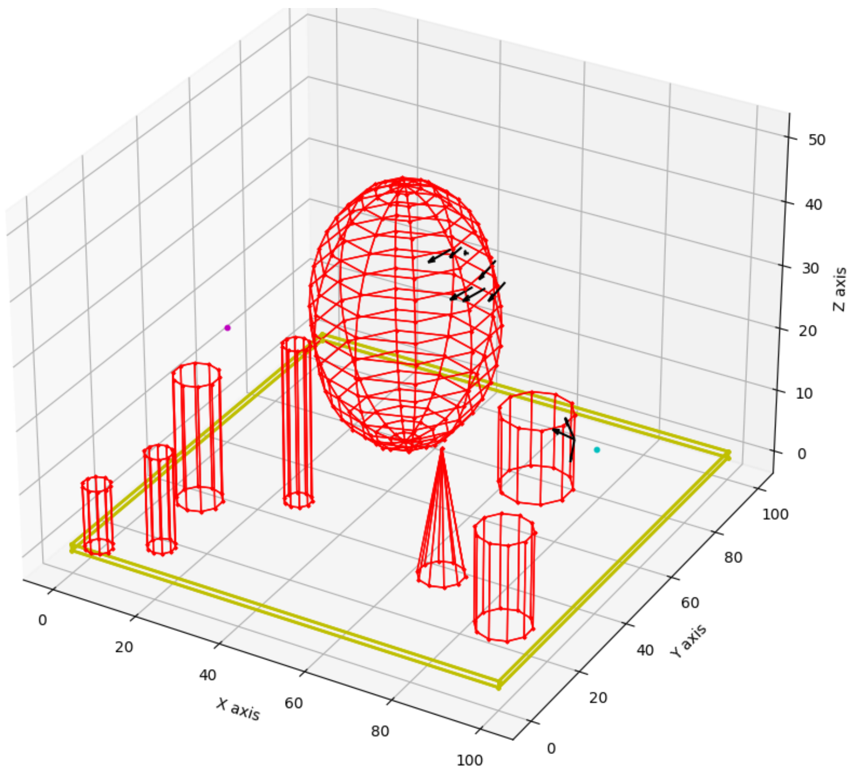
covered by UAV is displayed. The paths which encountered obstacles are shown in Figure 6.12b.

### 6.3.6 Comparison of Biased and Unbiased RRT

RRT is suitable for SAR missions where the details of the environment are not known. Unbiased RRT discussed in Subsection 6.3.4 converges near the goal point when the number of iterations is high. When the iterations tend to $\infty$, the tree covers all the open spaces and hence reaches the goal point. Sometimes the algorithm converges quickly because sometimes the randomness leads to the goal point fast. The algorithm was run for $2000$ iterations, and it reached to the goal in the $1624_{nd}$ iteration. Biased RRT discussed in Subsection 6.3.5 converges very fast and reaches the goal point. There can be various ways of biasing. The chosen bias logic of the random point selection is discussed in Subsection 6.3.5. The bias chooses points towards the goal point along the vectors of movement unless any obstacle is encountered. This convergence of this algorithm is faster than with no bias. The number of iterations required for the biased algorithm is relatively very low than without bias. The algorithm was run for $200$ iterations, and it reached to the goal in the $60_{th}$ iteration.

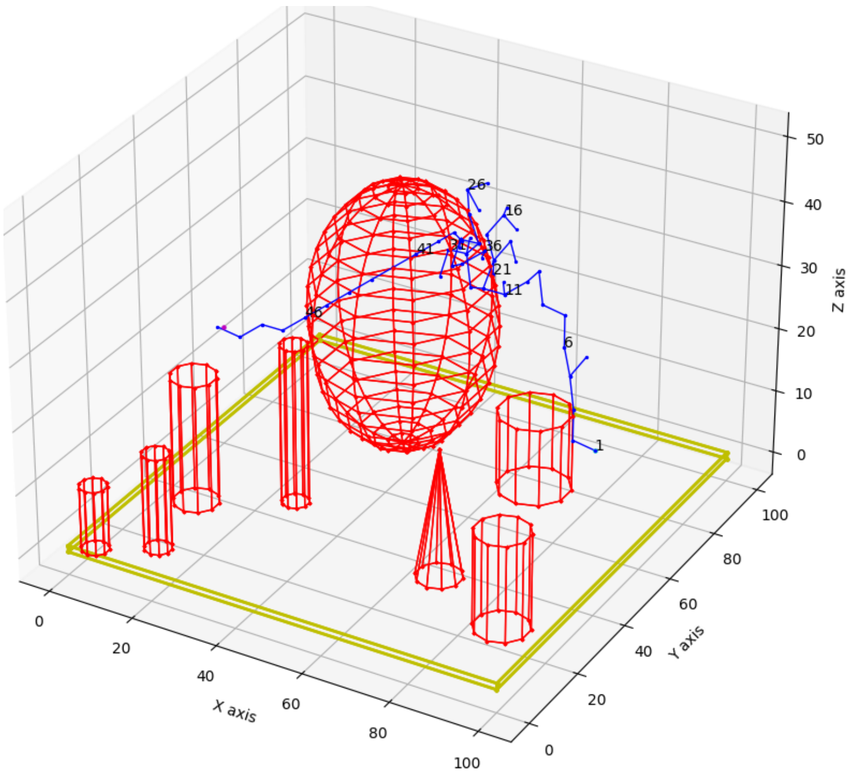



(a) Actual path travelled by UAV.




(b) Paths encountered with obstacles.

FIGURE 6.12: Biased RRT algorithm.




(c) Tree created by the RRT algorithm.

FIGURE 6.12: Biased RRT algorithm (continued).

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

Natural calamities and mishaps are inevitable, and human-created catastrophes emerge every now and then. SAR task forces are made to handle such incidents. UAVs are being used for such SAR missions, helping out to save lives and secure resources. The advancement in technology has increased the efficiency and reliability of the UAVs and their applicabilities in SAR missions.

This thesis has explored the path planning of UAVs in SAR missions. Region coverage algorithms and target reaching algorithms that are applicable for SAR were studied, and simulations were carried out. The simulations were carried out considering a single Unmanned Aerial Vehicle (UAV). A quadcopter was considered for region coverage algorithms, and a 3D figure model was considered for target reaching algorithms. Parallel line, creeping line, spiral long edge first, and spiral short edge first algorithms were considered for region coverage algorithms. Non-deterministic path planning algorithm RRT was considered for target reaching algorithm.

Region coverage algorithms were simulated in Gazebo with PX4 in SITL. Iris 3DS, a quadcopter model, was used. Flight logs of the mission using different algorithms were analyzed using Flight Review, an online flight analyzer. Based on the analysis form logs, comparisons among the algorithms were deduced based on mission time, time to return to base, and the number of turns in the path. The parallel line pattern took the least time to cover the region of interest. It had the least number of turns in the path as well. The creeping line pattern was the slowest among others in terms of mission time and had maximum numbers of turns in the path. Spiral patterns with long edge first and short edge first had almost the same performance. They had the least time to return among other patterns. All the considered patterns were implemented for a constant height and considering the region of interest as 2D plane.

RRT is a non-deterministic approach, i.e., the path is decided during the course of the mission. The decision is made based on the sensors data, and other information. Information of the environment is not required in prior, for RRT, which makes it suitable for SAR missions. RRT algorithm was implemented in 3D the environment by modeling Unmanned Aerial Vehicle (UAV) as a sphere, with 40 surface normals as its possible directions of movement. The algorithm was simulated with and without bias. The unbiased version took too many iterations to find the goal, while the biased version conversed quickly within few iterations.

