TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

Simulation and Real Time Implementation of Visual SLAM in an Indoor Environment using a Quadcopter

By:
Anil Kumar Shah (077BAS002)
Bibek Yonzan (077BAS005)
Lucky Babu Jayswal (077BAS019)
Samim Khadka (077BAS037)

A PROJECT REPORT TO THE DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE BACHELOR'S DEGREE IN AEROSPACE ENGINEERING

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING
LALITPUR, NEPAL

March, 2025

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a project report entitled **"SIMULATION AND REAL TIME IMPLEMENTATION OF VISUAL SLAM IN AN INDOOR ENVIRONMENT USING A QUADCOPTER"** submitted by **Anil Kumar Shah, Bibek Yonzan, Lucky Babu Jayswal, and Samim Khadka** in partial fulfillment of the requirements for the Bachelor's Degree in Aerospace Engineering.

Supervisor: **Dr. Nawraj Bhattarai**, Associate Professor
Department of Mechanical and Aerospace Engineering
Institute of Engineering, Pulchowk Campus

Supervisor: **Er. Biman Rimal**, Assistant Professor
Department of Mechanical and Aerospace Engineering
Institute of Engineering, Pulchowk Campus

External Examiner: **Er. Bishal Timilsina**, CEO
Nepware Pvt. Ltd.

Head of Department : **Dr. Sudip Bhattrai**, Assistant Professor
Department of Mechanical and Aerospace Engineering
Institute of Engineering, Pulchowk Campus

DATE OF APPROVAL: 9th March, 2025

COPYRIGHT

ABSTRACT

This project focuses on the simulation and real-time implementation of a visual Simultaneous Localization and Mapping (SLAM) algorithm in an indoor environment using a quadcopter. The system integrates a variety of modern technologies including the Intel RealSense D435 depth camera for vision, a Raspberry Pi 4 for onboard processing of data, and a Pixhawk 4 for flight control. The project aims to address the challenges of autonomous navigation in GPS-denied spaces by enabling the quadcopter to map its surroundings and determine its position in real time. The SLAM algorithm, RTAB-Map is first tested and refined in a simulated environment using Gazebo and ROS to ensure its robustness and accuracy before being deployed in a real-world indoor setting. The use of the Intel RealSense D435 enhances the quadcopter's capability to generate detailed 3D maps, while the Raspberry Pi 4 processes the complex SLAM computations. The project successfully implemented RTAB-Map on a quadcopter with a Raspberry Pi 4B companion computer. Autonomous navigation and obstacle avoidance using the A* algorithm was also achieved on the map. This implementation resulted in 2D grid and 3D point clouds maps of the environment with minimal deviations between generated and estimated trajectories. This project demonstrates the feasibility and effectiveness of using a low-cost, off-the-shelf components and open source software to achieve autonomous navigation, contributing to the development of more efficient UAV systems for indoor applications.

Keywords: Autonomous, Gazebo, GPS-denied, RTAB-Map, A, Depth camera, Raspberry Pi, ROS, SLAM, UAV*

ACKNOWLEDGEMENT

This project is prepared in partial fulfillment of the requirement for the Bachelor's degree in Aerospace Engineering. First and foremost, we would like to express our sincere gratitude towards Associate Prof. Dr. Nawraj Bhattarai and Assistant Prof. Biman Rimal, our project supervisors for their constant guidance, inspiring talks and precious encouragement. Without their invaluable supervision and suggestions, it would have been a difficult journey for us. Their useful suggestions for this whole work and cooperative behaviour are sincerely acknowledged.

Special thanks to Dr. Min Prasad Adhikari, Robotics Engineer at Ontario, Canada whose guidance, supervision and support provided invaluable direction through regular online meetings throughout this project. We would also like to thank IIEC for providing us the space to create indoor environment for flight testing.

We would like to thanks to Assistant Prof. Dr. Sudip Bhattrai, Head of Department of Mechanical And Aerospace Engineering and Assistant Prof. Kamal Darlami, Deputy Head of Department of Mechanical And Aerospace Engineering for their invaluable help and support during this duration.

We would like to thank the Department of Mechanical and Aerospace Engineering, Institute of Engineering, Pulchowk Campus for providing us opportunity of collaborative undertaking which has helped us to implement the knowledge gained over these years as major project for fourth year, and develop a major project of our own that has greatly enhanced our knowledge and provided us a new experience of teamwork.

We would also like to thank all of our friends, seniors and juniors who have directly and indirectly helped us in doing this project.

Authors:
Anil Kumar Shah
Bibek Yonzan
Lucky Babu Jayswal
Samim Khadka

TABLE OF CONTENTS

List of Figures

List of Tables

LIST OF ABBREVIATIONS

AR	Augmented Reality
AI	Artificial Intelligence
BLDC	Brushless Direct Current
BRIEF	Binary Robust Independent Elementary Features
CATIA	Computer Aided Three-Dimensional Interactive Application
CES	Center for Energy Studies
CoG	Center of Gravity
COTS	Commercial Off The Shelf
DC	Direct Current
DOF	Degree Of Freedom
DTAM	Dense Tracking and Mapping
DPPTAM	Dense Piecewise Planar Tracking and Mapping
DWA	Dynamic Window Approach
ECEF	Earth Centered Earth Fixed
EKF	Extended Kalman Filter
ESC	Electronic Speed Controller
FAST	Features from Accelerated Segment Test
FCU	Flight Control Unit
FPS	Frames Per Second
FRD	Forward, Right, Down
GPS	Global Positioning System
GUI	Graphical User Interface
GPIO	General Purpose Input/Output
HDG	Heading
HDMI	High Definition Multimedia Interface
HITL	Hardware in the Loop
I2C	Inter Integrated Circuit
IoT	Internet of Things
IR	Infrared
IEEE	Institute of Electrical and Electronics Engineers
IIEC	Incubation, Innovation and Entrepreneurship Center
IMU	Inertial Measurement Unit
KIT	Karlsruhe Institute of Technology
LAN	Local Area Network
LASER	Light Amplification by Stimulated Emission of Radiation

LiDAR	Light Detection and Ranging
LiPo	Lithium Polymer
LPE	Local Position Estimator
LSD	Large-Scale Direct Monocular SLAM
LTM	Long-Term Memory
MATLAB	Matrix Laboratory
NED	North East Down
NUC	Next Unit of Computing
ORB	Oriented FAST Rotated BRIEF
OS	Operating System
PF	Particle Filter
PID	Proportional Integral Derivative
PRM	Probabilistic Road Map
PTAM	Parallel Tracking and Mapping
QGC	QGround Control
RAM	Random Access Memory
RGB	Red, Green, Blue
RGB-D	Red, Green, Blue - Depth
ROS	Robot Operating System
RPP	Regulated Pure Pursuit
RTAB	Real-Time Appearance-Based Mapping
RRT	Rapidly Exploring Random Trees
RX	Receiver
SBC	Single-board Computer
SDF	Simulation Description Format
SDK	Software Development Kit
SITL	Software in the Loop
SLAM	Simultaneous Localization and Mapping
SONAR	Sound Navigation And Ranging
SSH	Secure Shell
STM	Short Term Memory
TEB	Time Elastic Band
UAS	Unmanned Aerial System
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
UKF	Unscented Kalman Filter
URI	Uniform Resource Identifier
URDF	Unified Robot Description Format
USB	Universal Serial Bus

V-SLAM	Visual SLAM
VO	Visual Odometry
VINS	Visual Inertial Navigation System
VIO	Visual Inertial Odometry
VR	Virtual Reality
WM	Working Memory
XML	Extensible Markup Language
YOLO	You Only Look Once

CHAPTER 1:
INTRODUCTION

1.1. Background

The advent of Unmanned Aerial Systems (UAS) has impacted several industries, most notably the defense and the transportation industry. In the 2022 Russia-Ukraine war, copious numbers of UASs were used[1]. Amazon, the e-commerce company, is also known to use drones to deliver packages to its customers[2]. Due to its relatively simple design and low cost of manufacturing, quadcopters are growing popular for a variety of missions. Quadrotors or quadcopters are a sub-type of rotor UAVs which, as the name suggests, contain four rotors in either an 'X' or a '+' configuration. Even more, these UAVs are also being used for indoor and outdoor mapping and real time tracking. Due to their design, they are very efficient and beneficial in missions where human intervention is difficult.

In the present context of UAVs, autonomous navigation is one of the major hurdles in its development. Various groups have been advancing in this field to develop a truly autonomous system without the need of human input. For autonomous aerial vehicles, localization and navigation are the two most important parameters in the pursuit of true autonomy. Most autonomous systems these days, such as spacecrafts and road vehicles, rely on a positional system like GPS (Global Positioning System) to localize the vehicle in 3D space. But in a GPS-denied environment, GPS based localization and navigation systems become obsolete. Locations such as indoor and underwater environments, and even in space beyond a certain altitude, GPS signals may not be available at all. This poses as a major problem and a hurdle to overcome in the development of a fully autonomous unmanned aerial system.

To solve this problem, most modern robotics system include a localization and navigation system known as SLAM (Simultaneous Localization and Mapping). SLAM helps in aiding the robot where prior knowledge of the environment is either inaccessible or unknown. In it, techniques such as map building and localization of the robot is applied at the same time to know the pose i.e., position and orientation of the vehicle. SLAM is a feasible solution to the aforementioned problem of navigating a UAV in a GPS denied environment. Using it, a local map of the environment is created and based on that map, the position of the UAV can be estimated. As such, SLAM presents itself as a viable and suitable option for localizing and navigating the vehicle in unknown and unmapped

terrains.

1.2. Problem statement

Autonomous aerial robots such as UAVs require a positioning system like GPS to execute missions and general navigation. But in a GPS denied environment, such as indoors, typical navigation systems fail. To be fully autonomous in a GPS denied environment, the UAV must first be able to build a map of its environment and using that localize itself in it. To address these problems, SLAM arises as a suitable solution. SLAM allows the vehicle to build a map and navigate its surroundings in a safe and efficient manner. Many modern self-driving cars utilize a combination of sensor data and complex algorithms to help navigate its environments. Inherently, SLAM itself is a complex problem[3]. To implement a suitable and efficient algorithm to create a reliable map is no easy endeavor. Problems such as the utilization of sensor data to create a definite map in a short period of time is also apparent. Other problems like the time complexity and sensor noise/errors also arise. As such the problem at hand is to implement a SLAM algorithm in a drone and produce a reliable map to find the most suitable path for the navigation of the drone.

1.3. Objectives

The general objective is to fabricate a quadcopter, and implement a suitable visual SLAM algorithm to produce a map of the environment. From the obtained map, using a path-planning algorithm, the quadcopter will select a feasible path while avoiding the obstacles in the environment.

1.3.1. Main Objective

- To fabricate a customized quadcopter and perform simulation and implementation of visual SLAM in an indoor environment

1.3.2. Specific Objective

- To fabricate a quadcopter with an integrated flight controller and D435 RGB-D Camera
- Choose a suitable SLAM algorithm and implement it on the onboard computer for localization and mapping
- Verify SLAM by simulating and mapping a controlled indoor environment
- Integrate a suitable and feasible path planning algorithm to avoid obstacles and navigate autonomously in the environment

1.4. Significance of Research

Ever since its conception, SLAM has been utilized in various different fields, from autonomous vacuum cleaners to self-driving cars and beyond. SLAM as a whole is a growing field, with plenty of innovation. SLAM, in its current state, have been applied in the following areas[4]:

1. Rover Localization and Navigation
2. Deep Space Exploration
3. Indoor navigation
4. Autonomous navigation over unknown areas

Considering the above areas, SLAM can be applied in domains such as:

1. Search and rescue missions where humans cannot intervene
2. Planetary and Underwater Exploration Mission
3. Exploring narrow deep caves where typical ground vehicles and humans are too big to explore
4. In military for surveillance and reconnaissance mission where GPS signals are affected
5. In warehouses for improving inventory management and reducing human error
6. In AR and VR devices to overlay digital information on the real world by understanding the physical environment in real-time
7. For autonomous self driving car to avoid obstacles by localizing and mapping the environment

1.5. Feasibility Analysis

1.5.1. Economic Feasibility

The total budget for this project is expected to be Rs 1,47,300. Typically, commercialized market, ready to fly drones for this similar mapping and navigation job cost around Rs 3,00,000-10,00,000. This shows our project is very reasonable in terms of cost and economic standpoint in the current market scenario.

1.5.2. Technical Feasibility

In terms of technical feasibility of the project, there are no major concerns regarding fabrication, as all essential components—frames, motors, propellers, power modules, sensors, and flight control computers—are readily available in the market. Only potential technical difficulty can arise in what kind of visual sensor to use. This decision will be based on the specific environmental conditions, user requirements and budget constraints.

1.5.3. Operational Feasibility

As this project is for indoor applications, regulatory authorities concern related to flying drones are minimized. Indoor environments comes under user jurisdiction which exempts it from various flying related regulations and restrictions.

1.6. System Requirements

1.6.1. Hardware Requirements

1. **Flight Controller Unit**

   The Pixhawk 4 is selected as the flight controller for this project, as it delivers incredible performance, flexibility, and reliability for controlling any autonomous vehicle and is mainly intended for use with UAVs and robotic platforms. It can also record flight parameters and preserve the flight history in onboard SD card or via the MAVlink protocol. The Pixhawk uses FRD (X Forward, Y Right, Z Down) as a body frame of reference. QGC(QGround Control) is an open soure Ground Control Station which is used for controlling and monitoring drones. QGC and Pixhawk are linked to offer an integrated drone control platform. The integration overall improves the user experience for controlling autonomous systems and makes possible a number of crucial features easily, otherwise difficult.

2. **Companion Computer**

   For the purpose of this project, the Raspberry Pi 4 microprocessor is selected as the companion computer for the quadcopter. Raspberry Pi 4 is a fourth generation single-board computer having adaptability for a wide range of applications and projects. Developed by the Raspberry Pi foundation, the Raspberry Pi is an affordable small single-board computer (SBC), in which the microprocessor, I/O functions, memory, and other features are all built on a single circuit board. The RAM for the Raspberry Pi is built in at a predetermined and non expandable amount. Furthermore, the series of computers come with a set of GPIO (general purpose input/output) pins, which allow for the control of electronic components and sensors for Internet of Things (IoT) applications.

   The Model B from the Raspberry Pi 4 series in figure 1.6.1 is a product from 2018. It's key features include a 64-bit quad-core A72 processor with processing speeds up to 1.8 GHz, 8GBs of RAM and 4K display resolutions via a micro-HDMI port. It also supports 4Kp60 hardware video decoding , dual-band 2.4/5.0 GHz wireless network, gigabit Ethernet, and USB 3.0 ports.

   Most Linux based operating systems compatible with the Raspberry Pi come in

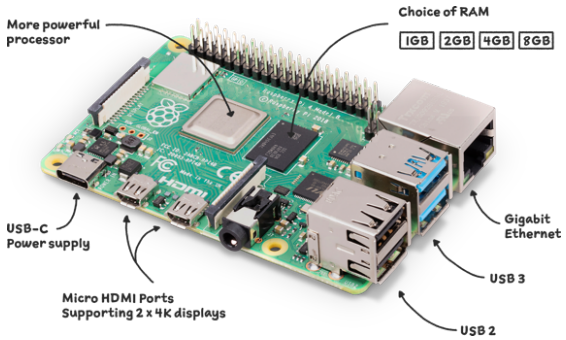


Figure 1.6.1: Raspberry Pi 4B [5]

two different versions: the desktop version, a version with a desktop and command line interface, or the server version, a lightweight version with just a command line interface. Out of the several Linux distributions available for the Raspberry Pi, Ubuntu MATE was selected for this project. Linux is well suited for this project as it focuses on security and stability, as compared to mainstream operating systems. And Ubuntu MATE was selected for its lightweight and resource-efficient features. Linux is crucial to the project as drivers for various sensors are readily available. This makes it easy to attach and use applications for various purposes.

3. **Intel RealSense Depth Camera**
   The Intel RealSense D435 is an RGB-D camera which has primarily been used in robotics and augmented/virtual reality applications. Images from standard digital cameras are output as a 2D grid of pixels. Each pixel is associated with a numerical value ranging from 0 to 255. Using the red-green-blue system (RGB), for example, the group (255, 0, 0) would mean a bright red pixel. However, for a depth camera the pixels have a different numerical value associated with them. Along with the RGB value, RGB-D cameras like the D435 have an additional 'depth' value, which is the distance between the object and the camera. This 'depth' value provides the camera the necessary numerical value needed to calculate the depth of the image. The most popular methods to calculate depth are the Structured Light, Coded Light, and Stereo Depth methods. Structured light and coded light depth cameras both rely on an emitter projecting infrared light onto the environment. The projected light is patterned and the projected pattern is known. Based on the known projected pattern, the camera sensor sees the pattern in the scene thus providing the depth information for the scene.

   The basis of stereo depth cameras is the concept of stereo vision, which compares the marginally different views from each eye to simulate how human eyes sense

Figure 1.6.2: Intel RealSense D435 [6]

depth. Its two lenses are arranged such that they are apart from one another, and though from slightly different viewpoints, these lenses capture photographs of the same scene. Analyzing the apparent shift in object location relative to one another, it compares the differences between these two images to determine the depth information of the objects in the scene.

4. **Airframe**
   The mechanical structure (body) of quadcopter is its airframe. It is the skeleton upon which the rest of components i.e., Flight Controller, Receiver (RX), Electronic Speed Controller (ESC), propellers, motors is attached. It is designed to fulfill specific requirements such as payload capacity, flight time and maneuverability.

5. **Brushless Motor and Propeller**
   A synchronous motor powered by direct current (DC) is known as a brushless DC electric motor (BLDC). The permanent magnet rotor rotates in space because magnetic fields are created when DC currents are switched to the motor windings with the use of an electronic controller. The controller adjusts the amplitude and phase of the current pulses to control the motor's speed and torque.
   Propellers convert rotary motion into thrust. By spinning and generating airflow, drone propellers create a pressure differential between their top and bottom surfaces, which helps the quadcopter gain altitude.

6. **ESC and LiPo Battery**
   ESC known as electronic speed controller is an electronic device used to regulate and control the speed and direction of electric motors. Most commonly used in robotics projects and radio-controlled models, it functions by interpreting signals from a microcontroller or microprocessor, then delivering the appropriate power to the motor, usually brushless DC motors. LiPo batteries are made of a polymer electrolyte, which makes them more flexible and lighter than Li-ion batteries. They have higher energy density hence, the popular choice in drones.

7. **LIDAR(VL53L1X)**

   A Time of Flight (ToF) ranging sensor that uses laser technology to detect distances precisely is the VL53L1X module. It is a multisensor module that combines a microcontroller, detector, and laser emitter. It incorporates ST's second generation FlightSense proprietary technology and a state-of-the-art SPAD array (single photon avalanche diodes). It's dimension is 4.9x2.9x1.56 *mm*. It is fast and accurate long distance ranging upto 400*cm* distance measurement and upto 50Hz ranging frequency. It's typical full field of view is 27°. It uses a single axis to measure distance in a straight line. Unlike a 3D LiDAR, it does not offer depth mapping or numerous points in a scene.

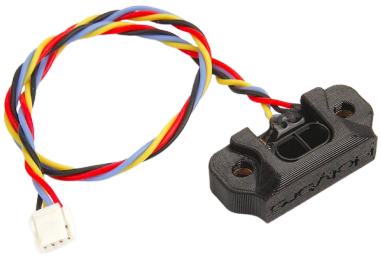


Figure 1.6.3: VL53L1X Lidar [7]

A small infrared laser pulse is released by the sensor. The time it takes for the pulse to strike an object and bounce back is then determined. The sensor calculates the distance using the speed of light.

1.6.2. Software Requirements

1.6.2.1. ROS

ROS (Robot Operating System) is a collection of tools and software libraries for creating robot applications. Hardware abstraction, low-level device control, modularity, inter-process messaging, and package management are among the services it offers that is anticipated from a robust operating system. Additionally, it offers libraries and tools for acquiring, writing, and executing code on numerous systems. The ROS environment and its supporting libraries make it simple and quick to construct autonomous robotic systems. One of the main advantages of building robots with ROS is the collection of drivers and already existing algorithms within the system. It is a community of highly trained and pragmatic robotics applying advanced robotics to every conceivable application while sharing their solution with each other. ROS 1 was used over ROS 2 because of its long term supports for SLAM algorithms such as gmapping (2D SLAM), and RTAB-Map are available in ROS Noetic. It is simpler for new users to begin with SLAM on ROS Noetic since a large number of SLAM-specific research publications

and tutorials are based on ROS 1. In ROS 1, these technologies have undergone a great deal of testing, adoption, and documentation and a thriving developer community also exists. Thus, ROS 1 was the choice for this project as well.

1. **Nodes:** A node is the basic computational unit of processor running in ROS. It is a process that uses ROS messages to interact with other nodes while carrying out a certain task. A node can offer other nodes a service, publish messages, or subscribe to messages via topics. For each use, ROS advises building a single node, and it is advised to design for simple re-usability. The software used to control mobile robots, for instance, is divided into specialized functions. Every function, including sensor drive, sensor data conversion, obstacle detection, motor drive, encoder input, and navigation, uses a specialized node. It is an executable that can be created by the user using either C++ or Python.

2. **Publisher:** ROS uses "Publisher" nodes to transmit message throughout the ROS network. The Publisher node takes either raw incoming data from a sensor or calculated data from the processor and publishes it to a ROS topic. Publisher nodes allow fast intra-communication between ROS nodes and between ROS ecosystems and therefore are an integral part to any ROS robotics build.

3. **Subscriber:** "Subscriber" refers to the process of getting relevant messages related to the subject. In addition to receiving publisher information that publishes related topics from the master, the subscriber node registers its own information and subject with the master. Once publisher data is received, the subscriber node sends a direct connection request to the publisher node and gets messages from the linked publisher node. It is possible for a subscriber to be declared more than once in a single node. Subscribers nodes are quite essential to any ROS build as it is one of the prominent ways for effective communication and retrieving data in between systems in a robot.

4. **Topic:** A ROS topic is a designated communication channel that nodes use to exchange messages. By decoupling nodes, topics eliminate the need for direct connections, enabling one or more publishers to submit data and one or more subscribers to receive it.

5. **Action:** For asynchronous bidirectional communication, ROS actions is an additional message communication technique, when it takes longer to react to a request and intermediate answers are needed until the outcome is provided, ac-

tion is utilized. Service files and action files have a similar structure. But along with the goal and outcome data sections—which are shown in the service as request and response, respectively—there is now a feedback data component for intermediate responses. The action client establishes the objective of the action, while the action server carries out the action defined by the objective and provides the action client with feedback and the outcome.