## 7.2 Future Work

Most SAR missions target wide regions and using a variety of sensors for a 3D mapping or the environment. Approaches described in this thesis apply to some sections of the region of interest. This work considers only single Unmanned Aerial Vehicle (UAV) and 2D planer region for coverage. Systems or humans will be required to divide the region into sections where these algorithms will be applicable. The algorithms do not consider optimization in the path for regions that require a UAV to re-fuel or recharge multiple times during the course of its mission. As mentioned in Section 1.2, path planning in 3D is NP-hard, and there is great scope of research in this area.

To expand the scope of this research, the following is considered for future developments:

- Path planning with multiple UAVs for coverage of larger regions. Factors to be considered would be: division of region, collaboration among Unmanned Aerial Vehicle (UAV), optimization of the path, and refuelling of UAV.
- Path optimization for covering larger regions using a single UAV considering returning to different bases for refuelling or recharging.
- Covering a 3D environment is challenging and is required for SAR missions in urban areas. Urban areas have skyscrapers and high rise buildings, and underground subways and tunnels. Path planning and optimization for such 3D regions using UAVs would be a good area to research.
- During catastrophes and calamities, the map of the region is unknown. SLAM with UAVs in SAR can be explored.

# Appendix A

# MAVSDK Main Code

```
#!/usr/bin/env python3
import sys
import asyncio
from mavsdk import System
from pathPlanner import *



async def mainloop():
    quadcopter = System()
    await quadcopter.connect(system_address="udp://:14540")

    async for state in quadcopter.core.connection_state():
        print(state)
        if state.is_connected:
            print(f"Drone discovered with UUID: {state.uuid}")
            break

    async for val in quadcopter.telemetry.position():
        print (val)
        break

    currPos = val
    polygon = getRectFromStart(300.0,100.0,currPos)

    if pathType == "creeping":
        missionItems = getCreepingLineMissionItem(polygon)
    elif pathType == "SpiralLongEdge":
        missionItems = getLongEdgeSpiralMissionItem(polygon)
    elif pathType == "SpiralShortEdge":
        missionItems = getShortEdgeSpiralMissionItem(polygon)
    else:
        missionItems = getParallelLineMissionItem(polygon)
    missionPlan = MissionPlan(missionItems)

    await quadcopter.mission.set_return_to_launch_after_mission(True)
    await quadcopter.mission.upload_mission(missionPlan)
    await quadcopter.action.arm()
    await quadcopter.mission.start_mission()


pathType = "parallel"
if len(sys.argv) == 2:
    pathType = sys.argv[1]
looper = asyncio.get_event_loop()
looper.run_until_complete(mainloop())
```