Receiving the client's aim and providing feedback and results is the responsibility of the "action server". In addition to receiving result or feedback data as inputs from the action server, the "action client" is responsible for sending the objective to the server.

1.6.2.2. Gazebo

Gazebo is an open-source 3D physics simulator widely used in the robotics community for simulating robots. Gazebo creates realistic simulation of robotic systems and their surrounding without the need of physical hardware. It is a free and open source software with accurate physics providing high-performance real-world simulations.

Drones can be used in simulated environments with dynamic obstacles, realistic lighting, and customizable terrains in Gazebo's virtual world. We can test how well a SLAM algorithm works at creating a map and locating the drone by simulating drone motion, which reduces the risk of damaging physical equipment during early developmental stages.

1.6.2.3. MATLAB

MATLAB is a matrix based programming environment for scientific computing. It is a technical computing language designed for data visualization and high performance data analysis. Data obtained from SLAM algorithms can be plotted using MATLAB in an efficient and easy manner.

1.6.2.4. Visual Studio Code

Visual Studio Code is a source-code editor developed by Microsoft. It can be run on a variety of operating systems and provides support for multiple languages like Python and C/C++. It provides efficient and comprehensive tools for code editing and writing. VS Code was chosen for its intelligent code completion and simple yet effective extension library, which provides support for ROS development, over its more popular contemporaries.

1.6.2.5. CATIA

Computer Aided Three-Dimensional Interactive Application (CATIA) is a software for product development from conceptualization, design and engineering to manufacturing. It is used for designing and 3D modeling various parts as per requirement. For the purpose of this project, CATIA was used to design models of the quadcopter airframe and sensor mounts. The designs of propeller-guard and depth camera mount were done in CATIA, which were used for protecting the motors, propellers and camera in the occurrence of a crash.

CHAPTER 2:
LITERATURE REVIEW

Simultaneous Localization and Mapping (SLAM) is a technique used in robots to map the environment around the robot and localize itself in it. It is used to perform applications like search and rescue, planetary observations, precision agriculture, and autonomous navigation in crowded environments [8]. SLAM uses data recorded from sensors such as LASER, SONAR, and cameras to build a map of the environment and predict the orientation and position of the robot relative to the map. The selection of sensors is based on criteria such as payload capabilities, processing power, and environmental parameters, with stereo cameras chosen for this project due to their higher efficiency and lower weight [9]. Mapping involves generating a map of the environment using sensors before the robot explores the area, while localization helps the robot calculate its trajectory and avoid obstacles based on the obtained map. SLAM, ever since its origin, has been evolving with time and its growth has been exponential. The first approach to SLAM appeared in 1986, which introduced the concept of implementing the estimation of spatial uncertainty [10]. Then in 1991, EKF-SLAM was implemented using a probabilistic approach [11].

SLAM has been an interesting challenge in the fields of mobile robotics and Artificial Intelligence (AI). The "SLAM problem" was incepted at an IEEE Robotics and Automation Conference in 1986 [12]. At the time, the problem was not of optimization, but one of application of the numerous theoretical methods involving mapping and localization. Over the years, solutions to solve this problem have been successful, however the problem has also grown significantly more complex. With the simultaneous development in UAVs (Unmanned Aerial Vehicles) and AI, the SLAM problem has also transformed into a multifaceted challenge. The idea of SLAM has evolved from probabilistic estimations in the past to robotic perception in dynamic environments at present. After it's inception, SLAM was treated as two separate problems i.e., localization and mapping. However, a monumental paper by Smith et al. [13] interpreted the two disciplines to form a single solution for the problem. The paper imagines the two individual problems of localization and mapping as one single probabilistic problem, through which a mobile robot can estimate its relative position in a dynamic environment and its relations to the objects in the map. Another monumental paper by Whyte et al. [14] provided the breakthrough needed to combine and converge the two problems of localization and mapping. The paper discussed the problems of using beacon-based localization such GPS to estimate the vehicle's position and presented the solution of

combining sensors (“sensor fusion”) and using a Kalman filter based algorithm for better localization of the robots. The paper also coined the term SLAM, defined as “incrementally both building a map and using that map to locate its position relative to the robot’s start point”.

With the rapid development of UAV technologies, UAVs have become a prime application area for SLAM. From driverless cars to UAVs in emergency scenarios, the concept of SLAM has been realized to some extent. SLAM has been crucial for UAVs to determine their position and map an unknown environment at the same time. This feature of SLAM is important for drones to be able to estimate its own position and create a map of its environment, particularly in GPS-denied environments. Out of the numerous SLAM algorithms developed over the years, Visual SLAM (V-SLAM) and LiDAR SLAM are the most prominent ones in use with UAVs. V-SLAM refers to the use of 2D image and depth data obtained from a camera sensor alongside the pose (position and orientation) data obtained from the IMU (Inertial Measurement Unit) sensor to estimate the motion of the camera and as such reconstruct its local environment [15]. LiDAR SLAM is a more sophisticated approach compared to normal V-SLAM algorithms, however the map built by the former is a lot less detailed as opposed to the one built by the latter [16]. It is interesting to note that even though V-SLAM is popular compared to LiDAR SLAM, the latter provides significantly more accuracy and low drift estimation. One of the key factors behind V-SLAM’s adoption can also be attributed to its ease of access and it’s modest pricing.

A standard V-SLAM algorithm consists of two halves: “a front-end and a back-end” [17]. The “front-end” is known as Visual Odometry (VO) and is used to estimate the camera pose of the robot. The principle of VO is to incrementally estimate the egomotion of the robot by detecting the changes in motion induced in the sensors onboard the robot [18]. Once the camera and other sensors in a robot have acquired data, or “images” in this case, the VO then estimates the position and the camera movement based on adjacent images and generates a rough map. The back-end then filters the obtained camera poses and optimizes the rough map to create a fully optimized map. The “backend” also deals with the noise obtained from the sensors. Visual SLAM can mostly be categorized into Monocular, RGB-D and Stereo approaches [3]. Using a single camera sensor for data acquisition is known as the monocular approach. Using a stereo camera or a camera with two lenses is known as the Stereo approach and combining a camera with an infrared sensor is termed as RGB-D SLAM. In recent years there has been rapid progress in developing novel algorithms to solve the SLAM problem and they can be categorized into two classifications, direct methods and feature based methods [15]. MonoSLAM is a popular example of a monocular SLAM algo-

rithm that uses the Extended Kalman Filter (EKF) algorithm to generate a map of the environment. The generated environment however suffers from the same problems as EKF itself: linearization [19]. The linearization causes issues during loop closure. Furthermore, due to the O(n^2) complexity of the EKF algorithm, it can only create sparse maps [20]. This limits the algorithm from being used in large environments. To counter this, a non-linear algorithm such as the Unscented Kalman Filter (UKF) can be applied. Another approach to counter the linearization issue was the application of a Particle Filter (PF) based monocular SLAM. Applying a PF did improve the map quality however at the expense of computational power. One of the most famous applications of a PF based monocular SLAM is ORB-SLAM. ORB-SLAM is based on the PTAM (Parallel Tracking and Mapping) framework, which analyzes and extracts features from a 2D-image [21]. ORB-SLAM is a feature based SLAM, which extracts ORB (Oriented FAST, Rotated BRIEF) from images and performs real time loop closing and camera relocalization. ORB-SLAM can be used in a variety of ways: monocular, RBG-D, which combines an RGB image source and an IR (Infrared) image source, and even stereo. Although ORB-SLAM is extremely powerful and simple to use, it creates a sparse map and cannot be used in UAVs [22].

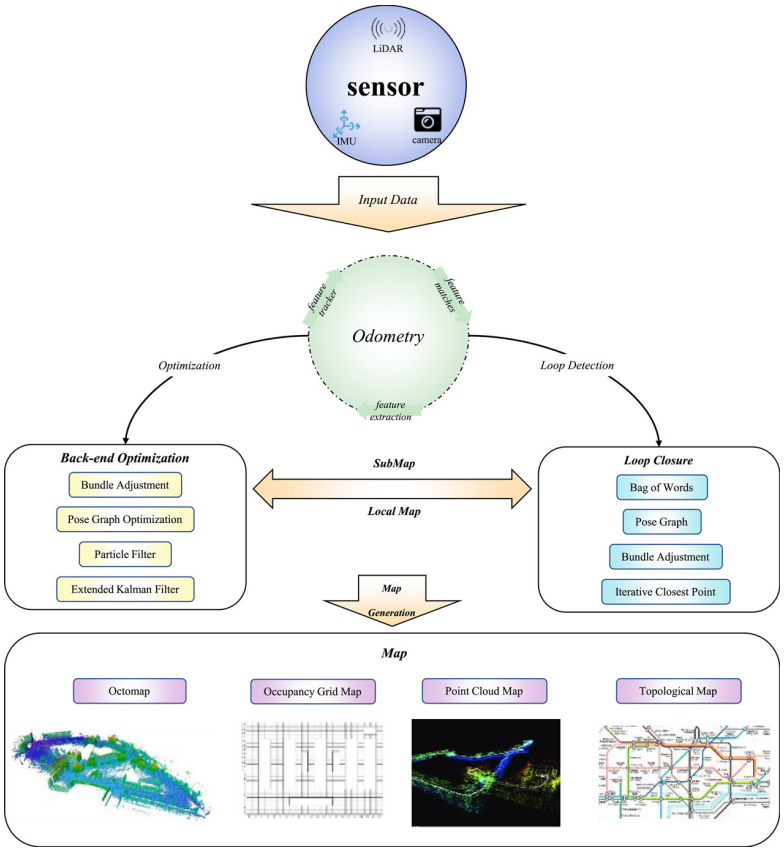


Figure 2.0.1: Overall SLAM Framework [23]

Another approach to V-SLAM is Real Time Appearance Based Mapping or RTAB-Map

for short. RTAB-Map is a graph based, dense RGB-D SLAM solution to the SLAM problem. It utilizes memory management to maintain the graph [24]. The memories consist of "Short-Term Memory (STM)", "Working Memory (WM)" and "Long-Term Memory (LTM)". The STM adds a new node to the graph everytime new data is received and has a fixed size. When the STM reaches its final size, the node with the longest storage time is moved to the LTM [25]. The implementation of SLAM in a quadcopter/UAV is not a novel concept, but the idea of using a quadcopter with RTAB-Map is a relatively new concept. Nevertheless, considering all the parameters such as computational power bandwidth and features, RTAB-Map stands out for UAV applications. Catania [26] discusses the usage of an RGB-D camera, a flight computer (ArduPilot) and a LiDAR with the RTAB-Map algorithm in a ROS Noetic wrapper to map indoor environments. Catania in his paper implements the YOLO algorithm for object recognition and feature extraction. Wu et al. [27] also present the idea of indoor localization using an RGB-D camera along with an IMU for a "vision based UAV". In [27], Wu et al. extract ORB as the features from the images and use EPNP as the motion estimation method. Also, Chen et al. [28] in 2023 published a paper using RTAB-Map for a real time 3D reconstruction of using a vision based UAV i.e., a quadrotor drone equipped with a Pixhawk flight controller and an Intel Realsense D435 depth camera. The paper also presents the idea of using multiple sessions of flight as the RTAB-Map algorithm allows synchronous mapping due to its unique memory management. Finally, Labbé in [29] presents an example of using a quadcopter in a simulated Gazebo environment. The quadrotor in this example simulates an Intel Realsense R200 camera and uses the data from it to avoid obstacles in the virtual environment.

Another important issue to be addressed in a UAV is path planning. Path planning can be defined as the computation of the UAV's trajectory to reach its desired destination or "target" [30]. As there are no onboard pilots in a UAV, the path planning ability of the UAV becomes vital, moreso in an autonomous one. Traditionally UAVs have been restricted to be used within a certain area, along a set path. But as technologies progress, the operating areas and use cases have also become more advanced. The path planning problem has also advanced from a geometric one, fixed objects and set locations, to a more dynamic one, avoiding obstacles while completing a set mission. At the same time the problem as a whole has transformed into a stochastic one[31]. In modern applications, it is not only necessary to just estimate the trajectory of the vehicle but also the pose of the objects in the local environment i.e., obstacles, irrespective of time and location.

Path planning algorithms in UAVs can be categorized into five types: Cooperative Techniques, Coverage and Connectivity, Security, Representation Techniques and Non-cooperative Techniques. The two most important steps in any modern path planning

algorithm is 1) the ability to represent the UAV in a 3D space and identification of the obstacles in the environment and 2) creating a map of the local environment. With the help of SLAM, the UAV can be localized in real time and the map created can then be used to effectively plan the trajectory of the vehicle, all while avoiding obstacles.

Path planning, for this use case, is divided into global and local planning modules. The former generates path on prior map knowledge or limited map and the latter generates path on the subset of the global map including local obstacle avoidance. Different global path planning methods are currently in use. Silhouette proposed by Canny in 1987[32] or Voronoi in 2007[33], utilize Roadmaps based on map analysis. These approaches to global planning assigns values to different regions of roadmaps to find lowest cost path[34]. Other prominent examples of an effective global path planning algorithms are A* and the Dijkstra algorithms. The paper [35] used A* to generate an initial coarse path for a fixed-wing UAV on a 3D map, then refined it using local map data and a search tree built from UAV motion primites. Other prominent examples of an effective global path planning algorithms are A* and the Dijkstra algorithms.

Dijkstra and A* both are solutions to the shortest path problem and are often used to solve town or road problems. A shortest path problem is the problem in which the shortest path between two nodes in a graph is analyzed [36]. Dijkstra and A* both are forms of the greedy algorithm. Dijkstra searches each node in the graph and does not have a negative side cost, whereas A* gives a heuristic estimate for the best route from the initial start point to the end goal. Proposed in 1959, the shortest path solution for Dijkstra's algorithm solves the problem based on route length parameters [37]. Meanwhile, the A* algorithm is the modified form of the Dijkstra's that estimates the optimal path required based on the sum of two functions, the estimated cost required to reach a node and the estimated cost to reach the goal [38].

After having the global path, now local path planning comes into play. Local planning works on a small data set of the global path. The local planner adjusts the global path by generating waypoints that account for dynamic obstacles and vehicle constraints. If the local path is blocked or inaccessible, the planner regenerates the path. To make this regeneration of path more faster, the map is reduced to the local surrounding of the vehicle and updates as it moves around in the environment. Using the updated local map, the local path plans a path which matches closely with the global path[39]. There are different kind of local planning techniques like DWA[40], TEB[41], RPP[42]. DWA planner selects the best velocity command by evaluating possible trajectories based on the robot's dynamic constraints, obstacle avoidance, and goal direction. TEB local planner optimizes a trajectory by minimizing time and distance while considering con-

straints like obstacles and kinematics. The selected local path planning is RPP. It is build upon the framework of pure pursuit algorithm by incorporating speed regulation to ensure smooth and safe path tracking. It selects a lookahead point on the path and calculates the required curvature for steering, similar to Pure Pursuit. However, RPP dynamically adjusts the robot's speed based on environmental conditions, slowing down in sharp turns and confined spaces while allowing higher speeds in open areas[42].

CHAPTER 3:
THEORETICAL BACKGROUND

3.1. Quadcopter Dynamics

Quadcopter dynamics is a big topic in itself with multitude of parameters involved. To model the dynamics of quadcopter some assumptions are needed to simplify the process. The assumptions are made for the forces, torques, motors, propellers, and structural components of the quadcopter. Those assumptions are:

1. The resistance of the motors is negligible.

2. The quadcopter structure is rigid and symmetric.

3. The blade flapping, or the deformation of propeller blades brought on by high velocities and flexible materials, is neglected.

4. The surrounding fluid velocities, such as wind and freestream velocity, are negligible.

5. The quadrotor's center of gravity (CoG) aligns with the origin of the body-fixed frame.

With these assumptions set, the next thing is the frame of reference. Frame of reference is a set of coordinates, such as positions and velocities of a quadcopter, in the frame being observed. There are mainly two different frame of references called the inertial and non-inertial frame of references. The inertial frame of reference is also commonly denoted as inertial frame or the Earth frame (NED: North-East-Down or ECEF: Earth-Centered, Earth-Fixed). The non inertial frame of reference is the body fixed frame, with the origin at the CoG of the quadcopter. Wind frame, Stability frame are some other frame of references. Each reference frame serves a specific purpose in dynamics and modelling of quadcopter.

The quadcopter moves by changing its thrust, where thrust is being generated by the rotation speed of the motor. In the current configuration, pitch which is the lateral motion, is achieved by changing the rotation speed of the motors 2 and 4 in Figure 3.1.1, which causes variation in thrust forces 2 and 4. The longitudinal motion, roll is achieved

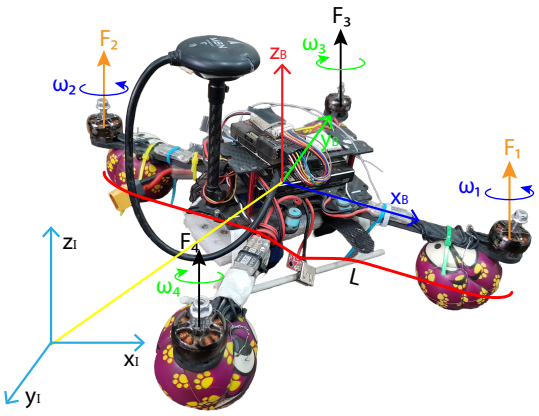


Figure 3.1.1: Isometric View of Quadcopter

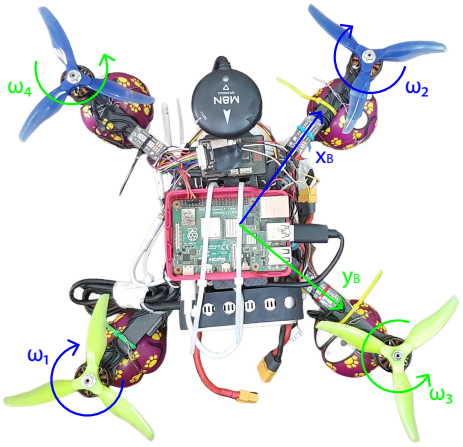


Figure 3.1.2: Top View of Quadcopter

by changing the rotation speed of motors 1 and 4, which causes variation in thrust forces 1 and 4. Yaw motion is generated by the difference in counter-torque between the pairs of propellers (F1, F2) and (F3, F4). This occurs when the clockwise-rotating rotors speed up while the counter-clockwise ones slow down, or the reverse. The overall thrust required for hovering or ascending is achieved by summing the individual forces produced by all the rotating propellers.

3.1.1. Rotor Dynamics

The modelling of the thrust forces and torques generated by the rotors can be represented by a simplified model. Each rotor with angular speed ω generates a force F in a direction perpendicular to the plane of rotation of the rotor given by,

$$F_i = \left(\frac{K_v * K_\tau \sqrt{2\rho A}}{K_t}\omega\right) \simeq k\omega^2 \tag{3.1.1}$$

where K_v and K_t are constants related to the motor properties, ρ is the density of the air, A is the area the rotor sweeps out, K_τ is a constant determined by the blade configuration and parameters, and k is the thrust factor[43].

The quadcopter's total thrust (in the body frame) is determined by adding up all of the forces which is given by

$$T_B = \sum_{i=1}^{4} T_i = k \begin{bmatrix} 0 \\ 0 \\ \sum \omega_i^2 \end{bmatrix} \tag{3.1.2}$$

The torque contributed by the rotor about the body frame z-axis, which is required to

keep the propeller spinning and providing thrust, generated due to drag is given by[44]

$$\tau_i = \frac{1}{2} R\rho C_D A(\omega_i R) \simeq b\omega_i^2 \tag{3.1.3}$$

where R represents the propeller radius, C_D is a drag coefficient, and b is the drag constant.

The total torque around the z-axis is calculated by summing the individual torques produced by each propeller

$$\tau_\psi = b(\omega_1 - \omega 3 + \omega 2 - \omega 4) \tag{3.1.4}$$

The roll and pitch torques is obtained from standard mechanics. The motors on the roll axis are arbitrarily chosen to be i = 1 and i = 3, thus

$$\tau_\phi = \sum r \times T = L(k\omega_1^2 - k\omega_3^2) = Lk(\omega_1^2 - \omega_3^2) \tag{3.1.5}$$

The pitch torque is given by the expression below:

$$\tau_\theta = Lk(\omega_2^2 - \omega_4^2) \tag{3.1.6}$$

where L is the center distance of the propellers. Summing all, the torques in the body frame are found to be:

$$\tau_B = \begin{bmatrix} Lk(\omega_1^2 - \omega_3^2) \\ Lk(\omega_2^2 - \omega_4^2) \\ b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \end{bmatrix} \tag{3.1.7}$$

3.1.2. Kinematics

The quadcopter in this model is considered a rigid body with body fixed frame attached to it. The inertial frame is stationary with respect to the ground, with gravity acting along the negative z-axis. In contrast, the body frame is attached to the quadcopter and moves with it; in this frame, the rotor axes point in the positive z-direction, while the quadcopter's arms extend along the x and y axes. Let $(x_B,\ y_B,\ z_B)$ be the body fixed frame of reference, where x_B and y_B are parallel to the arms of the quadcopter and orthogonal to each other, whereas z_B is oriented orthogonally to the plane Ox_By_B. The inertial frame coordinate system $(x_I,\ y_I,\ z_I)$ is fixed with respect to the ground.

The position of the quadcopter in the inertial frame is $x = (x,\ y,\ z)^T$ and velocity is

$\dot{x} = (\dot{x}, \dot{y}, \dot{z})^T$. In the body frame, roll, pitch, and yaw is defined as $\theta = (\phi,\ \theta,\ \psi)^T$ and angular velocity as $\dot{\theta} = (\dot{\phi}, \dot{\theta}, \dot{\psi})^T$

The data and calculation make more sense when they are in the inertial frame. But the flight computer unit (FCU) calculates all data in the body fixed frame. So, to transform these data a rotation matrix is required. The rotation matrix is obtained by applying three successive rotations around the axes of the body-fixed frame. Quadcopter orientation relative to the ground is described using XYZ fixed Euler angles. These angles are defined as the following: the roll angle ϕ is bounded by $-\frac{\pi}{2} < \phi < \frac{\pi}{2}$, the pitch angle θ is within $-\frac{\pi}{2} < \theta < \frac{\pi}{2}$, and the yaw angle ψ ranges from $-\pi < \psi < \pi$[45].

To convert from body fixed to inertial frame, the rotation matrix is given in Equation 3.1.8,

$$\mathbf{R}_I = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \tag{3.1.8}$$

Here $C_\cdot = \cos(\cdot)$ and $S_\cdot = \sin(\cdot)$.

3.1.3. Equations of Motion

There are two types of equation to define the quadcopter dynamics. The first set of equations is due to the Principle of Action, called the Euler-Lagrange or Lagrangian formulation. The second equation which describes the rotational and translational motion of rigid bodies is called Newton-Euler equations. The second set of equations Newton-Euler is used to derive the equations. The Newton-Euler method describes the quadcopter's motion by integrating Newton's Second Law of Motion with Euler's Equations for rotational dynamics. The Newton-Euler approach allows us to model both rotational and translational movements dynamics simultaneously[46].

The forces acting on the quadcopter in the inertial frame result from thrust, gravity, and friction. The thrust forces in the inertial frame are calculated by applying the rotation matrix to the thrust in the body frame. Thus the translational motion is described as:

$$m\ddot{x} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + F_D + RT_B \tag{3.1.9}$$

Here x is quadcopter's position, g is the acceleration due to gravity, F_D is the drag force, and T_B is the body frame thrust vector.

Since, the quadcopter rotates as well, an another equation is needed to describe its angular motion, just like how Newton's second law describes linear motion. This is given by Euler's rotational equation.

$$I\dot{\omega} + \omega \times (I\omega) = \tau \tag{3.1.10}$$

where, ω is the angular velocity (how fast it is rotating). I is the inertia matrix (describes how mass is distributed). τ is the external torque. The term $\omega \times (I\omega)$ accounts for the rotational effects due to angular velocity. This can be rewritten as

$$\dot{\omega} = I^{-1}\left(\tau - \omega \times (I\omega)\right). \tag{3.1.11}$$

The inertia matrix I is symmetrical, based on above mentioned assumptions and is given by

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{3.1.12}$$

The final expression for the rotational equations of motion is

$$\dot{\omega} = \begin{bmatrix} \tau_\phi I_{xx}^{-1} \\ \tau_\theta I_{yy}^{-1} \\ \tau_\psi I_{yy}^{-1} \end{bmatrix} - \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}}\omega_y\omega_z \\ \frac{I_{zz}-I_{xx}}{I_{yy}}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}}\omega_x\omega_y \end{bmatrix} \tag{3.1.13}$$

This completes the set of equations of motion describing the dynamics of the quadcopter[43].

3.2. SLAM: Simulataneous Localization and Mapping

3.2.1. Graph SLAM

Graph SLAM, as its name implies, is a graph-based SLAM algorithm. These algorithms aim to address the entire SLAM challenge. The core of Graph SLAM is the idea that the problem can be seen as a sparse graph with nodes and constraints between them. A node representing a robot pose and any two nodes are connected by a soft spatial constraint i.e., an edge. It is termed as "soft" because it is by relaxing these constraints in a strategic/optimal way that the robot path and the environment map can be estimated. These limitations may include measurement limitations between a robot position and an environmental feature or motion limitations between two consecutive robot poses. More precisely, each link in the graph is a nonlinear quadratic constraint. The most likely map and robot path are produced by minimizing the target function of GraphSLAM, which is the total of these constraints. Another way of interpreting the algorithm is that graph-based SLAM represents robot poses and map features as nodes of an elastic net, the

stiffness of each spring being related with the degree of uncertainty of the motion and measurement models for that observation. The minimal energy state of this net can then be calculated to determine the whole SLAM solution.

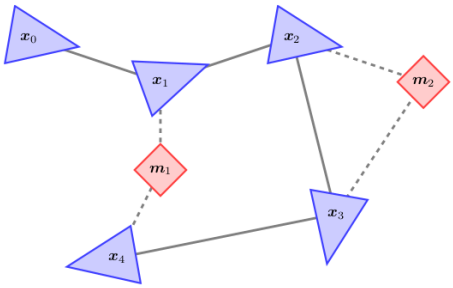


Figure 3.2.1: Graph SLAM with robot poses (blue), motion (solid),measurements (dashed), landmarks (red) [47]

3.2.2. RTAB-Map SLAM

RTAB-Map stands for Real-Time Appearance-Based Mapping as mentioned in [24]. It is a graph-based SLAM method that supports RGB-D, stereo, and LiDAR inputs, utilizing an incremental appearance-based loop closure detector. The Bag of Words approach is used to determine whether a new image corresponds to a previously visited location or a completely new one. A graph optimizer reduces the map's errors after a loop closure hypothesis is accepted and a new constraint is added to the graph. To map large-scale environments in real time, a memory management approach is used, which limits the number of images used for loop closure and graph optimization. Additionally, it can be utilized with a laser rangefinder for 3DoF mapping or alone with a portable Kinect, stereo camera, or 3D LiDAR for 6DoF mapping. RTAB-Map has C++ library and a ROS package, driven by practical shaped by practical needs such as:

- Online Processing
- Robust and low-drift Odometry
- Robust localization
- Practical map generation and exploitation
- Multi-session mapping (a.k.a. kidnapped robot problem)

The RTAB-Map algorithm needs to know the location of the sensors in relation to the main robot body, has to have access to an odometry source, be it from proprioceptive sensors or exteroceptive sensors, and access to a stream of RGB-D images or stereo im-

ages. A point cloud from a 3D LiDAR or a laser scan from a 2D LiDAR are examples of optional inputs. After being synced, these input messages are entered into the graph-SLAM algorithm. The outputs include the graph, corrected odometry, an optional 2D occupancy grid, an optional map point cloud, and map data with the most recent node added with compressed sensor data. A detailed breakdown, and practical considerations on the RTAB-Map algorithm can be consulted in Labbe and Michau's work [24].

3.2.3. Visual Inertial Odometry

Visual Inertial Odometry or VIO is the process of evaluating the position, orientation and velocity of a robot, using only an imaging unit (i.e., camera) and an IMU, attached to the body of the robot. It is an alternative to using GPS based state estimation and is useful in GPS denied or degraded environments. Cameras and IMUs are cheaper compared to LIDAR and GPS based options and are considered the more cost-efficient option. A visual inertial navigation system (VINS), like ones onboard autonomous vehicles, together with SLAM is called VIO[48]. Loop closure detection essentially determines whether a system uses VIO or VINS, whether it be localizing with a map or by using a GPS. For VIO, the camera captures 3D information from its surroundings and projects them onto a 2D image. The 2D image coordinates, u, is calculated as:

$$u = project(T_{CW}.w_p) \tag{3.2.1}$$

The IMU, acceleration and gyroscope, measures angular velocity ω and external acceleration as:

$$\omega = {}_I\omega + b_g + n_g,\ a = R_{IW}({}_Wa - {}_Wg) + b_a + n_a \tag{3.2.2}$$

where, ${}_I\omega$ is IMU's angular velocity in local frame, ${}_Wa$ is the IMU's acceleration in the global frame and ${}_Wg$ in the global frame as well. Similarly, b and n are the acceleration and the gyroscope biases and additive noises respectively. A VIO system estimates the state of the robot over at a set number of time. The biases and noises are necessary to calculate the sensor velocity and acceleration from the raw input data.

3.3. Path Planning

Path planning refers to finding an optimal path given a start and end point by avoiding obstacles if any between the points. It takes into account the temporal, physical and geometric constraints and finds the collision free path from start to end which is also optimal[49]. Various path planning techniques are described below.

3.3.1. Roadmaps

The roadmap is one of the earliest motion planning methods, where the free space is simplified into a network of one-dimensional curves or line segments [49]. This net-

work serves as a roadmap for the planner, which then searches for an optimal path using artificial intelligence-based approaches[49]. The nodes in the graph act as waypoints, guiding the quadcopter from the starting point to the goal while respecting the kinematic and dynamic constraints. Roadmap methods have been shown effectiveness in static polygonal environments but face challenges in handling narrow passages between obstacles[50].

The two prominent roadmap approaches are Visibility Graphs and Voronoi Diagrams.

1. Visibility Graph: This method connects nodes with straight-line segments, including the start, goal, and vertices of polygonal obstacles, ensuring the shortest path. However, since paths can touch obstacles, there is a risk of collision[51][52].

2. Voronoi Diagram: This diagram approach constructs a roadmap consisting of edges equidistant from obstacles, which reduces the likelihood of collision. It is not the most efficient technique for path planning, although the limitations of visibility graph is overcomed[52].

The early roadmap methods were primarily used in static environments, but later adaptations have made them more applicable for UAV path planning.

3.3.2. Cell Decomposition

The cell decomposition algorithms are classical approaches in path planning. These methods divide the obstacle-free space into discrete, non-overlapping regions called cells, enabling motion planning for robots with convex geometries[50]. Based on cell adjacency, a graph representing connectedness is constructed that allows path traversal across neighboring free cells from the starting the destination. There are two primary categories for this approach

1. Exact cell decomposition: The configuration space is exactly matched by the union of void cells.[53].

2. Approximate cell decomposition: The union of cells provides a bounded approximation of the free space[53].

Additional classifications to this method include regular grid-based and adaptive methods, where exact decompositions may utilize quadtree and octrees for 2D and 3D spaces[54]. While these methods effectively identify feasible paths, their performance depends on the cell structure. If the cells are too large or coarse, they may fail to find the shortest path efficiently. Moreover, this method struggles with real-time and dynamic

environments, and it is not suitable for 8-directional motion planning[51].

3.3.3. Potential Field

The quadcopter is modelled as a particle moving within an artificial potential field. The goal point exerts an positive force, pulling the quadcopter towards it, while obstacles generate repulsive forces, pushing the quadcopter away. The closer the quadcopter is to an obstacle or the goal, the stronger these forces become, which determines direction of movement[52].

In [55], this method was originally introduced, which was inspired by the behavior of electrical charges. It provides a simple and flexible mathematical framework, making it widely applicable for both single and multi-UAV path planning in real-time and offline scenarios.

Despite its advantages, such as fast response times and suitability for real-time control, it has a well-known limitation of getting trapped in local minima, preventing the quadcopter from reaching its goal[50]. This issue becomes of significant issue in complex environments with large search spaces, making additional strategies necessary to overcome local minima and to ensure successful navigation.

3.3.4. Sampling Based Method

Sampling based path planning is an approach that avoids fully exploring the entire configuration space. All the possible positions and orientations is called configuration space. This method is significantly more efficient than traditional grid-based methods[49]. Instead of dividing the space into grid cells and searching exhaustively, this method randomly samples points within the space and only considers a small subset of possible configurations[56].

The process starts by generating random samples, for example, to move a quadcopter, its x, y, z position as well as its roll, pitch, and yaw angles would be randomly selected. Each sampled configuration is then checked for validity (ensuring the quadcopter does not collide with anything). If a sample is valid, it is connected to nearby valid configurations, creating a network of possible paths. These connections are typically stored in a graph or tree structure, which can then be searched to find a feasible route from start to finish.

Because sampling-based methods do not need to fully explore the space, they can find solutions quickly and efficiently. However, this speed often comes at a trade-off: the solution may not always be the most optimal path. Some of the widely used planners of this technique are Probabilistic Roadmaps (PRM) and Rapidly Exploring Random

Trees (RRT), each offering different strengths for various planning scenarios.

3.3.5. Node Based Optimal Algorithm

Node based algorithms are also a kind of path planning techniques that operate on a decomposed graph representation of the environment. These methods explore a set of predefined nodes/cells where relevant information about the space such as obstacles and connectivity are stored[56]. Because the algorithm search through these nodes, they are capable of finding the shortest or most efficient path based on the given graph structure. Some of the well known algorithms are Dijkstra and A*. While these methods are effective for structured environments, they struggle in dynamic and unknown scenarios. Since the underlying graph is predefined, it must be updated whenever the environment changes. This can become difficult in dynamic environments where obstacles moves and the continuous regeneration of path becomes impractical[57].

CHAPTER 4:
METHODOLOGY

This section includes all the activities that was done from the beginning of the project. A thorough overview is shown in the flowchart as shown in Figure 4.0.1.

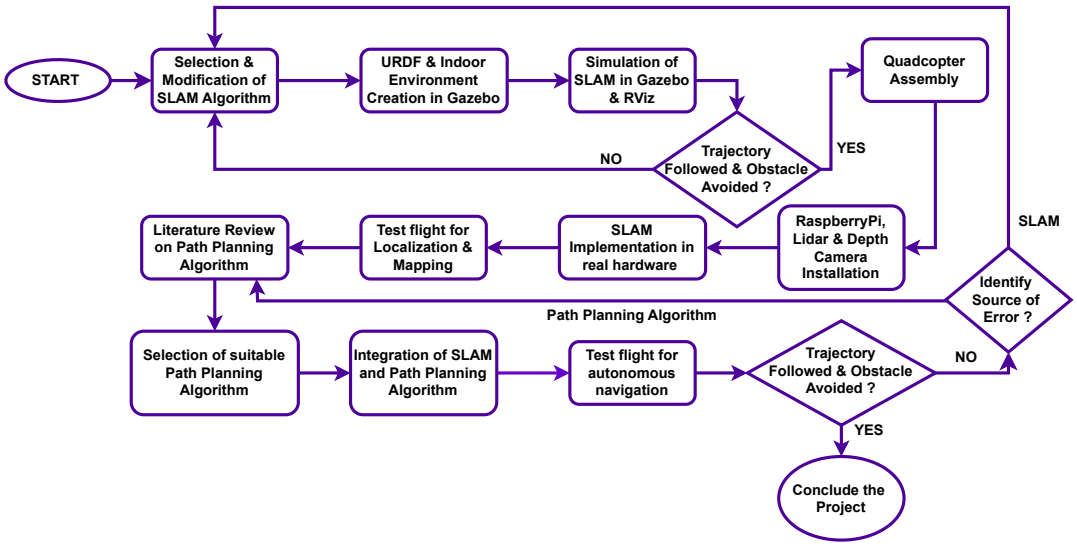


Figure 4.0.1: Overall Methodology of the project

Each block is described in detail in the following subsections:

4.1. Selection and Modification of SLAM algorithm

SLAM is a technique of mapping homogeneous environments by localizing the sensor position in the environment with centimeter accuracy. There are many SLAM algorithms developed so far for robotics applications, like Hector SLAM, Gmapping, ORB-SLAM, LSD-SLAM, RTAB-MAP, DPPTAM, etc. Among multiple SLAM algorithms only few are suitable for 3D localization and mapping for a quadcopter in an indoor cluttered environment. These algorithms either make use of an RGB-D camera or LIDAR as its sensing unit. The comparison between different SLAM algorithms is shown in Table 4.1.1. Out of all these algorithms, RTAB-Map was selected for this project, because of its feature rich map reconstruction and easy ROS implementation.

Table 4.1.1: Comparison of different SLAM algorithms

SLAM algorithm	Sensor	2D/3D	Map building density	Pose estimation
Hector SLAM	LIDAR	2D	Sparse	Poor
Gmapping	LIDAR	2D mono	Sparse	Good
ORB-SLAM	RGB-D	3D	Semidense	Good
RTAB-MAP	RGB-D	3D mono/stereo	Semidense	Good

The RTAB-Map SLAM algorithm has been selected and modified for this particular application. It has been modified to be suited for a particular image sensor (D435 depth camera), processing unit (Raspberry Pi 4B) and flight controller (Pixhawk 4).

4.1.1. RTAB-Map Algorithm

RTAB-Map SLAM algorithm is a graph-based SLAM approach which has been merged in ROS after 2013 as the rtabmap_ros package. The main ROS node known as rtabmap is shown in Figure 4.1.1. Odometry data for RTAB-Map are provided as an external input, which means that any odometry approach can be used to perform SLAM depending upon the specific application and type of robot. Graph-based map is structured with nodes and links. After synchronizing the sensor, a node is created by STM module for storing odometry pose, raw data of sensor's and extra information helpful for next modules (includes local occupancy grid used for Global Map Assembling, and visual words used for Proximity detection and Loop Closure) in the memory. The nodes are generated at a specified rate, 'Rtabmap / DetectionRate' which is set in milliseconds depending on the amount of data generated from the nodes that should overlap each other. Let us take an example of robot which is traveling fast but range of sensor is small, this rate should be increased to ensure that successive nodes data overlap, although setting to very higher could enhance computation time and usage of memory. The rigid transformation between nodes is contained by a link. Neighbor, Proximity and Loop Closure are 3 types of links. Neighbor links are connected in the STM with odometry transformation between consecutive nodes. The proximity and loop closuresure links are joined through proximity detection and loop closure detection, respectively. All links are considered as constraints for optimizing graph and graph optimization traverses the calculated error to the whole graph, when a new loop closure or proximity link is added to the graph in order to decrease odometry drift. Point cloud, 2D Occupancy Grid and OctoMap can be assembled together and transferred to the external modules using the optimized graph. In order to obtain the robot localization, odometry correction is available through tranform /map/odom.

The memory management approach of RTAB-Map runs above the graph management modules. This approach is dedicated to lower the size of the graph hence long term

online SLAM can be conducted in very large environments. With this approach, as the size of the graph increases, computation time for modules such as Graph Optimization, Global Map Assembling, Loop Closure and Proximity Detection can surpass real time constraints, that is computation time can exceed the node acquisition cycle time. The memory of RTAB-Map is divided into a Long Term Memory (LTM) and a Working Memory (WM). Once a node is shifted to LTM, that node won't be used for modules contained by WM. Once RTAB-Map update time surpasses the fixed time threshold, "Rtabmap/TimeThr", few of WM nodes are shifted to LTM to reduce the size of the WM and lower the update time. Alike fixed time limit, memory threshold, "Rtabmap/MemoryThr" also exists which can set the maximum nodes number that can be held by WM. In order to decide which nodes should transfer to LTM, important locations are identified by a weighting mechanism, mainly by heuristics such as depending upon a location that has been observed for longer duration, that location should contained in the WM. In order to achieve this, node's weight is initialized to 0 by STM while creating a new node and matches it visually with the graph's last node. If they are alike (where the % of visual words over the similarity threshold "Mem/RehearsalSimilarity"), weight of the last node is added to one to increase the weight of the new node. While the robot is stationary, weight of last node is set to zero and this node is removed in order to avoid increasing the graph size. When the limit of time or memory is reached, the oldest of lowest weighted nodes are firstly transferred to the LTM. When the loop closure occurs for a location in the WM, neighboring nodes can be transferred from LTM to WM for efficient loop closure and detections of proximity. While the robot is travelling in a recently visited zone, robot can remember the past locations to increase the current fully assembled map and perform localization using previous locations.

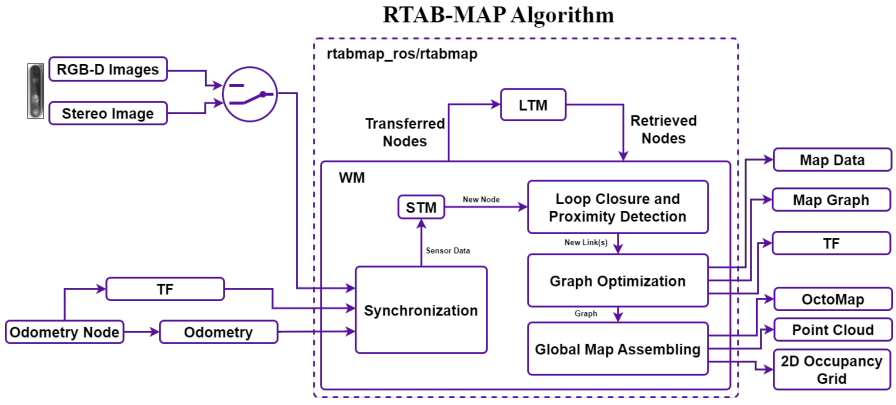


Figure 4.1.1: RTAB-Map

4.1.1.1. RTAB-Map Parameter Tuning

To increase visual odometry frequency and optimize RTAB-Map for the Raspberry Pi, specific parameters were tuned to reduce computational load and improve processing

efficiency. By adjusting these parameters, visual odometry becomes faster, and overall performance is increased for autonomous navigation. The relevant parameters are listed below.

Table 4.1.2: RTAB-Map Parameters

Parameter	**Value**	**Remarks**
Optimizer/GravitySigma	0.1	Specifies the uncertainty sigma for gravity optimization.
Vis/FeatureType	10	Determines the type of visual features to use.
Kp/DetectorStrategy	10	Strategy for keypoint detection. Affects feature extraction.
Grid/MapFrameProjection	true	Projects the map into a specific reference frame.
NormalsSegmentation	false	Enables or disables segmentation of normals.
Grid/MaxGroundHeight	1	Sets the maximum height considered as ground.
Grid/MaxObstacleHeight	1.6	Maximum allowable height for detected obstacles.
RGBD/StartAtOrigin	true	Starts the RGBD mapping process at the origin.
Vis/MaxFeatures	500	Limits the maximum number of visual features extracted.
Mem/ImagePreDecimation	2	Reduces image resolution before processing.
Mem/ImagePostDecimation	2	Decimates image resolution after processing.
Kp/DetectorStrategy	6	Alternative keypoint detection strategy.
OdomF2M/MaxSize	1000	Sets the maximum number of frames in odometry (Frame-to-Map).
Odom/ImageDecimation	2	Decimation factor applied to images in odometry processing.

4.2. URDF and Virtual Indoor Environment Creation

Unified Robot Description Format (URDF) is an XML file format for specifying the geometry and organization of robots in the ROS ecosystem. URDF of the quadcopter was created in order to test the SLAM algorithms to see how the algorithm will behave when implemented in the real world. Similarly, an indoor environment full of obstacles was created in Gazebo for testing SLAM.

URDF is essential for robot simulation, visualization, and control in the ROS ecosystem such as RViz and Gazebo. It represents robots as a tree of links (rigid bodies) and joints (connections with specific motion constraints) to define their layout and movement capabilities. It also includes detailed visual elements, such as shapes, colors, and external mesh files, for rendering in tools like RViz, while also providing simplified collision geometries for computational efficiency during motion planning. Physical properties, such as mass, center of gravity, and inertia are incorporated to enable dynamic simulations in physics simulators. Additionally, supports for sensors (e.g. cameras, lasers) and actuator interfaces to represent a robot's interactive capabilities can also be done through URDF. The extensibility through tools like xacro allows for modular and reusable descriptions, making URDF a vital component for robot modeling, simulation, and control in ROS ecosystems.

Now to create a quadcopter for simulation in ROS with Intel RealSense D435 camera the following steps were implemented:

4.2.1. Modeling of Quadcopter

The quadcopter's physical components were represented using URDF's tree structure, with links defining the rigid parts such as the quadcopter's body and arms, and joints specifying their connections. Basic geometries (e.g. cylinders, boxes) were used to describe the frame and rotors for simplicity and computational efficiency, while ensuring accurate placement and dimensions.