# Appendix B

# Coverage Path Planning Code

```python
import sys
from math import sin, cos, sqrt, atan2, radians
from mavsdk import System
from mavsdk.mission import (MissionItem, MissionPlan)
from mavsdk.geofence import Point, Polygon
from mavsdk.telemetry import Position, PositionNed
from pygeodesy.ellipsoidalKarney import LatLon
from pygeodesy.points import boundsOf, centroidOf, isenclosedBy
from shapely import geometry
import matplotlib.pyplot as plt
import geopandas


EAST_COMPASS_ANGLE = 90.0
WEST_COMPASS_ANGLE = 270.0
NORTH_COMPASS_ANGLE = 0.0
SOUTH_COMPASS_ANGLE = 180.0
NORTH_DIR = 0
SOUTH_DIR = 1
EAST_DIR = 2
WEST_DIR = 3

def plotGraph(area,path):
    pts = []
    for pt in area:
        pts.append((pt.lon,pt.lat))
    areaPolygon = geometry.Polygon(pts)
    waypts = []
    for mpt in path:
        waypts.append((mpt.longitude_deg,mpt.latitude_deg))
    wayPath = geometry.LineString(waypts)
    d1 = {'col1': ['path','area'], 'geometry': [wayPath,areaPolygon]}
    gdf1 = geopandas.GeoDataFrame(d1, crs="EPSG:4326")
    #fig, ax = plt.subplots(1, 1)
    gdf1.plot(legend=True, cmap='gnuplot',legend_kwds={'label': "Path in ROI",
                                                        'orientation': "horizontal"})
    return gdf1


def getOppositeCompass(compassAngle):
    if (compassAngle == EAST_COMPASS_ANGLE):
        return WEST_COMPASS_ANGLE
    elif (compassAngle == WEST_COMPASS_ANGLE ):
```

```
        return EAST_COMPASS_ANGLE
    elif (compassAngle == NORTH_COMPASS_ANGLE):
        return SOUTH_COMPASS_ANGLE
    elif (compassAngle == SOUTH_COMPASS_ANGLE):
        return NORTH_COMPASS_ANGLE
    else:
        return compassAngle + 180.0



# Point class is in mavsdk and Latlon class in pygeodesy
# coversion from one another is required
def PointToLatLon(pt:Point):
    return LatLon(pt.latitude_deg, pt.longitude_deg)

def LatLonToPoint(pt:Point):
    return LatLon(pt.latitude_deg, pt.longitude_deg)





def getRectFromStart(length:float, breadth:float, startPoint:Point):
    '''
    requires length and breadth in Km, and startPointA in lat and long
            North
    D---------------------C
    |b                    |
    |r                    |
    |e          +         |   East
    |d                    |
    |t                    |
    |h                    |
    |                     |
    |      length         |
    A---------------------B

    it will return list of polygon points with mavsdk.geofence.Point class
    '''
    # reset points to 0,0 if values are irregular
    if abs(startPoint.latitude_deg) > 90 or abs(startPoint.longitude_deg) > 180:
        startPoint.latitude_deg = 0
        startPoint.longitude_deg = 0
    pointALatLong = PointToLatLon(startPoint)
    pointBLatLong = pointALatLong.destination(length,EAST_COMPASS_ANGLE)
    pointCLatLong = pointBLatLong.destination(breadth, NORTH_COMPASS_ANGLE)
    pointDLatLong = pointALatLong.destination(breadth, NORTH_COMPASS_ANGLE)
    polygonPoints = [pointALatLong,pointBLatLong,pointCLatLong, pointDLatLong]
    # lets get startPointNed
    return polygonPoints


def getRect(length, breadth, centroidPoint):
```

```
    '''
    requires length and breadth in Km, and centroidPoint in lat and long
    D---------------------C
    |                     |
    |   centeroid Point   |
    |         +           |
    |                     |
    |                     |
    A---------------------B

    it will return mavsdk Polygon
    '''
    # if length and breadth are negative, just take abs
    length = math.fabs(length)
    breadth = math.fabs(breadth)
    # reset points to 0,0 if values are irregular
    if abs(centroidPoint.latitude_deg) > 90 or abs(centroidPoint.
   longitude_deg) > 180:
        centroidPoint.latitude_deg = 0
        centroidPoint.longitude_deg = 0
    pointA = Point(centroidPoint.latitude_deg - length, centroidPoint.
   longitude_deg - breadth)
    pointB = Point(centroidPoint.latitude_deg - length, centroidPoint.
   longitude_deg + breadth)
    pointC = Point(centroidPoint.latitude_deg + length, centroidPoint.
   longitude_deg + breadth)
    pointD = Point(centroidPoint.latitude_deg + length, centroidPoint.
   longitude_deg - breadth)
    polygon = Polygon([pointA,pointB,pointC,pointD], Polygon.FenceType.
   INCLUSION)
    return polygon


def checkLimitToDestination(pt, dst, currDir, pathDir, width):
    errorMargin=width/70
    if currDir != pathDir and pathDir != getOppositeCompass(currDir):
        return False
    if pt.distanceTo(dst[0]) <= (width/sqrt(2)) + errorMargin  or pt.
   distanceTo(dst[1]) <= (width/sqrt(2)) + errorMargin:
        return True
    else:
        return False


def getShortPathWidth(pt, dst,pathDir, width):
    '''
        if the width remaining from the fence is less than width, take
   width to 1/2 from the
        fence side.
    '''
    remainWidth = width
    if pathDir == EAST_COMPASS_ANGLE or pathDir == WEST_COMPASS_ANGLE:
        remainWidth = LatLon(0,pt.lon).distanceTo(LatLon(0,dst.lon))
    else:
        remainWidth = LatLon(pt.lat,0).distanceTo(LatLon(dst.lat,0))
    if remainWidth < width:
        #print (f"remaining width = {remainWidth}")
        return (remainWidth - (remainWidth - width/2))
```

```
    else:
        return width

def getPathAttributes(currPt, eastPt, northPt):
    '''
        This function returns list of the short and long side direction in a rectangle.
    '''
    longEdgeCompass = EAST_COMPASS_ANGLE
    shortEdgeCompass = NORTH_COMPASS_ANGLE
    if currPt.distanceTo(eastPt) > currPt.distanceTo(northPt):
        if currPt.lon < eastPt.lon:
            longEdgeCompass = EAST_COMPASS_ANGLE
        else:
            longEdgeCompass = WEST_COMPASS_ANGLE

        if currPt.lat < northPt.lat:
            shortEdgeCompass = NORTH_COMPASS_ANGLE
        else:
            shortEdgeCompass = SOUTH_COMPASS_ANGLE
    else:
        if currPt.lat < northPt.lat:
            longEdgeCompass = NORTH_COMPASS_ANGLE
        else:
            longEdgeCompass = SOUTH_COMPASS_ANGLE

        if currPt.lon < eastPt.lon:
            shortEdgeCompass = EAST_COMPASS_ANGLE
        else:
            shortEdgeCompass = WEST_COMPASS_ANGLE
    return [longEdgeCompass, shortEdgeCompass]

def getParallelLineMissionItem(covAreaPts, sweepWidth=10.0, height = 25, speed = 10):
    currPt = covAreaPts[0]
    mission_items = []
    covAreaBounds = boundsOf(covAreaPts)
    length = currPt.distanceTo(covAreaPts[1])
    longEdge = 0
    shortEdge = sweepWidth;
    breadth = currPt.distanceTo(covAreaPts[-1])
    if length > breadth :
        longEdge = length - (shortEdge);
    else:
        longEdge = breadth - (shortEdge);
    longEdgeCompass, shortEdgeCompass = getPathAttributes(covAreaPts[0], covAreaPts[1],covAreaPts[-1]) # points A and C will determine the direction
    #print(longEdgeCompass, shortEdgeCompass)
    iCnt = 0
    currCompass = longEdgeCompass
    currPt = currPt.destination(sweepWidth/2, shortEdgeCompass)
    currPt = currPt.destination(sweepWidth/2, longEdgeCompass)
    mission_items.append(MissionItem(currPt.lat,
                                     currPt.lon,
                                     height,
                                     speed,
                                     True,
```

```
                                                  float('nan'),
                                                  float('nan'),
                                                  MissionItem.CameraAction.NONE,
                                                  float('nan'),
                                                  float('nan')))
    while  checkLimitToDestination(currPt, [covAreaPts[2],covAreaPts[3]],currCompass, longEdgeCompass, sweepWidth) == False:
        if iCnt % 4 == 0:
            #long edge
            currCompass = longEdgeCompass
            currPt = currPt.destination(longEdge, currCompass)
        elif iCnt %4 == 2:
            #long edge
            currCompass = getOppositeCompass(longEdgeCompass)
            currPt = currPt.destination(longEdge, currCompass)
        else:
            #short edge
            currCompass = shortEdgeCompass
            currPt = currPt.destination( getShortPathWidth(currPt, covAreaPts[2],currCompass,shortEdge) , currCompass)

        mission_items.append(MissionItem(currPt.lat,
                                          currPt.lon,
                                          height,
                                          speed,
                                          True,
                                          float('nan'),
                                          float('nan'),
                                          MissionItem.CameraAction.NONE,
                                          float('nan'),
                                          float('nan')))
        iCnt = iCnt + 1
        #print(iCnt,currCompass,currPt, LatLon(0,currPt.lon).distanceTo(LatLon(0,covAreaPts[2].lon)))

    return mission_items

def getCreepingLineMissionItem(covAreaPts, sweepWidth=10.0, height = 25, speed = 10):
    currPt = covAreaPts[0]
    mission_items = []
    covAreaBounds = boundsOf(covAreaPts)
    length = currPt.distanceTo(covAreaPts[1])
    longEdge = 0
    shortEdge = sweepWidth;
    breadth = currPt.distanceTo(covAreaPts[-1])
    if length > breadth :
        longEdge = breadth - (shortEdge);
    else:
        longEdge = length - (shortEdge);
    shortEdgeCompass,longEdgeCompass = getPathAttributes(covAreaPts[0],covAreaPts[1],covAreaPts[-1]) # points A and C will determine the direction
    #print(longEdgeCompass, shortEdgeCompass)
    iCnt = 0
    currCompass = longEdgeCompass
    currPt = currPt.destination(sweepWidth/2, shortEdgeCompass)
    currPt = currPt.destination(sweepWidth/2, longEdgeCompass)
```

```
    mission_items.append(MissionItem(currPt.lat,
                                     currPt.lon,
                                     height,
                                     speed,
                                     True,