4.2.2. Adding the RealSense D435 Camera

The camera was modeled as an additional link attached to the quadcopter's frame through a fixed joint. Its physical properties including dimensions and weight were specified to accurately reflect the real device. The camera position and orientation relative to the base of the quadcopter were carefully set using the $< origin >$ tag to match the real world setup. The camera's field of view and frame reference were configured using custom parameters within URDF.

4.2.3. Visual and Collision Models

Separate models were created for visualization and collision. For the quadcopter's and camera's visual models, detailed mesh files (.stl or .dae) were used. Simplified geomet-

ric shapes like cylinders were defined for collision detection to optimize performance during simulation and motion planning.

4.2.4. Dynamic Properties

The inertial properties of each component, including the mass and moments of inertia ($I_{xx}, I_{xy}, I_{xz}, I_{yy}, I_{yz}, I_{zz}$), were defined to allow accurate dynamic simulations in Gazebo. This step ensured that the quadcopter's behavior under various forces, such as gravity and thrust, would be realistic.

4.2.5. Sensor Integration

The RealSense D435 camera's capabilities were described using ROS plugins for RGB-D cameras. The camera's parameters, including resolution, frame rate, and field of view, were integrated into the URDF model through a combination of tags and external configuration files.

4.2.6. Simulation and Validation

The complete URDF model was visualized in RViz to verify the structure and alignment of all components. The model was further tested in Gazebo, where the quadcopter's dynamics and camera's working were assessed in a simulated environment. Some adjustments were done to make a working model of the quadcopter .

To test the quadcopter, a virtual environment was created inside Gazebo. The simulation environment was defined using SDF in a .world file, which specifies the terrain, objects, lighting, and physics properties. The .world file typically includes a ground plane, gravity settings, and a directional light source, such as a sun model. Custom or predefined models from the Gazebo model database was added to the environment by including their URIs directly within the file. Once the .world file was created, the environment can be launched in Gazebo using a simple terminal command, or through a ROS launch file for more complex setups.

Figure 4.2.1 shows a virtual environment developed by the team for quadcopter testing.

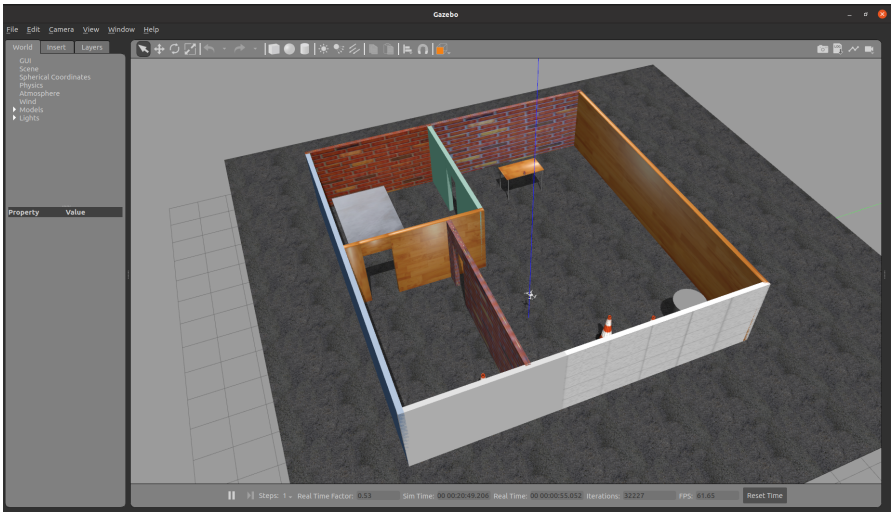


Figure 4.2.1: Virtual Environment

4.3. Simulation of SLAM in Virtual Indoor Environment

URDF of the quadcopter containing all the components like depth camera, RaspberryPi, Pixhawk 4, brushless motor, ESC, etc. was launched in ROS for simulation in Gazebo. The map was visualized using RViz.

For the simulation of the SLAM algorithm, the PX4 flight stack was selected. The PX4 flight stack provides versatile APIs for the sensors (camera and range finder) and the companion computer (Raspberry Pi). The PX4 flight stack also provides flexible Software-in-the-Loop (SITL) and Hardware-in-the-Loop (HITL) simulations. Jointly with Gazebo and the PX4 flight stack, a robust simulation environment was created. Gazebo provides a realistic physics-based environment for the quadcopter to generate real time data from the onboard sensors to input into the SLAM algorithm. The RTAB-Map SLAM algorithm then generates vision poses which subsequently are then used by the path planning algorithm to generate global and local trajectories between goal points. The entire process is facilitated by ROS, using ROS nodes and topics. The data from the topics are then visualized using RViz. A schematic representation of the mechanism is shown in Figure 4.3.1

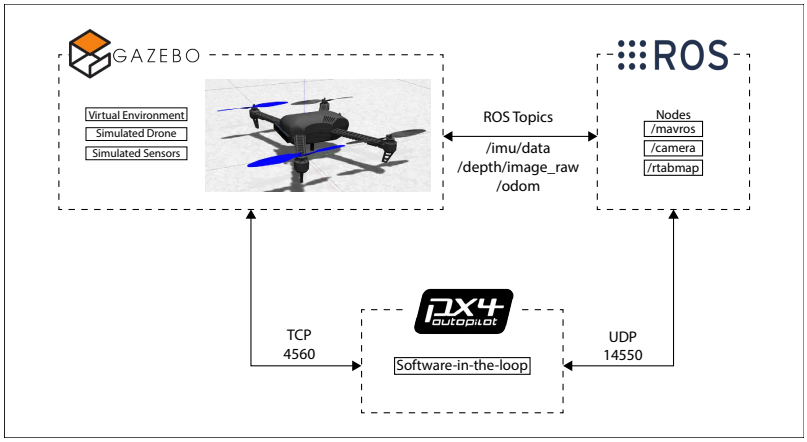


Figure 4.3.1: Simulator Schematic Representation

The PX4 SITL communicates with the mavros ROS node over UDP and the data is transferred using topics. Gazebo and the ROS environment also communicate via ROS topics and data such as images, poses and point clouds are transferred to the host ROS master. After installing all the required ROS packages correctly, the simulation can be launched by running this command in the terminal:

```
roslaunch rtabmap_drone_example slam.launch
```

The *slam.launch* file serves as a comprehensive launch file in ROS, combining all necessary components for the simulation and visualization of the SLAM algorithm. This command initializes the Gazebo simulation, starts the RTAB-Map SLAM algorithm, and opens RViz for visualization. By including these components, the launch file simplifies the process of running the entire simulation pipeline. Below shows all the tabs after launching the command above.

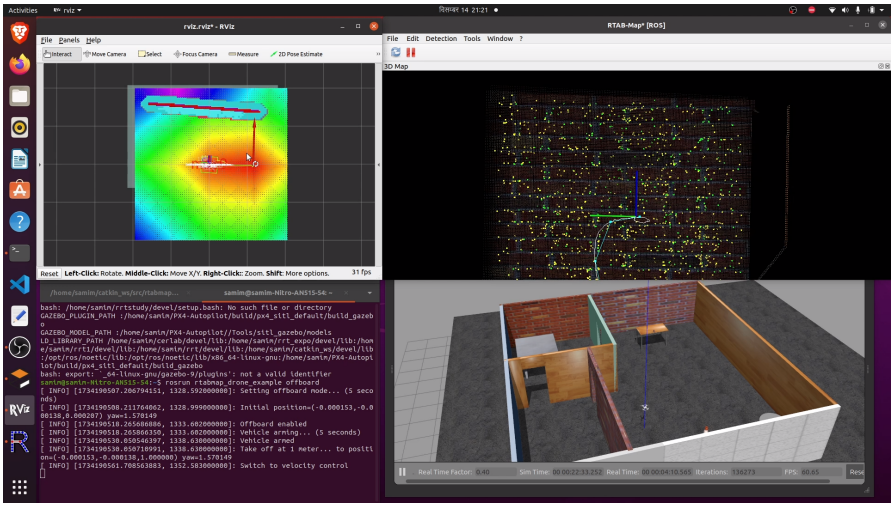


Figure 4.3.2: Simulation in Virtual Enviroment

4.4. Quadcopter Assembly

All the components like BLDC Motor, Electronic Speed Controller (ESC), Flight Controller, LiPo Battery, Propeller was assembled with a COTS (Commercial-Off-The-Shelf) quadcopter frame made of carbon fiber. The quadcopter was assembled with various other components like frame, motors and propellers, ESCs, Flight Controller (Pixhawk 4), Intel RealSense D435, VL53L1X lidar, Onboard Computer (Raspberry Pi) and battery. The carbon fiber frame provides the mounting points for all other components. Motors and propellers were attached to the arms of the frame, with each motor connected to a corresponding propeller. Electronic Speed Controllers (ESCs) were securely mounted on the frame, connected to the motors, and wired to the flight controller. The flight controller (Pixhawk 4) was centrally positioned on the frame using vibration-damping mounts to enhance stability and ensure accurate sensor readings. The Intel RealSense D435 depth camera was attached to the front of the frame to enable depth sensing, obstacle detection, and path planning. The onboard computer (Raspberry Pi) was fixed to the frame above the flight controller, handling SLAM computations and communications with between other components like the Pixhawk and the camera. The LiPo battery was securely mounted on the bottom of the frame for balanced weight distribution. The range finder sensor was attached to frame below the camera mount. The battery, which powers all onboard electronics, including the motors, ESCs, flight controller, sensors, and computer, was fixed securely in the battery compartment using a pair of Velcro straps.

The assembly process involved careful placement and wiring of each component to ensure proper alignment, weight distribution, and minimal wiring mess. Custom mounts for cameras, onboard computer, damping balls and zip ties were used where necessary to securely attach components while maintaining accessibility for maintenance. Fully assembled quadcopter can be seen in the figure below 4.4.1.

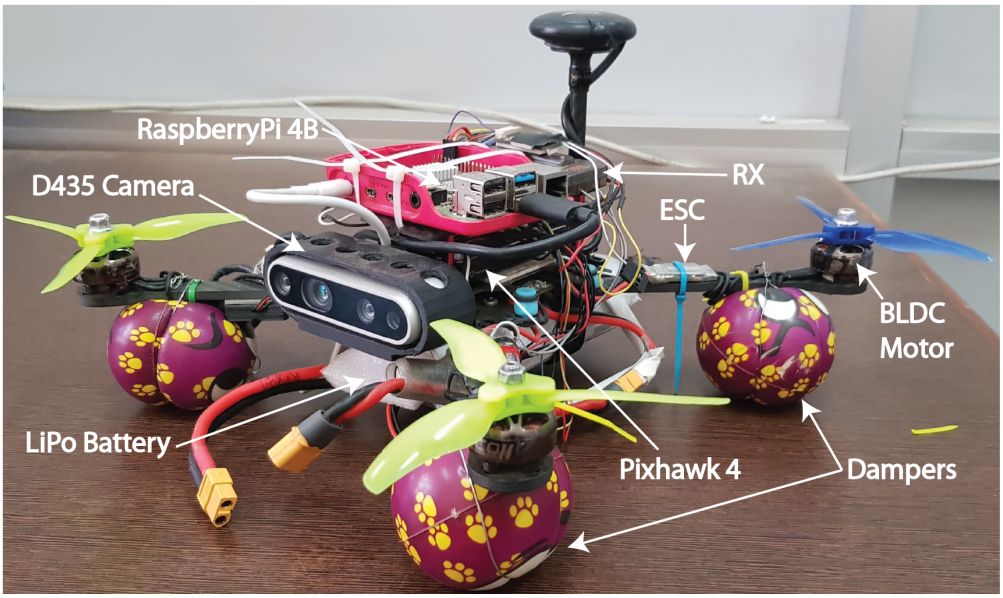


Figure 4.4.1: Assembled Quadcopter

4.5. Raspberry Pi, Lidar and Depth Camera Installation

Similarly, the Raspberry Pi microprocessor and depth camera were integrated in the frame so that quadcopter can communicate to get the control input from SLAM. To integrate the Raspberry Pi 4 with the depth camera for SLAM, it was first setup as a processing unit and then Ubuntu MATE was installed as the operating system, which provides a user-friendly Linux distribution optimized for Raspberry Pi. Next was installing the *librealsense* library, which provides the necessary drivers and tools for the Intel RealSense depth camera. Other necessary dependencies were also installed to help with the driver installation process. For ROS integration, ROS Noetic was installed, which is the recommended version for the Raspberry Pi and depth camera. $realsense-ros$ repository was cloned from Github[29], inside a workspace. Once the system was ready, the following command was typed in the terminal which connected the Realsense camera and launched the corresponding ROS driver to start the camera stream.

```
roslaunch realsense2_camera rs_aligned_depth.launch
```

After that, VL53L1X 1D lidar was connected to I2C port of pixhawk 4 flight controller. The main reason of including lidar was to maintain the constant altitude during autonomous navigation.

4.6. Sensors Calibration and Integration

Before implementing the SLAM algorithm, depth camera as well as flight controller must be calibrated to get the desired results with minimal error. Along with calibration, integration of depth camera, flight computer and an onboard computer for SLAM is another crucial part to get the desired result.

For the camera calibration, on-chip calibration was done. On chip calibration refers to the process of calibrating the sensors inside the camera to ensure that the depth and color data are aligned and accurate. The on chip calibration returns a number, which is an indication of the calibration health of the device and can be tracked over time. This self calibration feature enables re-calibration in as little as 0.6 seconds, restoring the camera's depth performance to near perfect levels in under a second. The process is incredibly simple, requiring no targets, checkerboard patterns, specific motion paths, or the need for the camera to remain stationary. All that is needed is to call an on-chip function on Realsense SDK for re-calibration. The figure below shows depth map during calibration captured through *realsense − viewer*.

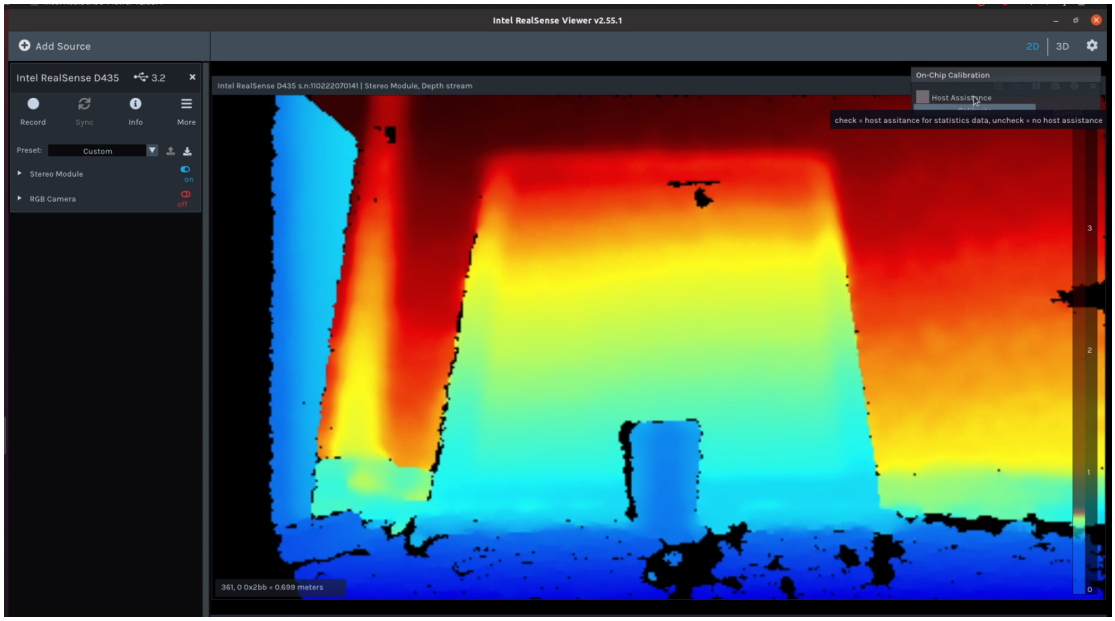


Figure 4.6.1: On Chip Calibration

For the flight computer calibration, the quadcopter was rotated 360° for compass calibration, held still for accelerometer calibration, maintained in flat, level ground for gyroscope and level horizon calibration. There are 6 different orientations for compass and accelerometer calibrations. This calibration process can be performed through the ground control software, QGroundControl, where the system guides users through each step and streamlines the entire calibration process.

Table 4.6.1: PX4 VIO Parameters

Parameters	Values
EKF2_HGT_REF	Vision
EKF2_EV_DELAY	200
EKF2_EVA_NOISE	2.86 deg
EKF2_GPS_CHECK	0
LPE_FUSION	132
ATT_EXT_HDG_M	1
MAV_1_CONFIG	TELEM 2
SER_TEL2_BAUD	921600
MAV_1_MODE	External Vision

The final step is the integration of the depth camera, Pixhawk 4 and Raspberry Pi. The flight computer unit and the companion computer communicate with each other via telemetry port 2 on the Pixhawk, using UDP at a data rate of 921600 bits per second. Depth camera and Raspberry Pi are connected through a USB 3.0 port for faster data transfer rates. Other parameters that are necessary for the flight computer to operate properly is shown in Table 4.6.1.

$EKF2\_HGT\_REF$ is a PX4 parameter which is set to Vision based as the quadcopter uses visual inertial navigation system for localization. Similarly, $EKF2\_EV\_DELAY$, $EKF2\_EVA\_NOISE$ and $EKF2\_GPS\_CHECK$ are all EKF2 parameters for the PX4 firmware onboard the Pixhawk, needed to set vision pose estimation delay, measurement noise for vision angles and define threshold values for the GPS on the flight controller board, respectively. The LPE_FUSION parameter is set to 132 to fuse optical flow, vision position and landing targets for the Pixhawk, while the $ATT\_EXT\_HDG\_M$ parameter is set to 1 for external vision based heading estimation.

It is important to set these parameters to these specific values, to ensure smooth functioning and proper localization, mapping and navigation. Finally, the integration of all the above components is shown in figure below:

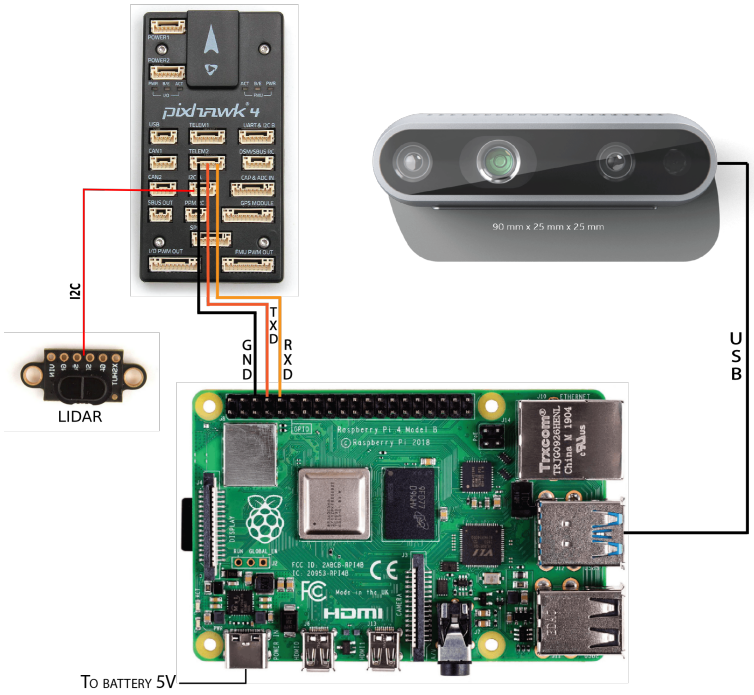


Figure 4.6.2: Interface Diagram

4.7. SLAM Implementation in Actual Hardware

The actual hardware implementation share similarities with the SITL setup, with a few key changes. For flight tests, the processes are divided into three interconnected boards. The quadcopter is equipped with two units, a companion computer and a flight control unit. The flight control unit consists of a Pixhawk 4 and the companion computer is a Raspberry Pi 4B. The Pixhawk controls the actuators i.e., the BLDC motors and ESCs while the companion computer connects to the ground control station via Wi-Fi. The ground control station is an Ubuntu 20.04 laptop with ROS Noetic installed, communicating with the onboard companion computer via SSH. The architecture for this implementation is shown in Figure 4.7.1

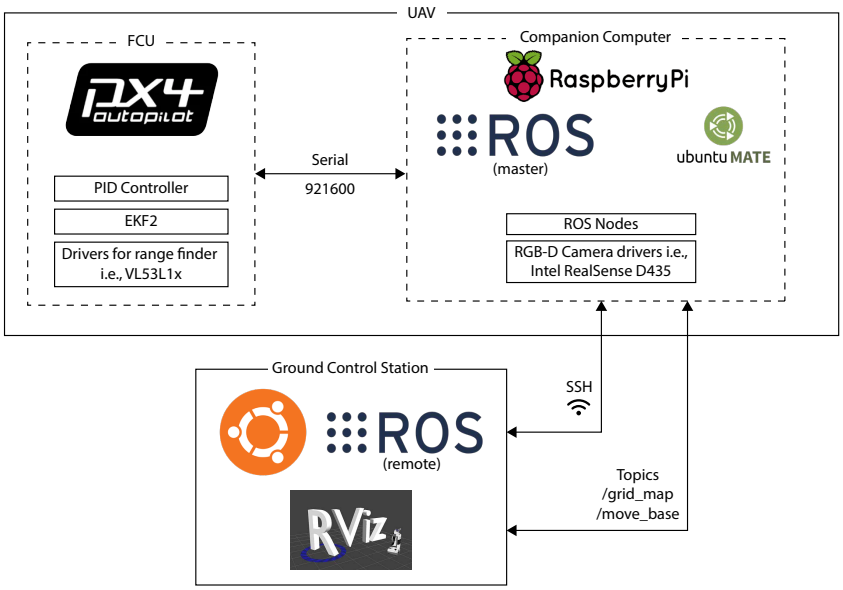


Figure 4.7.1: Hardware architecture for flight tests

The companion computer is fitted on board the quadcopter with Ubuntu 20.04 and ROS Noetic installed. It communicates with the flight controller Pixhawk via a serial line and data is transferred back and forth via ROS topics. The on board computer, or the Raspberry Pi starts various nodes that subscribe to the said topics for state estimation and trajectory generation. Over Wi-Fi, the ground control computer connects via SSH to the on-board computer and is used for visualization during flights.

The first step to implement the SLAM algorithm on the companion computer was to obtain data from the depth camera and the flight computer IMU into ROS. The depth camera provided visual data, including depth maps, RGB images, and point clouds which were published to specific ROS topics (like $/camera/depth/image\_raw$ and $/camera/color/image\_raw$). These topics were subscribed by various other ROS nodes, including the SLAM algorithm. Similarly, the IMU on the Pixhawk provided odometry data, such as acceleration and orientation (roll, pitch, yaw), via topics like $/mavros/imu/data$. These sensor data were then inputted into ROS, where they were processed for state estimation.

For RTAB-Map proper topic remapping was required. It expected specific topics for depth and RGB images, so if the camera published data on different topics, they had to be remapped to ensure RTAB-Map received the correct data for mapping. This was done within the ROS launch files. Additionally, it also relied on data from the camera and IMU for accurate localization and map building. The flight computer IMU data helped to correct and refine the SLAM process by providing information on the camera's and

quadcopter's position and orientation.

Once the necessary data was properly remapped and synchronized, RTAB-Map was launched on the Raspberry Pi. The algorithm then generated a 3D map of the environment, tracked the camera's movement, and updated the map as the robot moved. Although the mapping and localization process took place on the Raspberry Pi, visualization was done on a ground control laptop. By setting up proper ROS networking, the data processed by the Raspberry Pi was transmitted to a laptop, where RViz used for real-time visualization of the map. This setup allowed for efficient onboard processing and detailed remote monitoring, enabling effective SLAM operation on the Raspberry Pi for autonomous navigation and mapping.

4.8. Test Flight for Localization and Mapping

An indoor cluttered environment with lots of obstacles was created inside a room where GPS navigation is almost impossible. The quadcopter was deployed into the said environment with the SLAM algorithm activated so that it can localize itself and build a map of the environment simultaneously.

Before conducting an autonomous flight with the quadcopter in a cluttered indoor environment, initial tests were performed by moving the quadcopter manually by hand. This method allowed to test the algorithm without risking any damage to the quadcopter or the environment. By manually moving the quadcopter, it could be observed how well the RTAB-Map algorithm was able to track the quadcopter's movement, build the map, and localize the quadcopter in real-time. This initial testing phase also ensured that the depth and IMU data were being processed correctly and that all topics, such as the depth, RGB and infrared image streams from the camera and the IMU data, were mapped properly within ROS.

The reason for conducting these tests manually by hand before testing the quadcopter autonomously indoors was primarily for safety reasons. It also allowed us to ensure that the SLAM algorithm was functioning as expected, such as whether or not it was correctly detecting obstacles, updating the map, and localizing itself inside the map. Only after confirming that the integration and map building processes were performing as expected, the team considered moving on to more advanced testing, such as autonomous indoor flights with obstacles.

4.9. Selection of Suitable Path Planning Algorithm

There are number of path planning algorithms which are developed for aerial and ground robots for autonomous navigation with the shortest path. A proper path planning algorithm is necessary to be implemented with SLAM because on its own SLAM cannot navigate in a cluttered environment with lots of obstacles. There are multiple path planning algorithm such as A*, RRT, Dijkstra, Ant Colony Optimization Algorithm, etc. Among those algorithms, A* and RPP algorithm was selected as global and local planner respectively for navigation, by determining the shortest path between starting point and destination.

4.10. Path Planning Algorithm Implementation

An implementation of a popular path planning algorithm, the A* algorithm was done, making it compatible with the ROS ecosystem, so that the quadcopter could navigate autonomously. Autonomous vehicles base their navigation decisions on planner modules that create collision-free waypoints on the path to reach the destination point. These modules are capable of finding the optimal solution minimizing the computation time and distance covered by the vehicle avoiding static and dynamic obstacles. Some research groups test these modules on real vehicles such as PROUD in Parma in 2013 or Karlsruhe Institute of Technology (KIT) driving around the cities with advanced technology in perception, navigation, and decision making. In all cases, the navigation module is divided into a global planner and a local planner, where the former finds the optimal path with a prior knowledge of the environment and static obstacles, and the latter recalculates the path to avoid dynamic obstacles.

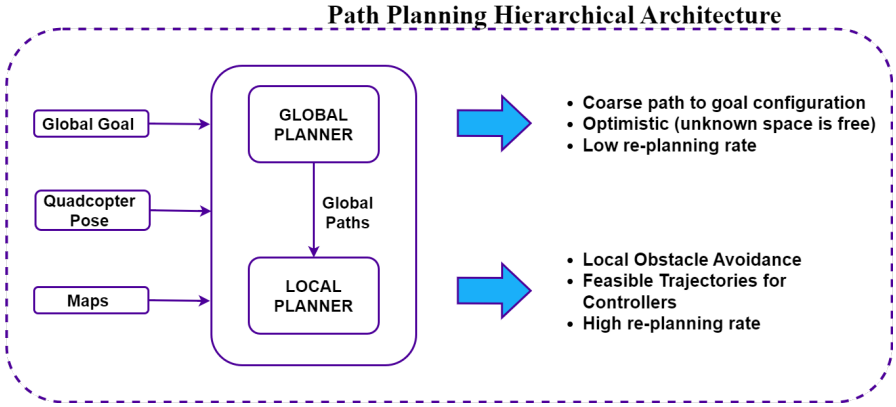


Figure 4.10.1: Path Planning Hierarchical Architecture

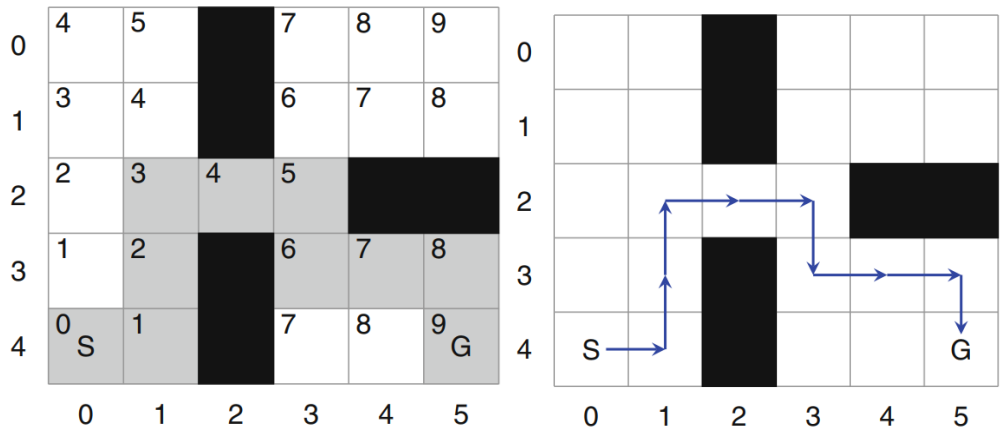
4.10.1. Global Planner

As mentioned before, the global planner requires a map of the environment to calculate the best route to the goal. Depending on the analysis of the map, some methods are based on Road maps like Silhouette, proposed by Canny in 1987 and Voronoi proposed in 2007[39]. Some of them solve the problem by assigning a value to each region of the

roadmap, or nodes, in order to find the path with the minimum cost. Some examples are the Dijkstra algorithm, Best First, and A*. Another approach is by dividing the map into small regions (cells) called cell decomposition. A similar approach by using potential fields, the most extended algorithm used few years ago rapidly exploring random trees or the new approach based on neural networks. Some solutions combine the aforementioned algorithms improving the outcome at the cost of high computational power.

4.10.1.1. Dijkstra's Algorithm

Dijkstra's algorithm applies the roadmap approach by transforming the problem into a graph search method, utilizing data from a grid cell map as illustrated in the figure 4.10.2. This approach begins with a list of potential nodes that the vehicle can drive through (free space), and then gives each one a cost value. Starting from the initial point, this value is incremented by the number of nodes that must be traversed to reach each subsequent node. For instance, the open area surrounding the beginning point in Figure 4.10.2 is worth one unit, the second generation of neighbors is worth two units, and so on. For every cell with a value, a corresponding checked cell is marked, and the process proceeds to the next set of neighboring cells until the target point is reached. The shortest path is the lowest value of the sum of all nodes from the beginning and the ending points. Once the global path from the start to the goal is successfully determined, all the chosen nodes are converted into positions relative to the reference coordinate axes. For every free cell, the global planner splits the map into nodes; however, the result is not smooth, and certain points do not match the kinematics and geometry of the vehicle.



(a) Grid Cell Map (b) Optimal Path

Figure 4.10.2: Dijkstra Algorithm Illustration

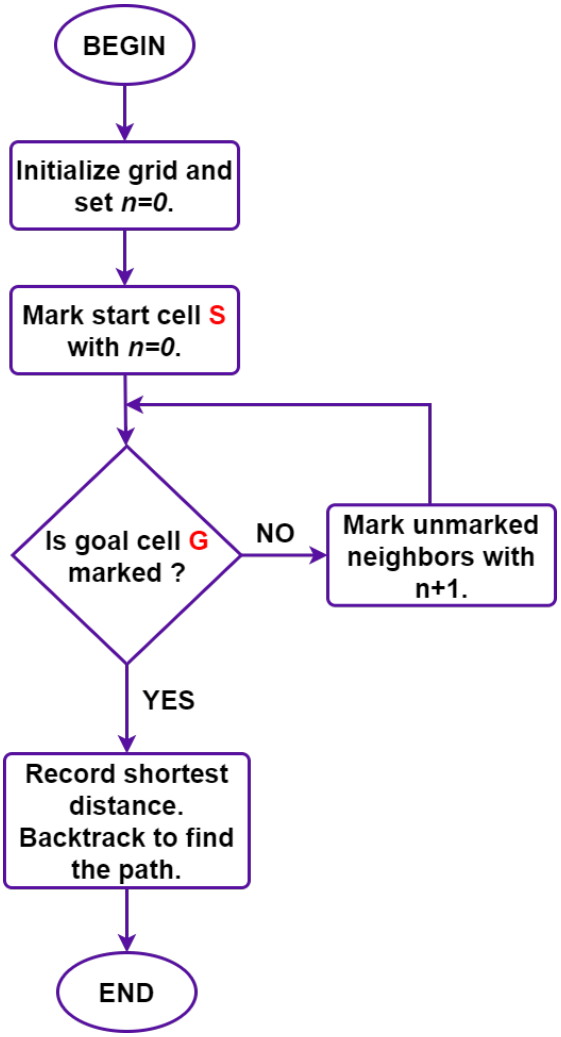


Figure 4.10.3: Dijkstra Algorithm Flowchart

Using Dijkstra's algorithm the shortest path between two navigation points or "nav goals" is calculated. The 2D nav goals, which can be provided via the visualizer RViz, are also used to orient the quadcopter in a particular direction. The algorithm is provided in the ROS ecosystem through the 'navfn' node. The navfn node is made available through a package sharing the same name which provides the navigation function implementation of Dijkstra's algorithm in ROS[58]. It is used to call the global planner to create a global path for the quadcopter with the move_base node.

4.10.1.2. A* Algorithm

Firstly, A* algorithm was developed in 1968 by Hart, Nilsson and Raphael. A* algorithm is an best-first search or an informed search type algorithm, which means it is developed based on the weighted graphs. It works by starting from a specific start node of a graph and directs to find a shortest path with least time to reach the goal node which will have the least cost as shown in Figure 4.10.4. It searches the shortest

path by generating multiple paths which originates at the start node and extends those paths only single edge at an instant till the goal node is reached. It can considered as an advancement on the Dijkstra's algorithm, because A* algorithm includes a heuristic function in it. Hence, this algorithm provides a priority considering the distance to the goal node and also the path length. At a lower level, this algorithm works very similar to the Dijkstra's algorithm, which explores all the nodes between start and goal point. A* algorithm is considered as the mostly implemented path planning algorithm in aerial and mobile robotics to solve graph traversal and path finding problems, due to it's accuracy, being simple and more effective making the use of heuristic function.

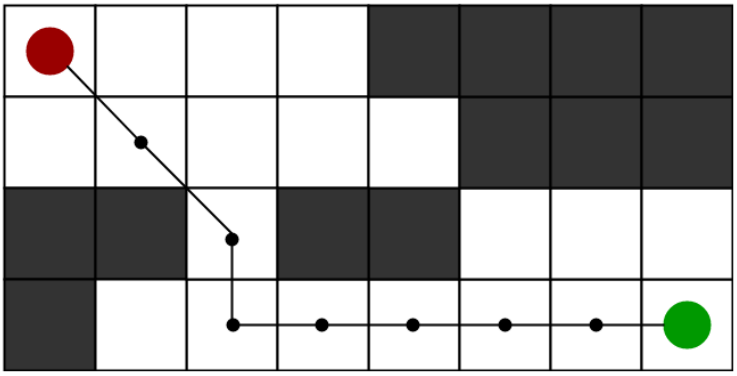


Figure 4.10.4: Path generated by A*

During each and every iteration of the main loop, this algorithm should determine the paths that must be extended. It conduct this task which depends on the cost of that path and an estimated cost desired to reach the goal node. The path that minimizes f(n) is selected by A* .

$$f(n) = g(n) + h(n) \tag{4.10.1}$$

In this expression, n denotes the upcoming node on the path, g(n) denotes the cost of the path from the start node to n, and h(n) is a heuristic function that determines the cost of the cheapest path from n to the goal node. The heuristic function depends upon the problem. Heuristic function is admissible which means it never overestimates the true cost to get to the goal node hence A* is fully guaranteed to return a least-cost path from start node to goal node. Heuristic function h(n) supplies an estimated cost from the current node to the goal node, acting as the algorithm's "informed guess" about the remaining path. Mathematically, for any given node n, the heuristic estimate must satisfy the condition $h(n) \leq h^*(n)$, where $h^*(n)$ is the actual cost to the goal, making it admissible by never overestimating the true cost. In grid-based or map-like problems, common heuristic functions include the Manhattan distance and Euclidean distance. For coordinates (x1, y1) of the current node and (x2, y2) of the goal node, these distances are calculated as:

Manhattan distance :

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

Euclidean distance :

$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

A* algorithm doesn't guarantee optimality of the generated path however it always provides a valid path whenever possible. The provided path depends on number of parameters. The major factor is the resolution of the map, where higher resolution corresponds to the large number of vertices in the generated map which leads to better and optimum path using lot of computational power. However, lower resolution corresponds to the quicker but non optimal path which increases the energy consumed by the quadcopter. Resolution is a major factor because of the increase in number of vertices of map.

A* algorithm was selected as the suitable algorithm to be used as global planner for this project because of the computation time and processor load which was lower than that of Dijkstra's algorithm. The overall workflow of A* algorithm is shown in the flowchart in Figure 4.10.5.

4.10.2. Local Planner

The other half of path planning is local path planning. In local path planning, the robot is either somewhat aware or completely unaware of its surroundings [59]. Thus the local path planning also focus on employing sensors to detect environments and sensor fusion to provide stable and reliable output from said sensors. Local path planning on its own is able to generate locally optimal paths, however global paths that ensure the target being reached is not guaranteed. This calls for an integration of global and local path planning to generate a "globally optimal path" and a local path to avoid obstacles. Using the onboard detecting sensors, the local planner generates "waypoints", accounting vehicle restrictions and the detected dynamic obstacles. Since, considering the entire map of the environment is not feasible due to the computational costs and inherent sensor constraints, a local planner recalculates the path at a specified rate, reducing the map of the environment to the vehicle's immediate surroundings and updating as the vehicle moves [39]. With the updated local map in addition with the global waypoints, the local planner generates a path that tries to closely replicate the global waypoints, while avoiding the obstacles en-route. Different approaches for the trajectory generation like Clothoid lines, Bezier lines and splines can be used.

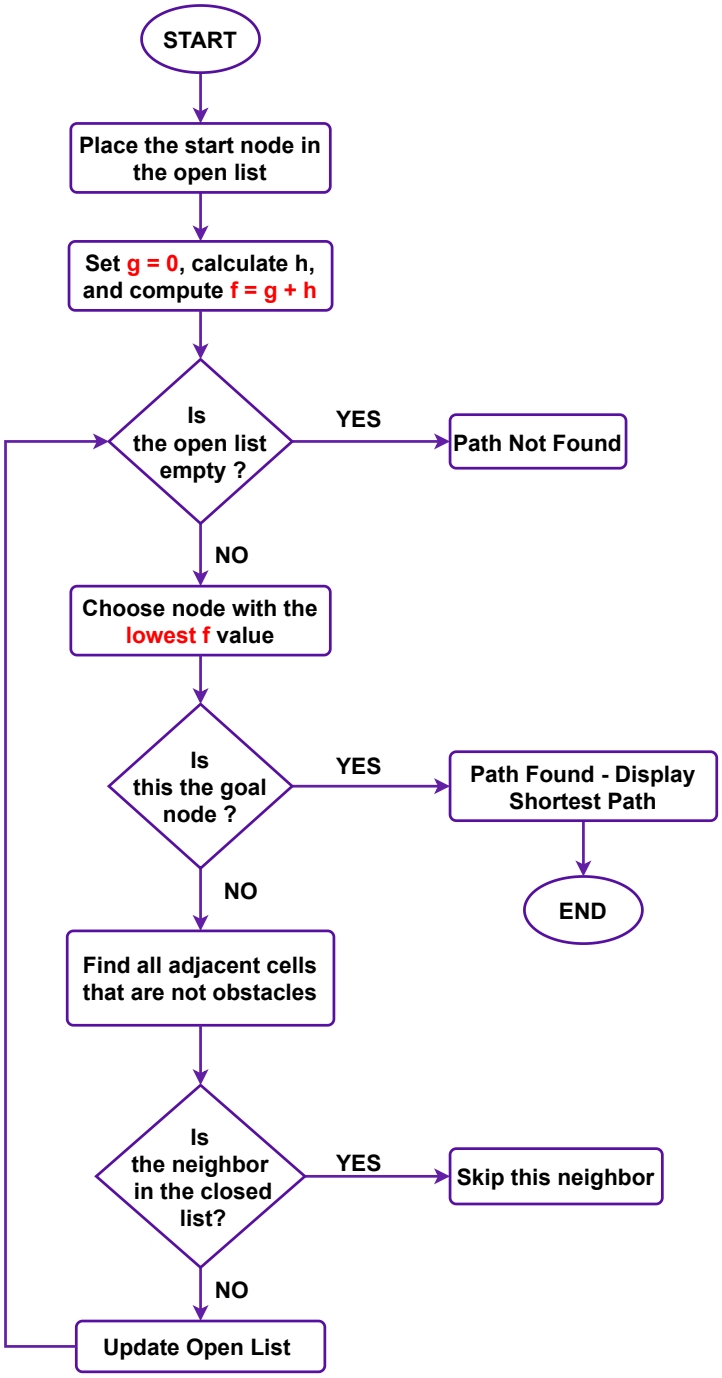


Figure 4.10.5: A* Algorithm

4.10.2.1. Dynamic Window Approach (DWA)

The dynamic window approach controls the robot by selecting speeds that are physically achievable and safe from obstacles. Be creating a “dynamic window” of such possible speeds, it selects the best velocity by maximizing an objective function that continuously updates the robot’s movement commands [40]. The DWA is provided as

a local planner in the *dwa_local_planner* package in ROS [60]. The package supports any robot with circular or a polygonal footprint. Given a global plan and a costmap to follow, the planner produces velocity commands for the mobile base to move the robot in a plane. Around the robot, the planner creates value functions represented by grid maps that translate the costs of traversing the grid cells. Using the value functions the planner sends the movement commands by determining the dx, dy, and dθ velocities. A basic instance of a DWA algorithm is as follows:

- Sample the robot's control space for possible velocities (dx, dy, $d\theta$).
- Forward simulate from the robot's current state for a short period of time to predict the outcome using the sampled velocities.
- Assign a score to each trajectory using a metric that considers key characteristics like goal proximity, obstacle proximity, global path alignment and speed. Trajectories that result in collision with obstacles should be discarded.
- Select the best scoring trajectory and send the associated veloctiy to the robot's movement commands.
- Reiterate the steps.

4.10.2.2. Regulated Pure Pursuit (RPP)

Regulated Pure Pursuit is a variation of the Pure Pursuit controller that targets industrial robotic needs i.e., "constrained and partially observable environments" [42]. The controller provides linear regulation cost functions that adapt the robot's translational velocity to certain environmental conditions like sharp turns or congested areas. The controller modulates forward speeds based on path curvature, significantly reducing the risk of overshooting when navigating blind corners at high velocities, thereby enhancing operational safety. This demonstrates superior path adherence compared to conventional Pure Pursuit implementations. It incorporates intelligent deceleration protocols when detecting nearby obstacles, automatically reducing speed to prevent potential collisions. The system also features dynamic adjustment of the lookahead distance based on current velocity, enabling consistent performance across diverse speed ranges.

The RPP controller incorporates proactive collision detection mechanisms. The maximum time window for potential collision prediction can be configured based on current movement commands. The system projects the robot's trajectory using present linear

and angular velocities for this specified duration to identify possible collision scenarios. While checking for collisions between the robot and the distant target point might seem intuitive, a time-based approach proves more practical in confined environments. This allows the system some flexibility for maneuvering in tight spaces. For example, when moving at 0.1 m/s, examining 10 meters ahead (equivalent to 100 seconds of travel) would be impractical. The time-based method scales collision detection appropriately—looking further ahead at higher speeds and focusing on immediate surroundings during slow, precise movements near obstacles, preventing false collision alerts during valid maneuvers. If the maximum permitted viewing distance is set high enough, collision detection extends all the way to the lookahead point without exceeding it. For computational efficiency, the global path undergoes continuous pruning to retain only the closest point to the robot, which can be seen in Figure 4.10.6, eliminating unnecessary processing of redundant path elements. The portion of the path within local costmap boundaries is then transformed to the robot's reference frame, and a target point is established using a predetermined distance. The controller deliberately reduces speed when approaching sharp turns into partially visible dynamic settings (such as corridors, aisles, and intersections), minimizing collision probability and potential impact. It also decreases linear velocity when near obstacles—including people and fixed structures—to enhance safety in limited indoor spaces. Unlike other variations, this system incorporates anticipatory collision detection capabilities. The RPP controller is also available in ROS as part of the Nav2 package [61].