                                     float('nan'),
                                     float('nan'),
                                     MissionItem.CameraAction.NONE,
                                     float('nan'),
                                     float('nan')))
    while  checkLimitToDestination(currPt, [covAreaPts[1],covAreaPts[2]],currCompass, longEdgeCompass, sweepWidth) == False:
        if iCnt % 4 == 0:
            #long edge
            currCompass = longEdgeCompass
            currPt = currPt.destination(longEdge, currCompass)
        elif iCnt %4 == 2:
            #long edge
            currCompass = getOppositeCompass(longEdgeCompass)
            currPt = currPt.destination(longEdge, currCompass)
        else:
            #short edge
            currCompass = shortEdgeCompass
            currPt = currPt.destination( getShortPathWidth(currPt, covAreaPts[2],currCompass,shortEdge) , currCompass)

        mission_items.append(MissionItem(currPt.lat,
                                     currPt.lon,
                                     height,
                                     speed,
                                     True,
                                     float('nan'),
                                     float('nan'),
                                     MissionItem.CameraAction.NONE,
                                     float('nan'),
                                     float('nan')))
        iCnt = iCnt + 1
        #print(iCnt,currCompass,currPt, LatLon(0,currPt.lon).distanceTo(LatLon(0,covAreaPts[2].lon)))

    return mission_items


def getLongEdgeSpiralMissionItem(covAreaPts, sweepWidth=10.0, height = 25, speed = 10):
    currPt = covAreaPts[0]
    mission_items = []
    covAreaBounds = boundsOf(covAreaPts)
    length = currPt.distanceTo(covAreaPts[1])
    longEdge = 0
    breadth = currPt.distanceTo(covAreaPts[-1])
    breadthToCover = breadth
    coveredBreadth = 0

    if length > breadth :
        longEdge = length - (sweepWidth)
        shortEdge = breadth - (sweepWidth)
    else:
```

```
            breadthToCover = length
            longEdge = breadth - (sweepWidth)
            shortEdge = length - (sweepWidth)
    longEdgeCompass, shortEdgeCompass = getPathAttributes(covAreaPts[0], covAreaPts[1],covAreaPts[-1]) # points A and C will determine the direction
    #print(longEdgeCompass, shortEdgeCompass)
    iCnt = 0
    currCompass = longEdgeCompass
    currPt = currPt.destination(sweepWidth/2, shortEdgeCompass)
    currPt = currPt.destination(sweepWidth/2, longEdgeCompass)
    mission_items.append(MissionItem(currPt.lat,
                                     currPt.lon,
                                     height,
                                     speed,
                                     True,
                                     float('nan'),
                                     float('nan'),
                                     MissionItem.CameraAction.NONE,
                                     float('nan'),
                                     float('nan')))

    shortEdgeEpsilon = sweepWidth/70
    while True:
        if iCnt % 4 == 0:
            #long edge
            currCompass = longEdgeCompass
            currPt = currPt.destination(longEdge, currCompass)
            if iCnt != 0:
                longEdge -= (sweepWidth)
        elif iCnt %4 == 1:
            #short edge
            #print(f"short edge = {shortEdge}")
            if shortEdge  <= shortEdgeEpsilon:
                break
            else:
                currCompass = shortEdgeCompass
                currPt = currPt.destination(shortEdge, currCompass)
                shortEdge -= (sweepWidth)
        elif iCnt %4 == 2:
            #long edge
            currCompass = getOppositeCompass(longEdgeCompass)
            currPt = currPt.destination(longEdge, currCompass)
            longEdge -= (sweepWidth)
        else:
            #short edge
            #print(f"short edge = {shortEdge}")
            if shortEdge  <= shortEdgeEpsilon:
                break
            else:
                currCompass = getOppositeCompass(shortEdgeCompass)
                currPt = currPt.destination(shortEdge, currCompass)
                shortEdge -= (sweepWidth)

        mission_items.append(MissionItem(currPt.lat,
                                         currPt.lon,
                                         height,
                                         speed,
```

```
                                      True,
                                      float('nan'),
                                      float('nan'),
                                      MissionItem.CameraAction.NONE,
                                      float('nan'),
                                      float('nan')))

        iCnt = iCnt + 1
        #print(iCnt,currCompass,currPt, LatLon(0,currPt.lon).distanceTo(LatLon(0,covAreaPts[2].lon)))

    return mission_items

def getShortEdgeSpiralMissionItem(covAreaPts, sweepWidth=10.0, height = 25, speed = 10):
    currPt = covAreaPts[0]
    mission_items = []
    covAreaBounds = boundsOf(covAreaPts)
    length = currPt.distanceTo(covAreaPts[1])
    longEdge = 0
    breadth = currPt.distanceTo(covAreaPts[-1])

    if length > breadth :
        longEdge = length - (sweepWidth)
        shortEdge = breadth - (sweepWidth)
    else:
        longEdge = breadth - (sweepWidth)
        shortEdge = length - (sweepWidth)
    longEdgeCompass, shortEdgeCompass = getPathAttributes(covAreaPts[0],covAreaPts[1],covAreaPts[-1]) # points A and C will determine the direction
    #print(longEdgeCompass, shortEdgeCompass)
    iCnt = 0
    currCompass = shortEdgeCompass
    currPt = currPt.destination(sweepWidth/2, longEdgeCompass)
    currPt = currPt.destination(sweepWidth/2, shortEdgeCompass)
    mission_items.append(MissionItem(currPt.lat,
                                      currPt.lon,
                                      height,
                                      speed,
                                      True,
                                      float('nan'),
                                      float('nan'),
                                      MissionItem.CameraAction.NONE,
                                      float('nan'),
                                      float('nan')))
    shortEdgeEpsilon = sweepWidth/70
    while True:
        if iCnt % 4 == 0:
            #short edge
            #print(f"short edge = {shortEdge}")
            if shortEdge <= shortEdgeEpsilon:
                break
            else:
                currCompass = shortEdgeCompass
                currPt = currPt.destination(shortEdge, currCompass)
                if iCnt != 0:
                    shortEdge -= (sweepWidth)
```

```
        elif iCnt %4 == 1:
            #long edge
            currCompass = longEdgeCompass
            currPt = currPt.destination(longEdge, currCompass)
            longEdge -= (sweepWidth)
        elif iCnt %4 == 2:
            #short edge
            #print(f"short edge = {shortEdge}")
            if shortEdge <= shortEdgeEpsilon:
                break
            else:
                currCompass = getOppositeCompass(shortEdgeCompass)
                currPt = currPt.destination(shortEdge, currCompass)
                shortEdge -= (sweepWidth)
        else:
            #long edge
            currCompass = getOppositeCompass(longEdgeCompass)
            currPt = currPt.destination(longEdge, currCompass)
            longEdge -= (sweepWidth)

        mission_items.append(MissionItem(currPt.lat,
                                         currPt.lon,
                                         height,
                                         speed,
                                         True,
                                         float('nan'),
                                         float('nan'),
                                         MissionItem.CameraAction.NONE,
                                         float('nan'),
                                         float('nan')))

        iCnt = iCnt + 1
        #print(iCnt,currCompass,currPt, LatLon(0,currPt.lon).distanceTo(
    LatLon(0,covAreaPts[2].lon)))

    return mission_items




def getSquareSearchMissionItem(covAreaPts, sweepWidth=10.0, height = 25,
     speed = 10):
    centrePt = centroidOf(covAreaPts)
    currPt = LatLon(centrePt[0],centrePt[1])
    mission_items = []
    covAreaBounds = boundsOf(covAreaPts)
    mission_items.append(MissionItem(currPt.lat,
                                         currPt.lon,
                                         height,
                                         speed,
                                         True,
                                         float('nan'),
                                         float('nan'),
                                         MissionItem.CameraAction.NONE,
                                         float('nan'),
                                         float('nan')))
    nextDir = NORTH_COMPASS_ANGLE
```

```
    pathMultiplier = 1
    count = 0
    rotationDir = 0
    while True:
        currPt = currPt.destination(sweepWidth * pathMultiplier ,
nextDir)
        count += 1
        mission_items.append(MissionItem(currPt.lat,
                                          currPt.lon,
                                          height,
                                          speed,
                                          True,
                                          float('nan'),
                                          float('nan'),
                                          MissionItem.CameraAction.NONE,
                                          float('nan'),
                                          float('nan')))
        if rotationDir == 0:
            nextDir += 90
        else:
            nextDir -= 90
        if count % 2 == 0:
            pathMultiplier += 1
        if not isenclosedBy(currPt, covAreaPts):
            break
    return mission_items
```