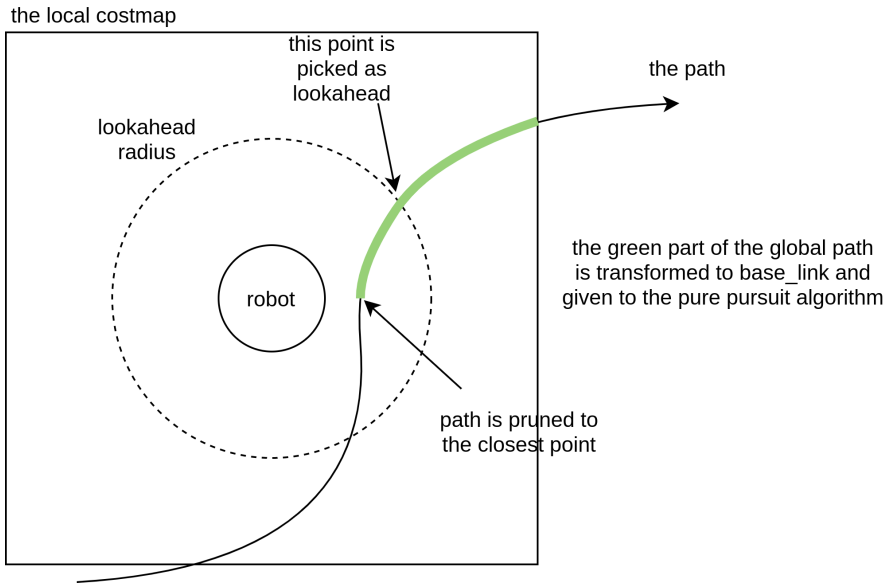


Figure 4.10.6: Regulated Pure Pursuit Local Path[61]

RPP was selected as the suitable algorithm to be used as local planner for this project because RPP creates much smoother paths locally than DWA was able to, hence the

quadcopter was able to navigate smoothly without any drifts and jerks in altitude. Different parameters of RPP are listed in Table 4.10.1.

4.11. Integration of SLAM and Path Planning Algorithm

After the path planning algorithm was selected and configured, it was combined with the SLAM algorithm so that the map developed by SLAM can be used for autonomous navigation, finding the shortest and most feasible path for the quadcopter.

In the SLAM launch files, a move_base node is initialized to move the quadcopter along the path generated by the local planner, which closely follows the global path/plan. For that, certain parameters and arguments need to be passed to the node. The grid map and odometry obtained from the SLAM algorithm are both passed onto the node along with global and local costmap parameters. From the generated odometry, the global frame is identified and further in the parameter file, the quadcopter kinematic parameters are defined. These include maximum and minimum velocities, accelerations, and

Table 4.10.1: RPP Planner Parameters

Parameter	**Value**
Convert Offset	0.0
Goal Distance Tolerance	0.4
Rotate Tolerance	0.9
Max Linear Velocity (v)	0.1
Min Linear Velocity (v)	0.01
Max Linear Velocity Increment	0.1
Max Angular Velocity (w)	0.1
Min Angular Velocity (w)	0.01
Max Angular Velocity Increment	0.1
Base Frame	base_link
Map Frame	map
Lookahead Time	0.2
Min Lookahead Distance	0.3
Max Lookahead Distance	0.4
Regulated Min Radius	0.01
Inflation Cost Factor	2.0
Scaling Distance	0.2
Scaling Gain	1.0
Approach Distance	0.1
Approach Min Velocity	0.0

rotational velocities in the X and Y directions. Also defined in the file are trajectory and simulation parameters. These help in calculating the cost function to effectively weigh each generated trajectory and the sampling constraints while forward simulating trajectories. From the costmap generated by RTAB-Map, the obstacles are identified in the XY plane. Based on those obstacle data, the global and local planner work in conjunction to generate the shortest path with the least obstruction.

The move_base node is an implementation of a ROS action, whose goal is to move the vehicle, quadcopter for this application, to a given goal in the world. It links both the global and local planner to achieve its navigation task[62]. Based on tolerances and parameters set on the local and global parameter files, the move_base moves the quadcopter through a path while avoiding the obstacles in the local environment.

4.12. Flight Testing for Autonomous Navigation

Once SLAM and the path planning algorithm were integrated on the onboard computer, the quadcopter was deployed in a custom, cluttered indoor environment where localization, mapping and autonomous navigation was done through the onboard computer.

The test flight procedures were conducted in three major steps:

4.12.1. Environment Creation

A safe environment of 6m x 6m, with artificial obstacles as shown in Figure 4.12.1 was created at Incubation, Innovation, and Entrepreneurship Center (IIEC) Pulchowk Campus, to perform test flights. Safety was of utmost importance while performing the test flights, to both the quadcopter and the people involved during flight tests. Thus, a netted enclosure with soft cushioning material on the ground was made, with foam boards acting as makeshift walls. Cardboard boxes and spare foam parts acted as obstacles. Posters were hung on the walls to make sure the custom made cluttered environment was feature rich and proper odometry would generate for localization.

4.12.2. Offboard Autonomous Launch

The autonomous offboard launch code[1] retrieves the current X and Y position of the SLAM output, which is then fed to the Pixhawk flight computer for navigation. The goal X and Y coordinates is selected and extracted from the RViz GUI.

The code first switches the flight computer to OFFBOARD mode, a control mode that allows the vehicle to be guided using position, velocity, acceleration, attitude, attitude rates, or thrust/torque set points. After OFFBOARD mode, the code changes to POSITION mode to take off and hover at the set altitude based on the LiDAR data.

[1]The offboard launch code is attached in the Appendix section of the report.

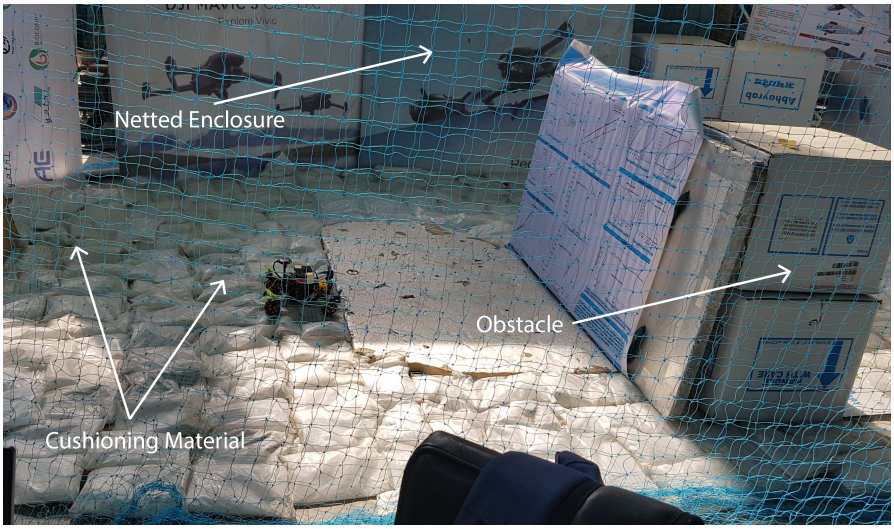


Figure 4.12.1: Feature rich, cluttered safe testing environment at IIEC Pulchowk

Once the quadcopter reaches and hovers at the set altitude, VELOCITY2D control mode is used to guide/navigate the quadcopter towards the target X and Y coordinates extracted from the RViz. The entire code relies on MAVROS vision pose commands to execute the autonomous flight.

4.12.3. Hover Tests

Once the safe environment for testing was created, the quadcopter was then deployed into the environment. The quadcopter hovered at a certain altitude ($\sim 0.5m$) as provided by the users as shown in Figure 4.12.2. This was necessary to ensure that the SLAM algorithm and the path planning algorithm were working as desired. This also served as a sanity check to ensure that the data from the IMU and the SLAM algorithm was being fused correctly, and proper odometry was being generated in the environment.

Figure 4.12.2: Quadcopter hover tests inside the netted enclosure

4.12.4. Test Flights

Once the hover tests cleared, the quadcopter was then deployed into the environment for autonomous navigation. The quadcopter mapped the obstacles in the area, while navigating in between obstacles, avoiding collision. The test flight process was a reiter-

ative process, and the data obtained from each flight test was used to study and improve upon the next test. Several tests with different conditions, with different obstacles and different environments were conducted to evaluate the system's adaptability over a wide range of scenarios and to ensure the robustness of this approach in real-world scenarios.

CHAPTER 5:
RESULTS AND DISCUSSION

5.1. Output

Results and discussion section is divided into three subsections i.e. for simulated, indoor and real world environment. The project was tested and validated in three different kinds of environments for implementing SLAM and autonomous navigation of a quadcopter.

5.1.0.1. Simulated Environment

Initial simulations demonstrated mapping and localization capabilities in two virtual Gazebo worlds. The 3D point clouds and the trajectories followed by the quadcopter, obtained through SLAM, were visualized using RTAB-Map GUI. The 3D map in the .db file format is converted into a .ply file for plotting with MATLAB. Figures 5.1.2, 5.1.3, 5.1.5 and 5.1.6 show 3D and 2D plots for both of the simulations room. Additionally, the move_base ROS framework was utilized in conjunction with RTAB-Map to navigate autonomously. This framework integrated global and local path planning algorithms, where the global planner implemented A* algorithm and the local planner employed RPP for efficient obstacle avoidance. The ROS framework also allows sending 2D navigation goals through RViz, enabling 2D goal-based simulations within the environment. The 2D grid and 3D point cloud maps for environment 1 is shown in Figure 5.1.3 and 5.1.2 respectively. Similarly, for environment 2 Figures 5.1.6 and 5.1.5 shows the 2D grid and 3D point cloud map. The 2D grid shows obstacles with black pixels and free spaces with white. The trajectory the quadcopter follows is shown in the 2D grid map. The 3D point cloud map is the 3D reconstruction of the environment generated using the SLAM algorithm.

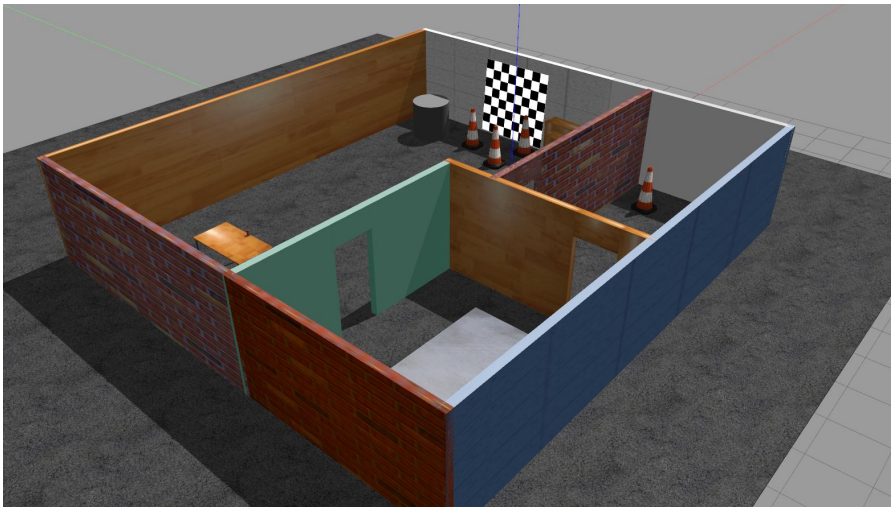


Figure 5.1.1: Environment 1 in Gazebo

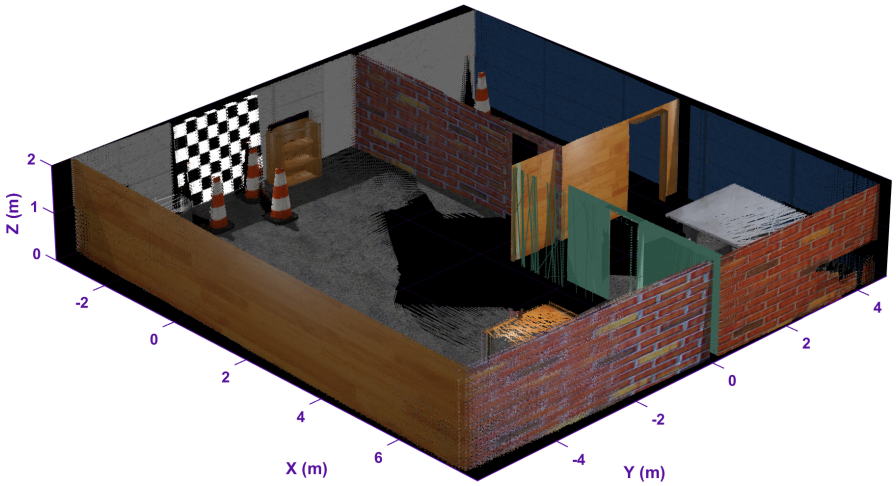


Figure 5.1.2: 3D Point Cloud Map of Environment 1

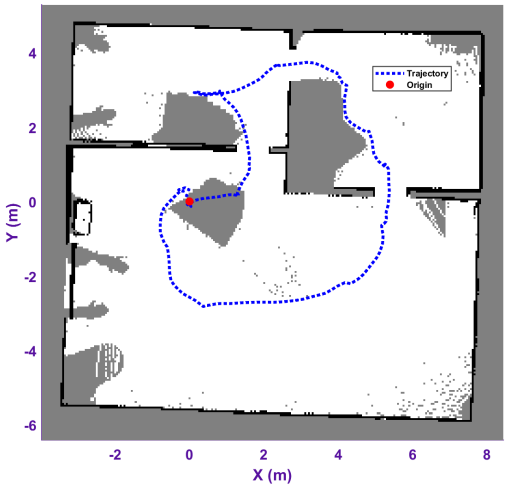


Figure 5.1.3: 2D Grid Map of Environment 1

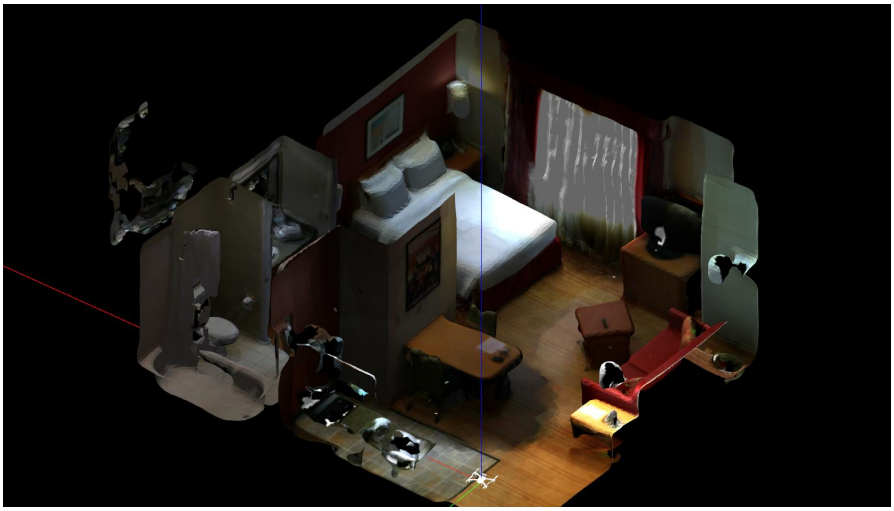


Figure 5.1.4: Environment 2 in Gazebo

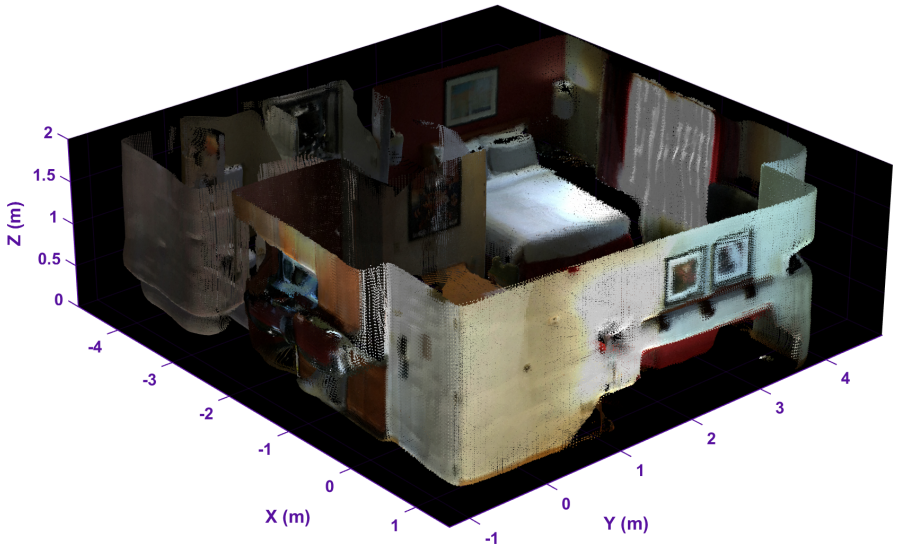


Figure 5.1.5: 3D Point Cloud Map of Environment 2

5.1.0.2. Indoor Environment

A manual handheld mapping experiment was also conducted within the department's Manufacturing Laboratory. The Figure 5.1.7 below shows the 3D point cloud map plotted in MATLAB. This was done to ensure successful integration and working of all components related to SLAM. Further, autonomous flight of quadcopter was tested in an indoor environment created in the IIEC, with dimensions of 35 × 25 feet and nets installed on the sides for safety. An image of the environment is shown below in Figure 5.1.8. In this controlled setting, posters were placed in front of the camera to provide additional visual cues. The quadcopter demonstrated stable hovering at the desired altitude using fused SLAM data from depth camera inputs and IMU data from the Pixhawk. Instead of RGB images, which uses lots of processing power of the Rasberry Pi, infrared data from the depth camera was utilized for efficiency. The infrared camera

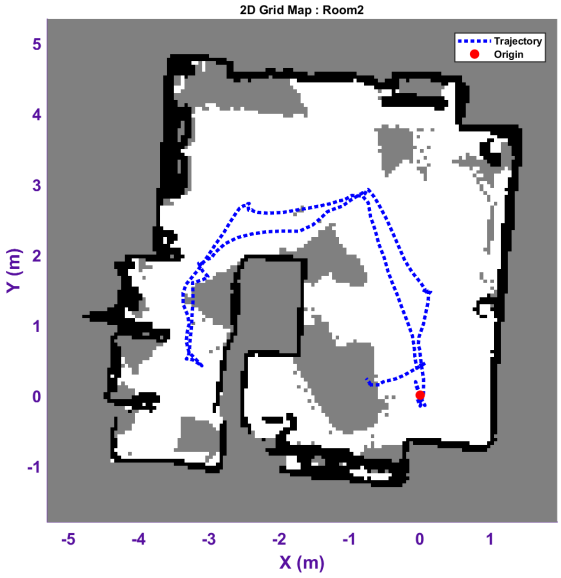


Figure 5.1.6: 2D Grid Map of Environment 2

was used because it has a good global shutter and has larger field of view, two important things to get best visual odometry.

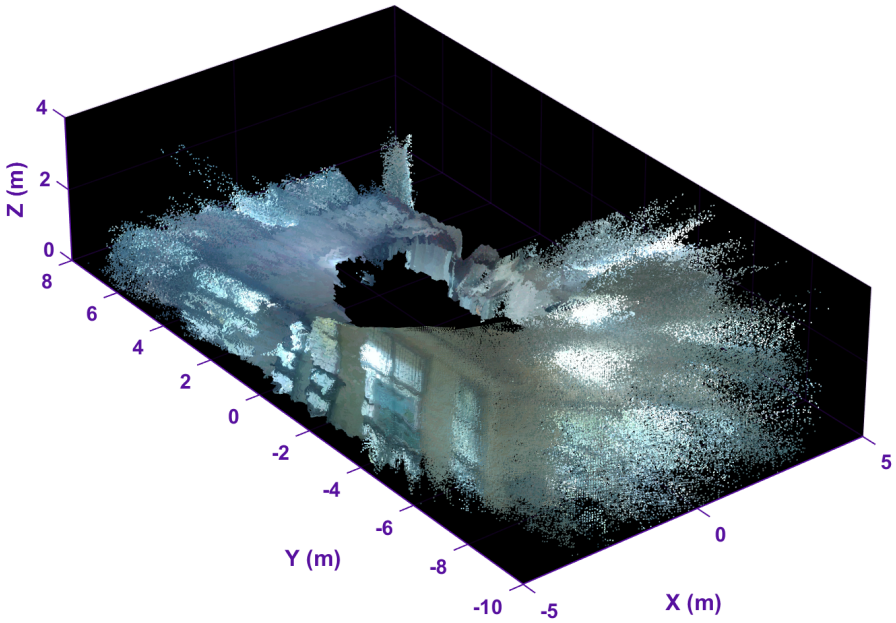


Figure 5.1.7: Manufacturing Lab

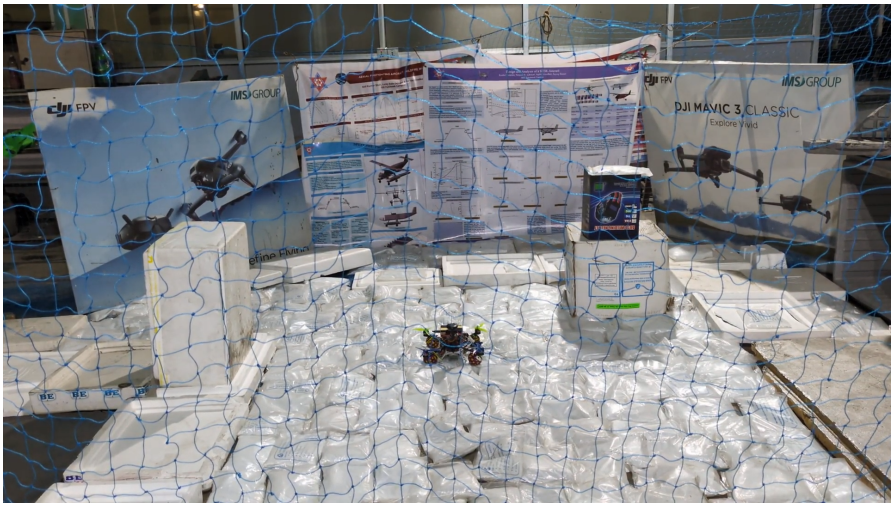


Figure 5.1.8: Test Environment inside IIEC

Since the main objective was to conduct flight tests of the quadcopter to be used in a GPS-denied environment, first an indoor environment of dimension 6m x 6m was created at IIEC, Pulchowk Campus. The indoor environment consists of different objects so that the quadcopter can see various features to localize itself taking reference from its environment more accurately. Styrofoam blocks were kept to cover the floor of the indoor environment so that if the quadcopter fell due to some errors in the initial testing phase, it would be safe without breaking the components of the quadcopter. Posters were pasted around the environment on all four sides so that the quadcopter wouldn't get lost without features in the environment. 0.5m x 0.5m x 0.3m cardboard boxes were stacked on top of each other to create obstacles while testing obstacle avoidance. Also, safety nylon nets were used to cover the environment from all four sides to ensure the quadcopter remains inside and won't get outside due to some drifts during the initial testing phase.

Flight testing in the indoor environment was done in three different configurations of the environment to ensure the robustness of the experiment efficiently. The configurations were a free environment without keeping obstacles, a single obstacle environment, and two obstacle environments as shown in Figure 5.1.15. First, the quadcopter was made to take off up to 0.5 m in an obstacle-free environment and then multiple goal points were provided inside that environment using the RViz graphical user interface to navigate the quadcopter throughout the whole environment. The 3D map and trajectory data were recorded as a database file which was used later for post-processing the results. A 3D point cloud map of the obstacle-free environment can be seen in Figure 5.1.15a which shows distinct environment features like posters, corners, and floors. Similarly, a 2D grid map of the environment can be seen in Figure 5.1.11a in which the boundary of the

environment and free space is represented by the colors black and white, respectively. Similarly, the red curve represents the trajectory of the quadcopter. Also, the start point and goal points are represented by blue dots and black stars respectively. The trajectory and altitude of the quadcopter can be seen in Figure 5.1.10. The trajectory plot signifies the quadcopter's path over time in a given space, showing how the drone moves from one point to another. This is crucial for localization accuracy to see how closely the trajectory follows the expected path. Also, a smooth trajectory indicates stable and accurate navigation. The mean and maximum deviation in altitude were obtained to be 0.024 m and 0.102 m respectively. Since, only 2D path planning algorithm was implemented, the altitude plot signifies that quadcopter navigates in 2D with very less error (4.8 %) in altitude.

Figure 5.1.9 illustrates the actual trajectory, color-coded by path error, against the trajectory connected by goal points. The color gradient indicates path error (in meters), which shows least path error, represented by blue, followed by an increase in path error, represented with red color. This visualization signifies deviations along the trajectory, supporting error analysis and performance evaluation. The mean and maximum path error was found to be 0.077 m and 0.412 m.

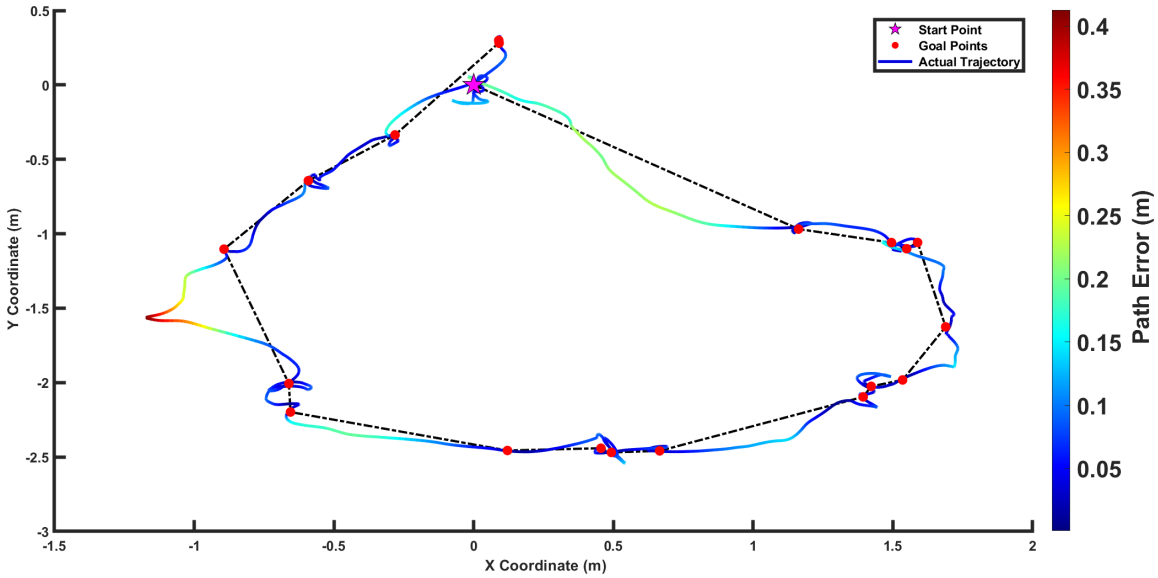
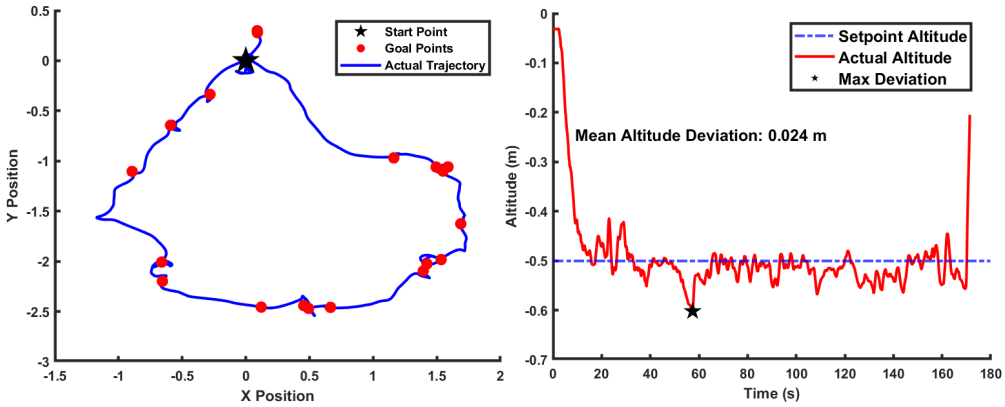
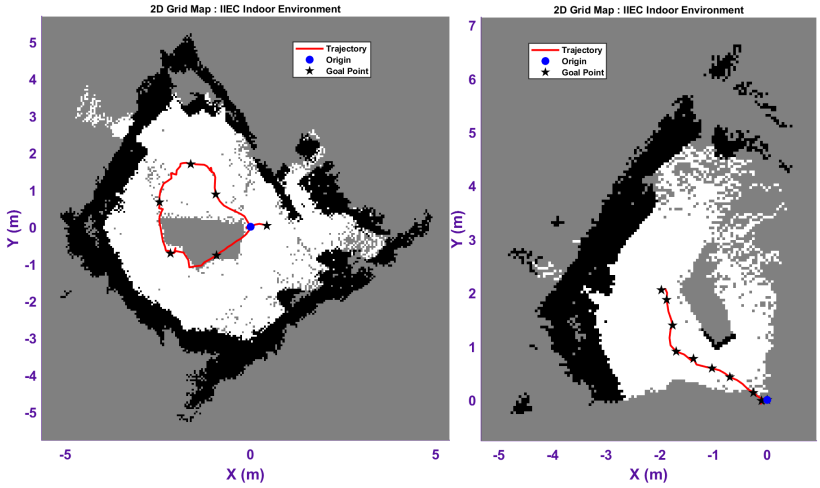


Figure 5.1.9: Path error measured from goal point to goal point: Obstacle Free Environment

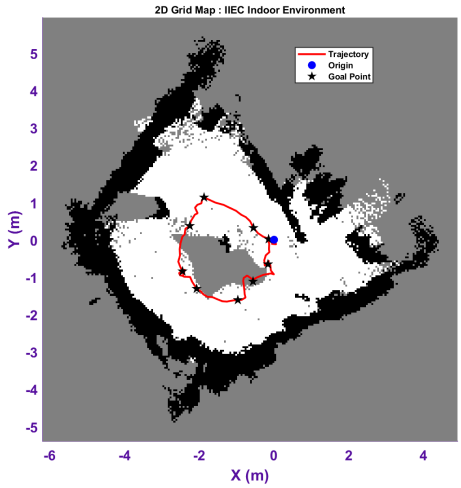


(a) Quadcopter Trajectory (b) Quadcopter Altitude

Figure 5.1.10: Obstacle Free Environment



(a) Obstacle Free Environment (b) Single Obstacle Environment

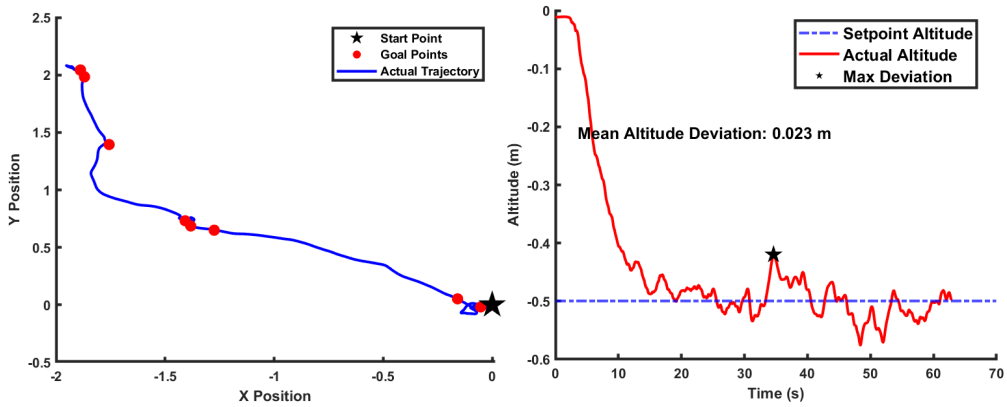


(c) Double Obstacle Environment

Figure 5.1.11: Indoor Environment at IIEC

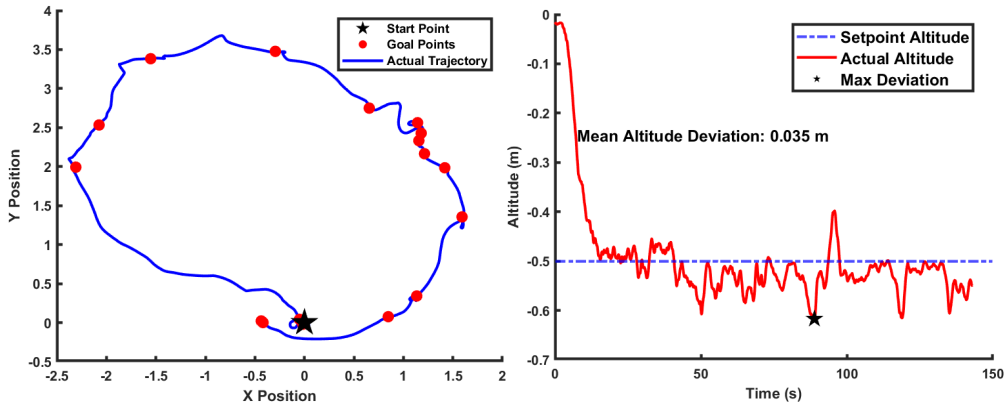
Similarly, for the next flight tests, one obstacle was kept inside the environment and

obstacle avoidance testing was done at first providing a goal point just behind the obstacle. After that the trajectory was generated automatically using the implemented path planning algorithm as shown in Figure 5.1.15. The 2D grid map and 3D point cloud map of the environment can be seen in the Figure 5.1.11b and 5.1.14b respectively. The trajectory and altitude of the quadcopter can be seen in Figure 5.1.12. The mean and maximum deviation in altitude were obtained to be 0.023 m and 0.0801 m respectively. Also, another flight test was done in which the quadcopter was made to rotate around the obstacle by providing multiple goal points using RViz. The 2D grid map and 3D point cloud map of the environment can be seen in Figures 5.1.11b and 5.1.14b respectively. The trajectory and altitude of the quadcopter can be seen in Figure 5.1.12. The mean and maximum deviation in altitude were obtained to be 0.035 m and 0.117 m respectively.



(a) Quadcopter Trajectory (b) Quadcopter Altitude

Figure 5.1.12: Single Obstacle Environment: Avoidance Only

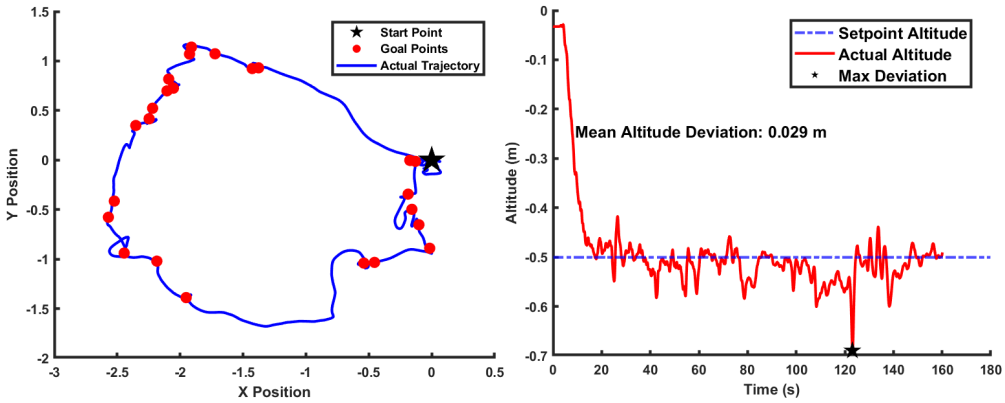


(a) Quadcopter Trajectory (b) Quadcopter Altitude

Figure 5.1.13: Single Obstacle Environment: Loop around obstacle

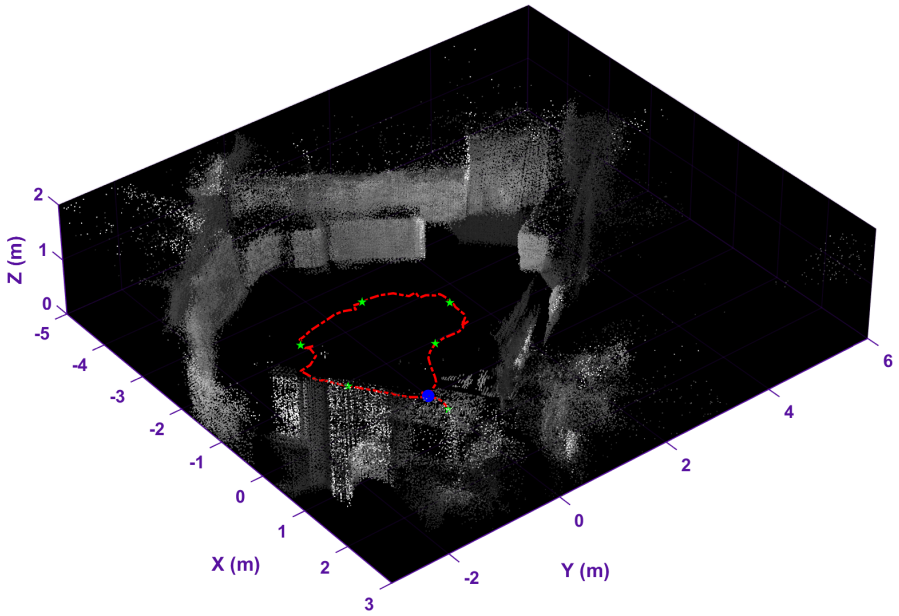
Lastly, two different obstacles were kept inside the environment at two different loca-

tions and the quadcopter was made to navigate through them autonomously by providing multiple goal points. After complete mapping of the environment, the quadcopter lands at the nearby location of the starting point. 2D grid map and 3D point cloud map of the environment can be seen in the Figure 5.1.11c and Figure 5.1.14c respectively which show two obstacles as well as the boundary of the indoor environment distinctly. The trajectory and altitude of the quadcopter can be seen in Figure 5.1.14. The mean and maximum deviation in altitude were obtained to be 0.029 m and 0.190 m respectively.

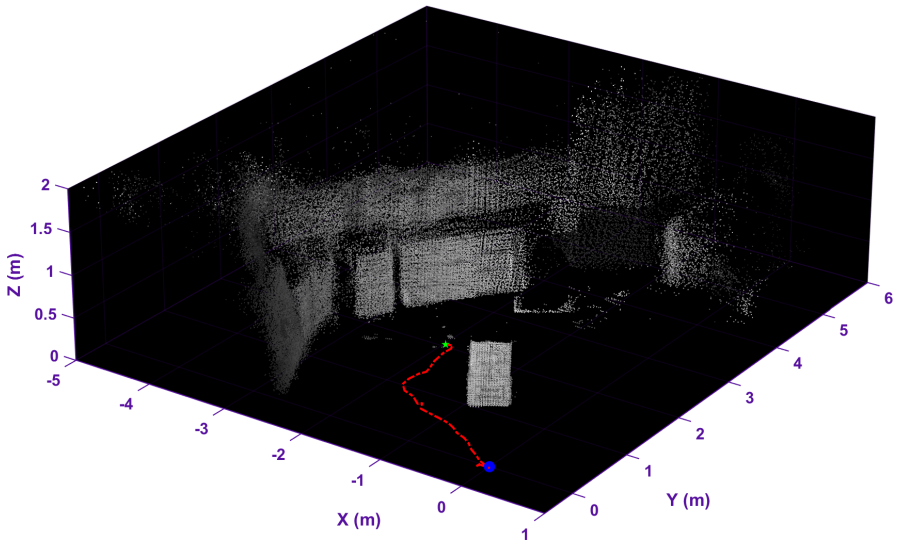


(a) Quadcopter Trajectory (b) Quadcopter Altitude

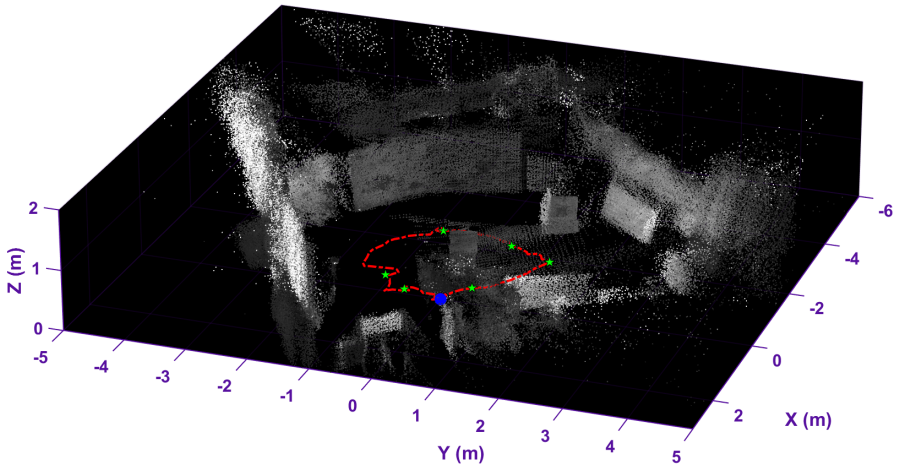
Figure 5.1.14: Double Obstacle Environment: Navigating through obstacle



(a) No Obstacle



(b) One Obstacle



(c) Two Obstacles

Figure 5.1.14: 3D Point Cloud Maps of the Indoor Environment at IIEC

5.1.0.3. Real World Environment

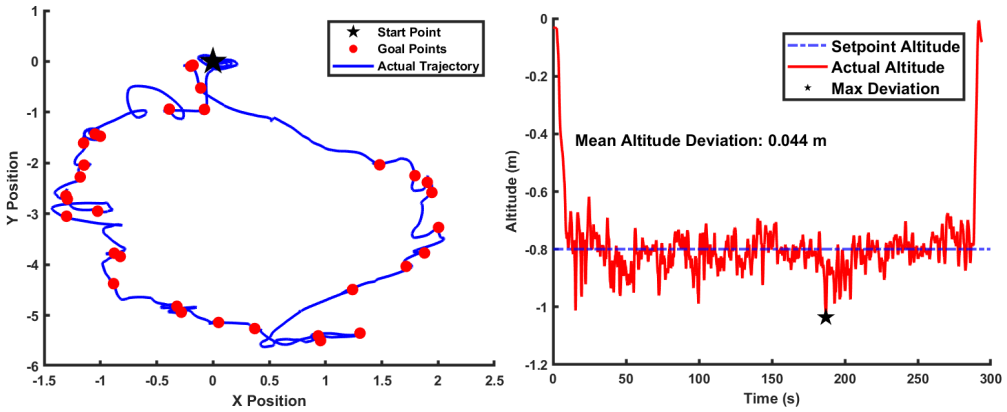
The autonomous quadcopter was also tested in complex real world environments. This was done to validate that the system can be applied to realistic operational environments and not just be limited to indoor settings. Three different real world scenarios were tested and are described below.

First was the garden area outside of CES building. The garden area outside the CES building was chosen to replicate an environment with a dense forest-like setting. This environment provides essential visual features for the SLAM algorithm to determine the quadcopter's pose. It consists of trees, bushes, and plants with enough spacing for the quadcopter to navigate between them. The quadcopter was flown at an altitude of 0.8 meters instead of 0.5 meters which was for indoor setting. The wind during the

Figure 5.1.15: Indoor Environment at IIEC, Pulchowk Campus

Figure 5.1.16: Forest Environment: In front of the CES Building

testing was $1ms^{-1}$. The results from this test demonstrated that the quadcopter was able to efficiently navigate to the commanded goals, even when some goal points were behind obstacles. It successfully maneuvered through the entire environment avoiding obstacles. Additionally, the quadcopter also completed a full loop of the test area and achieved a successful landing at its original starting location, which validated the effectiveness of the navigation system. The 2D grid map is shown in Figure 5.1.18 and 3D point cloud is shown in figure 5.1.19. The trajectory and altitude of the quadcopter can be seen in Figure 5.1.17. The mean and maximum deviation in altitude were obtained to be 0.044 m and 0.236 m respectively.