# Appendix C

# RRT Code

```
#!/usr/bin/env python
# coding: utf-8

import Geometry3D as g3d
import random
import copy
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
#%matplotlib
#get_ipython().run_line_magic('matplotlib', 'widget')




# functions createArrow and graphPLotter are taken from Geometry3D.
    renderer
# and modified to have updates in same graph while progressing

def createArrow(start_pt, end_pt):
    vec = g3d.Vector(start_pt,end_pt)
    distance = g3d.distance(start_pt,end_pt)
    #seg = g3d.Segment(rrt.uavMotionPath[idx],rrt.uavMotionPath[idx+1])
    u = vec.normalized() * g3d.x_unit_vector()
    v = vec.normalized() * g3d.y_unit_vector()
    w = vec.normalized() * g3d.z_unit_vector()
    arw = g3d.render.arrow.Arrow(start_pt.x,start_pt.y,start_pt.z,u,v,w,
     distance)
    return arw

class graphPlotter():
    def __init__(self,instantPlot=True):
        self.rendObj = g3d.Renderer()
        self.fig = plt.figure()
        self.ax = Axes3D(self.fig)
        self.instantPlot = instantPlot

    def show(self):
        self.plotFromRenderer()
        plt.show()

    def plotPoint(self,point_tuple):
        point = point_tuple[0]
        color = point_tuple[1]
        size = point_tuple[2]
        self.ax.scatter(point.x,point.y,point.z,c=color,s=size)
```

```

    def plotSegment(self,segment_tuple):
        segment = segment_tuple[0]
        color = segment_tuple[1]
        size = segment_tuple[2]
        x = [segment.start_point.x,segment.end_point.x]
        y = [segment.start_point.y,segment.end_point.y]
        z = [segment.start_point.z,segment.end_point.z]
        self.plotPoint((segment.start_point,color,size+1))
        self.plotPoint((segment.end_point,color,size+1))
        self.ax.plot(x,y,z,color=color,linewidth=size)

    def plotArrow(self,arrow_tuple):
        x,y,z,u,v,w,length = arrow_tuple[0].get_tuple()
        color = arrow_tuple[1]
        size = arrow_tuple[1]
        self.ax.quiver(x,y,z,u,v,w,color = color,length = length)

    def plotConvexPloygon(self,obj,normal_length=0):
        for point in obj[0].points:
            self.plotPoint((point,obj[1],obj[2]))
        for segment in obj[0].segments():
            self.plotSegment((segment,obj[1],obj[2]))
        if normal_length > 0:
            cpg = obj[0]
            plane = cpg.plane
            normal = plane.n.normalized()
            array = g3d.render.arrow.Arrow(cpg.center_point.x,cpg.
center_point.y,cpg.center_point.z,normal[0],normal[1],normal[2],
normal_length)
            self.plotArrow((array,obj[1],obj[2]))

    def add(self,obj,normal_len=0):
        if self.instantPlot:
            if isinstance(obj[0],g3d.Point):
                self.plotPoint(obj)
            elif isinstance(obj[0],g3d.Segment):
                self.plotSegment(obj)
            elif isinstance(obj[0],g3d.render.arrow.Arrow):
                self.plotArrow(obj)
            elif isinstance(obj[0],g3d.ConvexPolygon):
                self.plotConvexPloygon(obj,normal_length = normal_len)
            elif isinstance(obj[0],g3d.ConvexPolyhedron):
                for cpg in obj[0].convex_polygons:
                    self.plotConvexPloygon((cpg,obj[1],obj[2]),
normal_length = normal_len)
            else:
                raise ValueError('Cannot add object with type:{}'.format
(type(obj[0])))
        else:
            self.rendObj.add(obj,normal_len)

    def plotFromRenderer(self):
        for point_tuple in self.rendObj.point_set:
            self.plotPoint(point_tuple)

        for segment_tuple in self.rendObj.segment_set:
            self.plotSegment(segment_tuple)
```

```python

        for arrow_tuple in self.rendObj.arrow_set:
            self.plotArrow(arrow_tuple)




class UavWorld:
    ''' create a cube/parallelpeiod as ground with height -1
        add many obstacles, like sphere/ cyclinder/ Parallelepiped on the ground
        or above the ground.
        return [ground, list[obstacles], list[xmin,xmax], list[ymin,ymax], list[zmin,zmax]]
    '''
    def __init__(self):
        self.xLimits = [0,100]
        self.yLimits = [0,100]
        self.zLimits = [0,100]
        self.nVal = 10
        self.ground = g3d.Parallelepiped(g3d.Point(self.xLimits[0],self.yLimits[0] , 0), self.xLimits[1] * g3d.x_unit_vector(), self.yLimits[1] * g3d.y_unit_vector(),1 * g3d.z_unit_vector())
        self.Obstacles = []
        self.Obstacles.append(g3d.Cylinder(g3d.Point(4,4,0),3, 10* g3d.z_unit_vector(),n=self.nVal))
        self.Obstacles.append(g3d.Cylinder(g3d.Point(15,10.6,0),3, 15* g3d.z_unit_vector(),n=self.nVal))
        self.Obstacles.append(g3d.Cylinder(g3d.Point(30,40,0),3, 25* g3d.z_unit_vector(),n=self.nVal))
        self.Obstacles.append(g3d.Cylinder(g3d.Point(12,30,0),5, 20* g3d.z_unit_vector(),n=self.nVal))
        self.Obstacles.append(g3d.Cylinder(g3d.Point(70,70,0),8, 12* g3d.z_unit_vector(),n=self.nVal))
        self.Obstacles.append(g3d.Cylinder(g3d.Point(90,20,0),6, 15* g3d.z_unit_vector(),n=self.nVal))
        self.Obstacles.append(g3d.Cone(g3d.Point(70,30,0),5, 20* g3d.z_unit_vector(),n=self.nVal))
        self.Obstacles.append(g3d.Sphere(g3d.Point(50,50,30),20,n1=self.nVal,n2=self.nVal))

    def collisionWithObstacle(self,startPt, endPt, radius, lineCollision=True):
        if lineCollision:
            return self.collisionWithObstacleSegment(startPt,endPt)
        cylinderAxisVec = g3d.Vector(startPt,endPt)
        newPathCylinder = g3d.Cylinder(startPt,radius,cylinderAxisVec,n=self.nVal)
        for idx in range(len(self.Obstacles)):
            collisionArea = g3d.intersection(newPathCylinder,self.Obstacles[idx])
            # if collisionArea var is not null, there are some points intersected
            # so return collided and index of collided obstacle
            if bool(collisionArea):
                return [True,idx]
        return [False,-1]

```

```python
    def collisionWithObstacleSegment(self,startPt, endPt):
        seg = g3d.Segment(startPt,endPt)
        for idx in range(len(self.Obstacles)):
            collisionArea = g3d.intersection(seg,self.Obstacles[idx])
            # if collisionArea var is not null, there are some points intersected
            # so return collided and index of collided obstacle
            if bool(collisionArea):
                return [True,idx]
        return [False,-1]

    def pointInsideObstacle(self,pt):
        for idx in range(len(self.Obstacles)):
            collisionArea = g3d.intersection(pt,self.Obstacles[idx])
            # if collisionArea var is not null, there are some points intersected
            # so return collided and index of collided obstacle
            if bool(collisionArea):
                return [True,idx]
        return [False,-1]

    def addObstacle(self,newObj):
        self.Obstacles.append(newObj)


    def addToRenderer(self,rendObj):
        rendObj.add((self.ground,'y',2))
        for idx in range(len(self.Obstacles)):
            rendObj.add((self.Obstacles[idx],'r',1))





class Uav:
    '''
    This class is creates a model of UAV with default radius, movement vectors, safe height,
    safe distance, delta Travel Distance
    '''
    def __init__(self):
        self.radius = 1 # max dimension of UAV
        self.safeDistance = 1 # safe distance from obstacle
        self.safeHeight = 3 # safe distance from ground
        self.deltaTravelDistance = 5 # distance Uav look forward to travel
        self.moveVectorList = [] # add all the vectors of direction it can move
        # create a sphere with 10 points on longitude and 2 points on half lattitude
        # the normal vectors of the sphere will be direction of the motion
        # add -z and +z vectors for top and down
        s1 = g3d.Sphere(g3d.Point(0,0,0),1,n1=10,n2=2)
        for obj in s1.convex_polygons:
            self.moveVectorList.append(obj.plane.n.normalized())
        xyzVectorList = [g3d.Vector(1,0,0),g3d.Vector(-1,0,0),g3d.Vector(0,1,0),g3d.Vector(0,-1,0),g3d.Vector(0,0,1),g3d.Vector(0,0,-1)]
```

```
        for val in xyzVectorList:
            if not val in self.moveVectorList:
                self.moveVectorList.append(val)




class PathNode:
    '''
    This class creates a node with edges connections and edge weight
    '''
    def __init__(self,nodePoint=None, parentNode=None,nodeIdx = 0, nodeCost=None):
        self.parent = parentNode
        self.selfIdx = nodeIdx
        self.point = nodePoint
        self.cost = nodeCost
        self.children = []




class RrtPlannerUAV:
    def __init__(self, maxIterCnt, deltaDistance, startPt, goalPt, xLims, yLims, zLims, moveVectors):
        self.graph = []
        self.uavMotionPath = []
        self.uavObstaclePath = []
        self.maxIter = maxIterCnt
        self.deltaEdge = deltaDistance
        self.uavMotionPath.append(startPt)
        self.startNode = PathNode(startPt)
        self.startNode.parent = self.startNode
        self.graph.append(self.startNode)
        self.goalNode = PathNode(goalPt)
        self.rendObj = None
        self.moveVectorList = []
        self.xLimits = xLims
        self.yLimits = yLims
        self.zLimits = zLims
        self.currentNodeIdx = 0
        self.treeRendFlag = True
        self.treeRendColor = 'b'
        self.treeRendBrush = 1
        self.failedRendFlag = False
        self.failedRendColor = 'k'
        self.failedRendBrush = 1
        self.pathRendFlag = False
        self.pathRendColor = 'g'
        self.pathRendBrush = 1
        self.errorAllowed = 3
        self.moveVectorList = moveVectors
        #below will be used to save last states of getting valid new random node
        self.lastFailedMoves = []
        self.lastMove = None
        self.vectors = []
        self.translationVector = []
```

```
        self.randNodeCount = 0

    def addRendObj(self, rendObj):
        self.rendObj = rendObj

    def getRandNode(self):
        self.randNodeCount += 1
        return self.noBiasRandNode()
        if self.randNodeCount % 10 == 0:
            return self.translationToGoal()
        else:
            return self.noBiasRandNode()

    def setToLimits(self,newPoint):
        if newPoint.x < self.xLimits[0]:
            newPoint.x = self.xLimits[0]
        if newPoint.x > self.xLimits[1]:
            newPoint.x = self.xLimits[1]
        if newPoint.y < self.yLimits[0]:
            newPoint.y = self.yLimits[0]
        if newPoint.y > self.yLimits[1]:
            newPoint.y = self.yLimits[1]
        if newPoint.z < self.zLimits[0]:
            newPoint.z = self.zLimits[0]
        if newPoint.z > self.zLimits[1]:
            newPoint.z = self.zLimits[1]

    def uniformRandNode(self):
        xVal = random.uniform(self.xLimits[0],self.xLimits[1])
        yVal = random.uniform(self.yLimits[0],self.yLimits[1])
        zVal = random.uniform(self.zLimits[0],self.zLimits[1])
        newPt = g3d.Point(xVal,yVal,xVal)
        node = PathNode(newPt)
        return node

    def resetLastMoveData(self):
        self.lastFailedMoves.clear()
        self.lastMove = None
        self.vectors.clear()
        self.translationVector.clear()
        self.vectors = copy.deepcopy(self.moveVectorList)

    def nodeToGoalAlongVector(self):
        vecToGoal = g3d.Vector(self.graph[self.currentNodeIdx].point,
    self.goalNode.point)
        #vecToGoal = vecToGoal.normalized()
       # for val in self.lastFailedMoves:
       #     if val in vectors:
       #           vectors.remove(val)
        newPointNotFound = True
        pi = 3.14159265
        angleVecDict = dict()
        for idx in range(len(self.moveVectorList)):
            vecAngle = vecToGoal.angle(self.moveVectorList[idx])
            angleVecDict[vecAngle]=self.moveVectorList[idx]
        dictItems = angleVecDict.items()
        sortedItems = sorted(dictItems)
        for val in sortedItems:
```

```
            newPoint = copy.deepcopy(self.graph[self.currentNodeIdx].point).move(self.deltaEdge * val[1])
            self.setToLimits(newPoint)
            if newPoint == self.graph[self.currentNodeIdx].point:
                continue
            else:
                break
        if self.goalLiesInLineSegment(newPoint, self.graph[self.currentNodeIdx].point):
            node = PathNode(self.goalNode.point)
            return node
        else:
            node = PathNode(newPoint)
            return node

    def nodeTowardsGoal(self):
        vecToGoal = g3d.Vector(self.graph[self.currentNodeIdx].point, self.goalNode.point)
        vecToGoal = vecToGoal.normalized()
        newPoint = copy.deepcopy(self.graph[self.currentNodeIdx].point).move(self.deltaEdge * vecToGoal)
        self.setToLimits(newPoint)
        node = PathNode(newPoint)
        if self.goalLiesInLineSegment(newPoint, self.graph[self.currentNodeIdx].point):
            #print("got the goal in line segment")
            node = PathNode(self.goalNode.point)
        #print(node.point)
        return node


    def noBiasRandNode(self):
        vectors = copy.deepcopy(self.moveVectorList)
       # for val in self.lastFailedMoves:
       #     if val in vectors:
       #          vectors.remove(val)
        newPointNotFound = True
        while newPointNotFound:
            vec = random.choice(vectors)
            newPoint = copy.deepcopy(self.graph[self.currentNodeIdx].point).move(self.deltaEdge * vec)
            self.setToLimits(newPoint)

            if newPoint == self.graph[self.currentNodeIdx].point:
                newPointNotFound = True
                vectors.remove(vec)
            else:
                newPointNotFound = False
        if self.goalLiesInLineSegment(newPoint, self.graph[self.currentNodeIdx].point):
            node = PathNode(self.goalNode.point)
            return node
        else:
            node = PathNode(newPoint)
            return node

    def goalLiesInLineSegment(self,ptA,ptB):
        '''
```

```
            (x - x1) / (x2 - x1) = (y - y1) / (y2 - y1) = (z - z1) / (z2 - z1)
            x1 < x < x2, assuming x1 < x2, or
            y1 < y < y2, assuming y1 < y2, or
            z1 < z < z2, assuming z1 < z2
        '''
        if ptB.x == ptA.x or ptB.y == ptA.y or ptB.z == ptA.z:
            return False
        xSlope = (self.goalNode.point.x - ptA.x)/(ptB.x - ptA.x)
        ySlope = (self.goalNode.point.y - ptA.y)/(ptB.y - ptA.y)
        zSlope = (self.goalNode.point.z - ptA.z)/(ptB.z - ptA.z)
        if xSlope == ySlope and ySlope == zSlope:
            # point lines in the line
            # check if point lies in the line segment
            if ptA.x > ptB.x and self.goalNode.point.x >= ptB.x and self.goalNode.point.x <= ptA.x:
                return True
            if ptA.x < ptB.x and self.goalNode.point.x >= ptA.x and self.goalNode.point.x <= ptB.x:
                return True
        return False

    def goalReached(self, node=None):
        if g3d.distance(self.graph[self.currentNodeIdx].point,self.goalNode.point) < self.errorAllowed:
            return True
        return False

    def addFailedNode(self, node):
        self.uavObstaclePath.append(createArrow(self.graph[self.currentNodeIdx].point, node.point))
        if self.failedRendFlag and bool(self.rendObj):
            seg = g3d.Segment(self.graph[self.currentNodeIdx].point,node.point)
            self.rendObj.add((seg,self.failedRendColor,self.failedRendBrush))

    def addNodeToParent(self, newNode, parentNode):
        newNode.Parent = parentNode
        newNode.selfIdx = len(self.graph)
        self.graph.append(newNode)
        parentNode.children.append(newNode.selfIdx)
        if self.treeRendFlag and bool(self.rendObj):
            if parentNode.point != newNode.point:
                seg = g3d.Segment(parentNode.point, newNode.point)
                self.rendObj.add((seg,self.treeRendColor,self.treeRendBrush))
            else:
                print(f"{newNode.point} is same as parent and child")


    def setCurrentNode(self, node):
        if self.pathRendFlag and bool(rendObj):
            seg = g3d.Segment(self.graph[self.currentNodeIdx].point,node.point)
            self.rendObj.add((seg,self.pathRendColor,self.pathRendBrush))
        self.uavMotionPath.append(node.point)
```

```
        self.currentNodeIdx = node.selfIdx

    def findNearestNode(self,newNode):
        dist = g3d.distance(self.graph[0].point,newNode.point)
        nearestIdx = 0
        newNode.cost = dist
        for idx in range(1,len(self.graph)):
            dist = g3d.distance(self.graph[idx].point,newNode.point)
            if dist < newNode.cost:
                newNode.cost = dist
                nearestIdx = idx
        return self.graph[nearestIdx]

    def run(self,world, uavRadius = 1, moveParams = [10,2]):
        '''
            This function will select next node along the vector of movements randomly without any bias
        '''
        for itr in range(self.maxIter):
            newNode = self.getRandNode()
            [isCollision,_] = world.collisionWithObstacle(self.graph[self.currentNodeIdx].point,newNode.point,uavRadius)
            if isCollision:
                if bool(self.lastMove):
                    self.lastFailedMoves.append(self.lastMove)
                self.addFailedNode(newNode)
                continue
            self.lastFailedMoves.clear()
            nearNode = self.findNearestNode(newNode)
            if newNode.point == nearNode.point:
                # got same random point, ignore it
                #print(f"getting same point {newNode.point},{nearNode.point}, idx = {nearNode.selfIdx} ")
                self.setCurrentNode(nearNode)
                pass
            else:
                self.addNodeToParent(newNode, nearNode)
                self.setCurrentNode(newNode)
            if self.goalReached():
                break

    def runToGoal(self,world, uavRadius = 1, moveParams = [10,2]):
        '''
            This function will select next node along the vector joining current point
            and goal point. It will converge quickly
        '''
        randNodeCnt = -1
        randNodeMax = 10
        for itr in range(self.maxIter):
            if randNodeCnt < 0 or randNodeCnt >= randNodeMax:
                newNode = self.nodeTowardsGoal()
                randNodeCnt = -1
            else:
                newNode = self.getRandNode()
                randNodeCnt += 1
            [isCollision,_] = world.collisionWithObstacle(self.graph[self.currentNodeIdx].point,newNode.point,uavRadius)
```

```
            if isCollision:
                if randNodeCnt < 0:
                    randNodeCnt = 0
                if bool(self.lastMove):
                    self.lastFailedMoves.append(self.lastMove)
                self.addFailedNode(newNode)
                continue
            self.lastFailedMoves.clear()
            nearNode = self.findNearestNode(newNode)
            if newNode.point == nearNode.point:
                # got same random point, ignore it
                #print(f"getting same point {newNode.point},{nearNode.point}, idx = {nearNode.selfIdx} ")
                self.setCurrentNode(nearNode)
                pass
            else:
                self.addNodeToParent(newNode, nearNode)
                self.setCurrentNode(newNode)
            if self.goalReached():
                break

    def runToGoalAlongVector(self,world, uavRadius = 1, moveParams = [10,2]):
        '''
            This function will select next node the movement vector which makes least
            angle with vector joining current point and goal point. In case of obstacle
            encounter, 10 random points will be tried till it finds a free space.
            It converges quickly.
        '''
        randNodeCnt = -1
        randNodeMax = 10
        for itr in range(self.maxIter):
            if randNodeCnt < 0 or randNodeCnt >= randNodeMax:
                newNode = self.nodeToGoalAlongVector()
                randNodeCnt = -1
            else:
                newNode = self.getRandNode()
                randNodeCnt += 1
            [isCollision,_] = world.collisionWithObstacle(self.graph[self.currentNodeIdx].point,newNode.point,uavRadius)
            if isCollision:
                if randNodeCnt < 0:
                    randNodeCnt = 0
                if bool(self.lastMove):
                    self.lastFailedMoves.append(self.lastMove)
                self.addFailedNode(newNode)
                continue
            self.lastFailedMoves.clear()
            nearNode = self.findNearestNode(newNode)
            if newNode.point == nearNode.point:
                # got same random point, ignore it
                # print(f"getting same point {newNode.point},{nearNode.point}, idx = {nearNode.selfIdx} ")
                self.setCurrentNode(nearNode)
                pass
```

```
            else:
                self.addNodeToParent(newNode, nearNode)
                self.setCurrentNode(newNode)
            if self.goalReached():
                break



if __name__ == "__main__":
    world = UavWorld()
    uav = Uav()
    #startPoint = g3d.Point(70,50,20)
    startPoint = g3d.Point(80,78,5)
    goalPoint = g3d.Point(7,50, 20)
    #goalPoint = g3d.Point(7,8, 10)

    #RRT with no Bias
    rendObjNoBias = graphPlotter()
    rendObjNoBias.ax.set_xlabel('X axis')
    rendObjNoBias.ax.set_ylabel('Y axis')
    rendObjNoBias.ax.set_zlabel('Z axis')
    rendObjNoBias.add((startPoint,'c',10))
    rendObjNoBias.add((goalPoint,'m',10))
    world.addToRenderer(rendObjNoBias)
    rrtNoBias = RrtPlannerUAV(maxIterCnt=200, deltaDistance=uav.deltaTravelDistance, startPt = startPoint, goalPt= goalPoint, xLims=[0, 100], yLims=[0,100], zLims = [0,100], moveVectors=uav.moveVectorList)
    rrtNoBias.addRendObj(rendObjNoBias)
    rrtNoBias.run(world)
    plt.show()

    # render the actual path travered by UAV
    rendObjNoBias1 = graphPlotter()
    rendObjNoBias1.ax.set_xlabel('X axis')
    rendObjNoBias1.ax.set_ylabel('Y axis')
    rendObjNoBias1.ax.set_zlabel('Z axis')
    rendObjNoBias1.add((startPoint,'c',10))
    rendObjNoBias1.add((goalPoint,'m',10))
    world.addToRenderer(rendObjNoBias1)
    for idx in range( len (rrtNoBias.uavMotionPath) -1 ):
        seg = createArrow(rrtNoBias.uavMotionPath[idx],rrtNoBias.uavMotionPath[idx+1])
        rendObjNoBias1.add((seg,'g',1))
    plt.show()

    # render the paths encounterd with obstacle
    rendObjNoBias2 = graphPlotter()
    rendObjNoBias2.ax.set_xlabel('X axis')
    rendObjNoBias2.ax.set_ylabel('Y axis')
    rendObjNoBias2.ax.set_zlabel('Z axis')
    rendObjNoBias2.add((startPoint,'c',10))
    rendObjNoBias2.add((goalPoint,'m',10))
    world.addToRenderer(rendObjNoBias2)
    for arrw in rrtNoBias.uavObstaclePath:
        rendObjNoBias2.add((arrw,'k',1))
    plt.show()
```

```


    print ( "Number of iterations = ",len (rrtNoBias.uavMotionPath) + len (rrtNoBias.uavObstaclePath))



    # rrt with bias
    rendObjBias = graphPlotter()
    rendObjBias.ax.set_xlabel('X axis')
    rendObjBias.ax.set_ylabel('Y axis')
    rendObjBias.ax.set_zlabel('Z axis')
    rendObjBias.add((startPoint,'c',5))
    rendObjBias.add((goalPoint,'m',5))
    world.addToRenderer(rendObjBias)
    rrtBias = RrtPlannerUAV(maxIterCnt=200, deltaDistance=uav.deltaTravelDistance, startPt = startPoint, goalPt= goalPoint, xLims=[0, 100], yLims=[0,100], zLims = [0,100], moveVectors=uav.moveVectorList)
    rrtBias.addRendObj(rendObjBias)
    rrtBias.runToGoalAlongVector(world)
    plt.show()


    # render the actual path travered by UAV
    rendObjBias1 = graphPlotter()
    rendObjBias1.ax.set_xlabel('X axis')
    rendObjBias1.ax.set_ylabel('Y axis')
    rendObjBias1.ax.set_zlabel('Z axis')
    rendObjBias1.add((startPoint,'c',10))
    rendObjBias1.add((goalPoint,'m',10))
    world.addToRenderer(rendObjBias1)
    for idx in range( len (rrtBias.uavMotionPath) -1 ):
        seg = createArrow(rrtBias.uavMotionPath[idx],rrtBias.uavMotionPath[idx+1])
        rendObjBias1.add((seg,'g',1))
    plt.show()

    # render the paths encounterd with obstacle
    rendObjBias2 = graphPlotter()
    rendObjBias2.ax.set_xlabel('X axis')
    rendObjBias2.ax.set_ylabel('Y axis')
    rendObjBias2.ax.set_zlabel('Z axis')
    rendObjBias2.add((startPoint,'c',10))
    rendObjBias2.add((goalPoint,'m',10))
    world.addToRenderer(rendObjBias2)
    for arrw in rrtBias.uavObstaclePath:
        rendObjBias2.add((arrw,'k',1))
    plt.show()

    print ( "Number of iterations = ",len (rrtBias.uavMotionPath) + len (rrtBias.uavObstaclePath))
```

# References


[1] Kimon P. Valavanis and George J. Vachtsevanos. *Handbook of Unmanned Aerial Vehicles*. Springer Publishing Company, Incorporated, 2014. ISBN 9048197066.

[2] O. Souissi, R. Benatitallah, D. Duvivier, A. Artiba, N. Belanger, and P. Feyzeau. Path planning: A 2013 survey. In *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management (IESM)*, pages 1–8, 2013.

[3] J. Van Sickle. *GPS for land surveyors*. Taylor and Francis, 2001.

[4] Dronecode. Px4 autopilot, . URL https://px4.io/.

[5] Dronecode. Px4 user guide, . URL http://docs.px4.io/master/.

[6] Tom D. Crouch. *Lighter Than Air*. Johns Hopkins University Press, 2009.

[7] Brett Holman. The first air bomb: Venice, 15 July 1849. URL https://airminded.org/2009/08/22/the-first-air-bomb-venice-15-july-1849/.

[8] David Daly. A not-so-short history of unmanned aerial vehicles (uav). URL https://consortiq.com/short-history-unmanned-aerial-vehicles-uavs/.

[9] Bas Vergouw, Huub Nagel, Geert Bondt, and Bart Custers. *Drone Technology: Types, Payloads, Applications, Frequency Spectrum Issues and Future Developments*, pages 21–45. T.M.C. Asser Press, The Hague, 2016. ISBN 978-94-6265-132-6. doi: 10.1007/978-94-6265-132-6_2. URL https://doi.org/10.1007/978-94-6265-132-6_2.

[10] R. H. Major, editor. *Early Voyages to Terra Australis*. The Hakluyt Society, London.

[11] Luke Shimanuki and Brian Axelrod. Hardness of 3d motion planning under obstacle uncertainty.

[12] J. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 49–60, 1987. doi: 10.1109/SFCS.1987.42.

[13] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016. doi: 10.1109/TRO.2016.2624754.

[14] Tauã M. Cabreira, Lisane B. Brisolara, and Paulo R. Ferreira Jr. Survey on coverage path planning with unmanned aerial vehicles. *Drones*, 3(1), 2019. ISSN 2504-446X. doi: 10.3390/drones3010004. URL https://www.mdpi.com/2504-446X/3/1/4.

[15] H.L. Andersen. Path planning for search and rescue mission using multicopters. Master's thesis, Institutt for Teknisk Kybernetikk, Trondheim, Norway, 2014.

[16] M. M. Trujillo, M. Darrah, K. Speransky, B. DeRoos, and M. Wathen. Optimized flight path for 3d mapping of an area with structures using a multirotor. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 905–910, 2016. doi: 10.1109/ICUAS.2016.7502538.

[17] Martin Rosalie, Grégoire Danoy, Serge Chaumette, and Pascal Bouvry. From random process to chaotic behavior in swarms of uavs. 11 2016. doi: 10.1145/2989275.2989281.

[18] M. Rosalie, J. E. Dentler, G. Danoy, P. Bouvry, S. Kannan, M. A. Olivares-Mendez, and H. Voos. Area exploration with a swarm of uavs combining deterministic chaotic ant colony mobility with position mpc. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1392–1397, 2017. doi: 10.1109/ICUAS.2017.7991418.

[19] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[20] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136.

[21] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[22] Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings Ninth IEEE International Conference on Computer Vision*, pages 1403–1410 vol.2, 2003. doi: 10.1109/ICCV.2003.1238654.

[23] G. Klein and D. Murray. Parallel tracking and mapping for small ar workspaces. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 225–234, 2007. doi: 10.1109/ISMAR.2007.4538852.

[24] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. Dtam: Dense tracking and mapping in real-time. In *2011 International Conference on Computer Vision*, pages 2320–2327, 2011. doi: 10.1109/ICCV.2011.6126513.

[25] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 834–849, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10605-2.

[26] María de Miguel Molina and Virginia Santamarina Campos, editors. *The Drone Sector in Europe*, pages 16–42. Springer Nature, 2018. ISBN 978-3-319-71087-7.

[27] DJI. DJI Agras T 16, . URL `https://www.dji.com/no/t16`.

[28] DJI. DJI Agras MG 1, . URL `https://www.dji.com/no/mg-1`.

[29] Amazon Inc. Amazon Prime Air. URL https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011.

[30] Mark Zuckerberg. The technology behind Aquila. URL https://m.facebook.com/nt/screen/?params=%7B%22note_id%22%3A670584150260175%7D&path=%2Fnotes%2Fnote%2F&_rdr.

[31] General Atomics Aeronautical. MQ-9A "Reaper". URL https://www.ga-asi.com/remotely-piloted-aircraft/mq-9a.

[32] Baykar. Bayraktar TB2. URL https://baykardefence.com/uav-15.html.

[33] European Union Aviation Safety Agency. *Easy Access Rules for Unmanned Aircraft Systems(Regulations (EU) 2019/947 and (EU) 2019/945)*, 2021.

[34] Canadian Forces1. *NATIONAL SAR MANUAL B–GA–209–001/FP–001 DFO 5449*, 1998.

[35] International Maritime Organisation. *IAMSAR Manual - International Aeronautical and Maritime Search and Rescue Manual Volume II - Mission Co-ordination*, 2016.

[36] Joshua Fried, Eugene Davydov, and Weilyn Pa. Robotics and motion planning. URL https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/robotics/basicmotion.html.

[37] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984. ISSN 0360-0300. doi: 10.1145/356924.356930. URL https://doi.org/10.1145/356924.356930.

[38] Aaron Knoll. A survey of octree volume rendering methods.

[39] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136.

[40] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991. ISSN 0360-0300. doi: 10.1145/116873.116880. URL https://doi.org/10.1145/116873.116880.

[41] Stevens BL and Lewis FL. *Aircraft control and simulation*. Wiley, 2003.

[42] Guowei Cai, Ben M. Chen, and Tong Heng Lee. *Coordinate Systems and Transformations*, pages 23–34. Springer London, London, 2011. ISBN 978-0-85729-635-1. doi: 10.1007/978-0-85729-635-1_2. URL https://doi.org/10.1007/978-0-85729-635-1_2.

[43] Kiam Heong Ang, G. Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 13(4):559–576, 2005. doi: 10.1109/TCST.2005.847331.

[44] DJI. DJI Phantom 4 Pro V2, . URL https://www.dji.com/no/phantom-4-pro-v2.

[45] Emad Samuel Malki Ebeid, Martin Skriver, Kristian Terkildsen, Kjeld Jensen, and Ulrik Schultz. A survey of open-source uav flight controllers and flight simulators. *Microprocessors and Microsystems*, 61, 05 2018. doi: 10.1016/j.micpro.2018.05.002.

[46] E. Ebeid, M. Skriver, and J. Jin. A survey on open-source flight control platforms of unmanned aerial vehicle. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 396–402, 2017. doi: 10.1109/DSD.2017.30.

[47] Open Source Robotics Foundation. Gazebo Sim. URL http://gazebosim.org/.

[48] Dronecode. Pixhawk, . URL https://pixhawk.org/.

[49] Dronecode. Mavlink, . URL https://mavlink.io/.

[50] Flightgear. Flightgear flight simulator. URL https://www.flightgear.org/.

[51] JSBSim. Jsb flight simulator. URL http://jsbsim.sourceforge.net/index.html.

[52] jMavSim. jmavsim simulator. URL https://github.com/PX4/jMAVSim.

[53] Airsim. Airsim simulator. URL https://microsoft.github.io/AirSim/.

[54] Dronecode. Qgroundcontrol, . URL http://qgroundcontrol.com/.