(a) Quadcopter Trajectory (b) Quadcopter Altitude

Figure 5.1.17: Forest: Navigating through trees

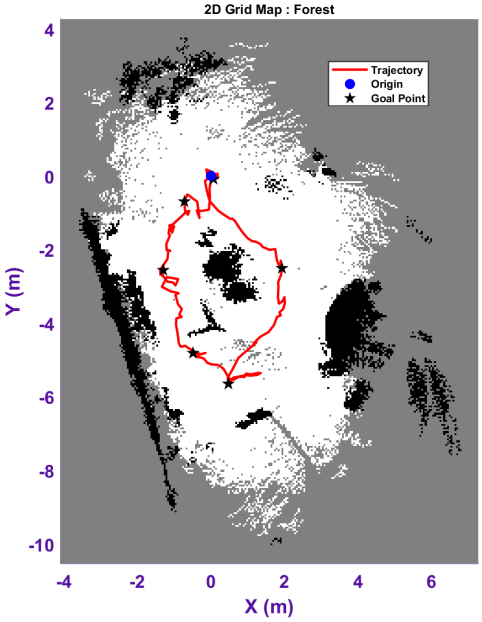


Figure 5.1.18: 2D Grid Map: Forest in front of CES

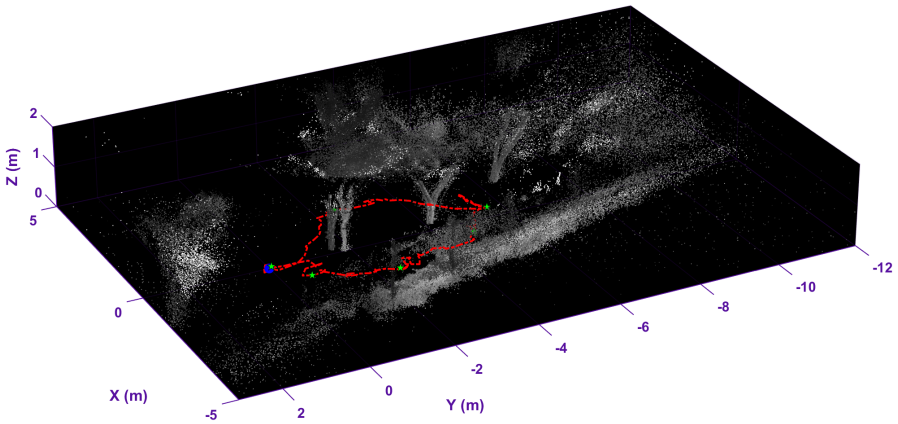


Figure 5.1.19: 3D Point Cloud Map: Forest infront of the CES building

Second was the Aerospace department's Manufacturing lab. The manufacturing lab had a door opening through which the quadcopter had to pass and navigate to the corner on the opposite end of the room. The environment also had enough visual features for SLAM. The quadcopter was flown at an altitude of 0.5 meters which was same as that of the other indoor settings. This test was also successful, with the quadcopter navigating through gaps while avoiding the doors. After passing through the door, other goal points were given to the far corner of the room, where the quadcopter yawed first and headed successfully to the other goal. Navigating through the second door which is situated at the end of lab was not possible due to to the gap being too small and the risk of damage to the laser cutter and even the quadcopter in case of any mishaps. Hence, the quadcopter was landed in midway of the lab. The 2D grid map of environment is shown in Figure 5.1.20 and 3D point cloud is in Figure 5.1.21. The trajectory and altitude of the quadcopter can be seen in Figure 5.1.22. The mean and maximum deviation in altitude were obtained to be 0.026 m and 0.127 m respectively.

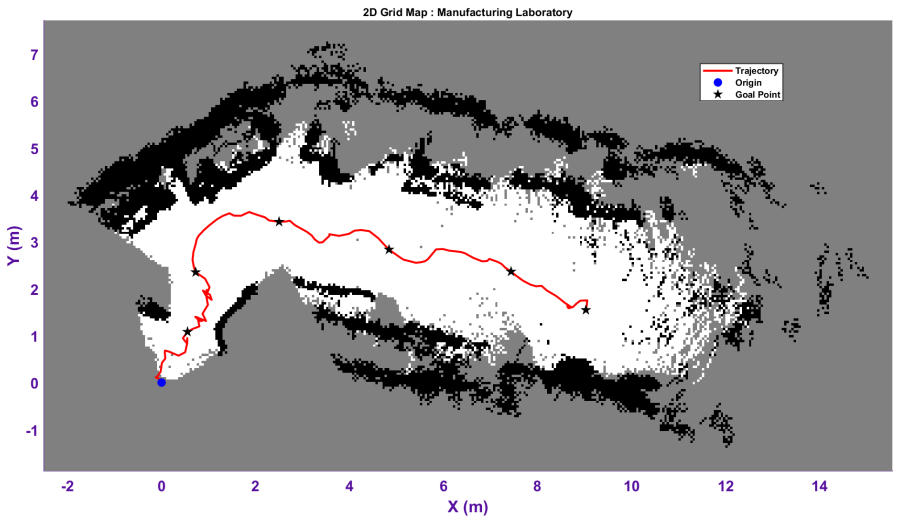


Figure 5.1.20: 2D Grid Map: Manufacturing Laboratory

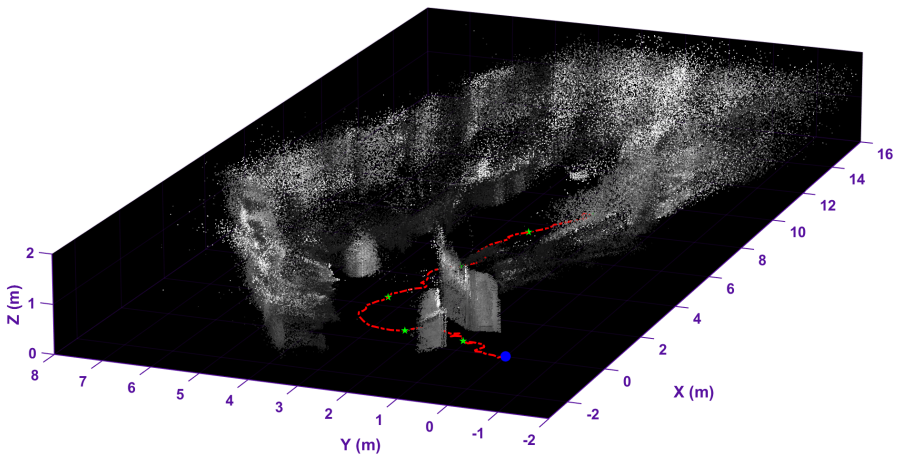
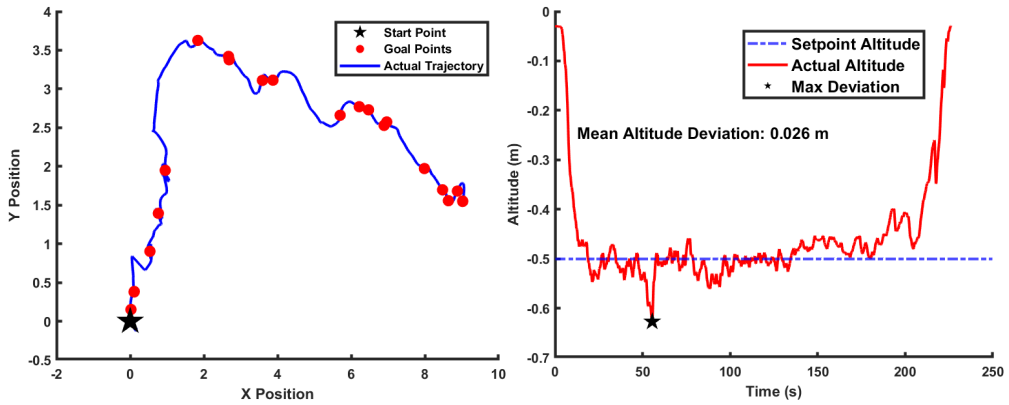


Figure 5.1.21: 3D Point Cloud Map: Manufacturing Laboratory



(a) Quadcopter Trajectory (b) Quadcopter Altitude

Figure 5.1.22: Manufacturing Laboratory

Last, was the senior classroom of the same Aerospace department. This environment[2] had a similar obstacle to that of the manufacturing lab. The quadcopter had to first navigate through the door and enter the class. Then, a static obstacle (chair) was placed in the path of the quadcopter. There were desks and benches on either side of the room which also acted as obstacles and visual cues for localization. The quadcopter was again flown at an altitude of 0.5 meters.

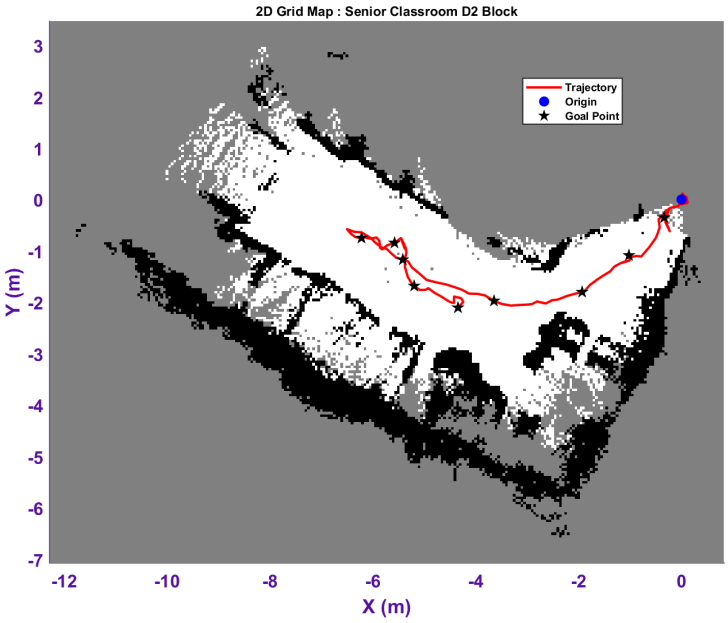
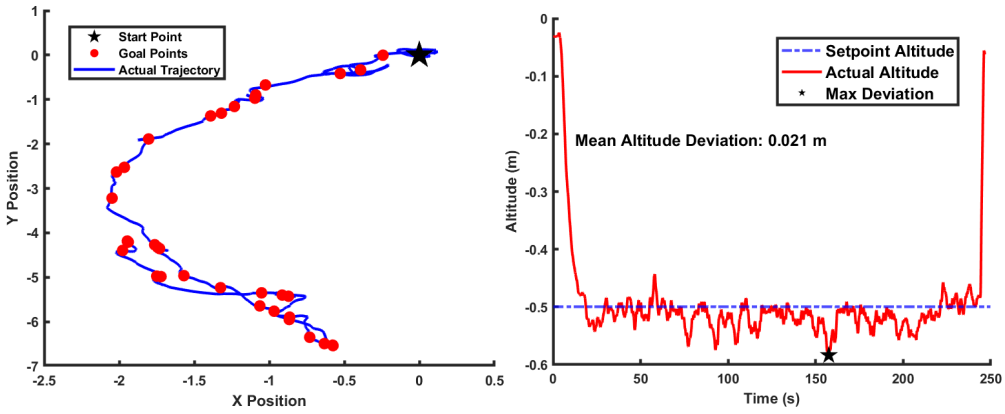


Figure 5.1.23: 2D Grid Map: Senior Classroom (D2 Block)

The 2D grid map is Figure 5.1.23 and Figure 5.1.25 is the 3D point cloud. The result from this test showed that the quadcopter executed the flight as intended by entering

[2]The testing environment for the flight tests in the Manufacturing Lab and the senior classroom is shown in Appendix C.

the room and avoiding the obstacles and door. The quadcopter navigated between the gap of the desk and benches which was right in the middle of the room. The trajectory and altitude of the quadcopter can be seen in Figure 5.1.24. The mean and maximum deviation in altitude were obtained to be 0.021 m and 0.084 m respectively.



(a) Quadcopter Trajectory (b) Quadcopter Altitude

Figure 5.1.24: Senior Classroom (BAS)

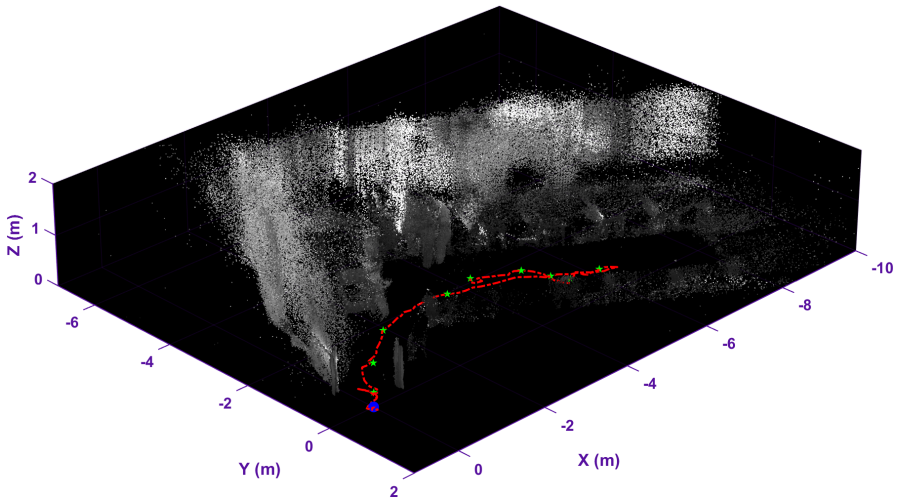


Figure 5.1.25: 3D Point Cloud Map: Senior Classroom (D2 Block)

5.2. Limitations

Some limitations of the project were:

1. Speed of the quadcopter: Due to the weight of the quadcopter and the onboard sensors, the speed of the qudacopter was severely impacted.

2. Processing power: The nature of the V-SLAM algorithm requires heavy computational power and it could hinder the speed at which the quadcopter moves as well as the speed at which the drone localizes itself. Camera resolutions and FPS needed to be reduced to enhance processing speeds.

3. Battery Power: There is a limit to the capacity of the onboard battery due to the limitations of the size of the quadcopter and the battery it can carry.

4. 3D Path Planing: Even though a 3D map has been successfully generated using the SLAM algorithm, the quadcopter is currently unable to plan a collision-free path in 3D.

5. Flight Time: Due to the quadcopter being heavy and V-SLAM requiring slow motion of drone to avoid losing VO, the flight time for drone is reduced significantly only 5 minutes at maximum.

6. Lack of MoCap System: The absence of a motion capture system restricted us to estimate errors on the estimated and tracked path for the quadcopter in indoor environments, leading to setting benchmarks and data quantization issues.

7. 3D Map Quality: The 3D point cloud map was generated using the infrared cameras and not RGB which hindered the ability of map to be interpreted. Number of point clouds were also reduced to enhance performance which made the map less detailed.

5.3. Problems Faced

Numerous problems were faced, particularly related to processing power and sensor data. One significant issue was the computational load required for real-time SLAM processing. To handle this, higher processing power is essential, considering options like the Intel NUC or NVIDIA Jetson series, including the Jetson Xavier, TX2, and Orin. These platforms provide the necessary computational capabilities to process the large amounts of data generated by the SLAM system in real-time.

Instead of using RGB and depth images for mapping, this project used IR and depth images. While this choice helped improve performance, it also led to challenges in the understanding of the generated map. The IR and depth images produce maps that are not intuitive or easily understood by a layman, as they lack color and visual features. However, these maps are perfect to the quadcopter's SLAM algorithm, allowing the drone to navigate effectively.

Next was the configuration of parameters within the RTAB-Map SLAM algorithm. In order to ensure real-time performance, several parameters were adjusted. While these adjustments optimized the system for lower computational demands, they also led to a trade-off in map detail. As a result, the 3D map produced contains fewer details compared to a fully detailed map, which may limit the ability to capture features of the environment. However, this reduction in map detail was necessary to maintain the overall performance of the quadcopter during real world testing.

Finally in path planning, the ROS generated paths were often straight or zigzag due to grid map, which were difficult for the quadcopter to follow. If a dynamic obstacle appeared, the quadcopter's path planning struggled to adjust its path, as these paths didn't account for such situations. Additionally, the update rates of both global and local planners was also difficult to tune. The paths were sometimes generated too close to obstacles, despite adding an inflation layer, which created potential collisions. These issues, alongside the processing power limitations were some of the hindrance faced during the project, but all these problems provided valuable knowledge and key take-aways for future improvements.

5.4. Budget Analysis

Table 5.4.1: Budget Estimation

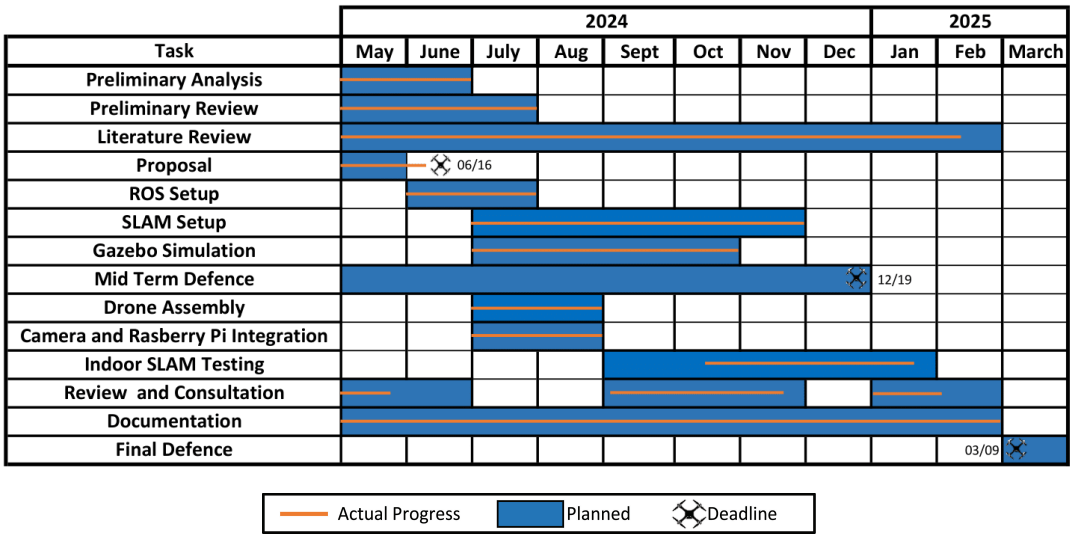
SN	Description	Quantity	Estimated Cost(NRs)
1	Raspberry Pi 4	1	20,000
2	Intel RealSense D435 Camera	1	45,000
3	PixHawk 4	1	45,000
4	Quadcopter Frame	1	8,000
5	Motor	4	10,800
6	ESC	4	5,500
7	Battery	1	5,000
8	Vl53l1x Lidar	1	2,000
9	Miscellaneous		6,000
Total			1,47,300

The project required a total budget of Rs.1,47,300. This budget made sure that the team could receive all the resources they needed while making the most of what was already available and to cut costs to be financially efficient.

5.5. Work Timeline

The following timeline represents the progress for the duration of the project with key milestones shown.

Table 5.5.1: Work Timeline



CHAPTER 6:
CONCLUSION AND FUTURE ENHANCEMENT

6.1. Conclusion

The project met all of the specific objectives outlined in the proposal, being achieved both in simulation and implemented in the real world as well.

First, the creation of a 3D map of the indoor environment was accomplished both in simulations and through handheld mapping. This demonstrated that the SLAM algorithm was effectively able to reconstruct a 3D environment. The SLAM successfully localized the system in simulation and handheld mapping for localization and autonomous navigation. Both in simulation and the real world, autonomous navigation was successful. The project achieved dependable autonomous navigation when applied to the real quadcopter. For autonomous navigation, the quadcopter's height was obtained from using external LiDAR sensor, while the X and Y poses were derived from SLAM. The quadcopter effectively navigated avoiding obstacles in various different scenarios, and optimized its path to provided goal positions.

Despite milestone results, several meaningful challenges and limitations were noticed while deploying the quadcopter in authentic real world scenarios. The quadcopter's capacity to carry out path planning and obstacle avoidance was restricted only to 2D. Since the real world is filled with cluttered and complex obstacles, this limitation made it challenging for the quadcopter to effectively navigate requiring 3D path planning.

In conclusion, the project successfully achieved the creation of 3D point cloud maps for various environments using SLAM and autonomous navigation successfully. Localization was also effectively implemented, along with the successful simulation and real world testing of navigation. Further, stable altitude was also achieved using a LiDAR, and fully dependable autonomous navigation and obstacle avoidance in indoor environments were successfully implemented. Outdoor testing in front of CES was also conducted, replicating a forest environment to assess the robustness and dependability of the autonomous quadcopter.

6.2. Scope for Future Enhancement

The following sectors can be enhanced in the future to make the quadcopter more robust and applicable for real world implementations:

1. Performance Analysis and Benchmarking: Conduct detailed quantitative evaluations of processing speed, power consumption, and flight endurance to set benchmarks.

2. Improving Processing Capabilities: Upgrade the companion computer from a Raspberry Pi to a more powerful processor for visual SLAM and navigation computations.

3. Increasing Flight Time: LiPo Battery with more energy density while low weight would be beneficial for significantly improving flight time.

4. 3D Path Planning: Implementing full 3D path planning instead of 2D planning can improve navigating in complex clutter environments which is crucial for aerial applications.

5. Offboard Processing and Remote Data Transfer: The quadcopter can transmit the camera feed and sensor data wirelessly to a more powerful ground control station, where SLAM and path planning calculations are performed. The computed trajectory and navigation commands can then be sent back to the quadcopter for execution, improving performance while leveraging more powerful computing resources.

6. Motion Capture for Ground Truth: Inhouse motion capture system can be built or integrated to provide precise ground truth positioning of the quadcopter. This allows for quantification of the tracked versus estimated trajectory, allowing for better evaluation of SLAM and path planning algorithms.

7. SLAM and Path Planning Optimization: The parameters of the path planning procedure and the SLAM algorithm should be fine tuned for effective operation on the Raspberry Pi or its alternative companion computer. This means lowering the computing needs while preserving acceptable performance by setting parameters to their lowest allowable values. This allows visual SLAM to perform within the Raspberry Pi's processing limits, while achieving good results.

REFERENCES

[1] S. G. Jones, J. Harrington, C. K. Reld, and M. Strohmeyer, "Combined arms warfare and unmanned aircraft systems: A new era of strategic competition," tech. rep., Center for Strategic and International Studies (CSIS), 2022.

[2] T. Apodaca, "An inside look of Amazon's drones deliveries in San Joaquin County - CBS Sacramento," Oct. 2023.

[3] H. Taheri and Z. C. Xia, "SLAM; definition and evolution," *Engineering Applications of Artificial Intelligence*, vol. 97, p. 104032, Jan. 2021.

[4] K. Di, W. Wan, H. Zhao, Z. Liu, R. Wang, and F. Zhang, "Progress and applications of visual slam," *Journal of Geodesy and Geoinformation Science*, vol. 2, no. 2, p. 38, 2019.

[5] Raspberry Pi Ltd, "Buy a Raspberry Pi 4 Model B." `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/`.

[6] Intel RealSense, "Intel realsense depth camera d435." `https://www.intelrealsense.com/depth-camera-d435/`, 2025. Accessed: 2025-03-05.

[7] Holybro, "St vl53l1x lidar." `https://holybro.com/products/st-vl53l1x-lidar`, 2025. Accessed: 2025-03-05.

[8] M. F. Ahmed, K. Masood, V. Fremont, and I. Fantoni, "Active SLAM: A Review on Last Decade," *Sensors (Basel, Switzerland)*, vol. 23, p. 8097, Sept. 2023.

[9] A. Annaiyan, M. A. Olivares-Mendez, and H. Voos, "Real-time graph-based SLAM in unknown environments using a small UAV," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 1118–1123, IEEE.

[10] A. R. Khairuddin, M. S. Talib, and H. Haron, "Review on simultaneous localization and mapping (SLAM)," in *2015 IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, (Penang, Malaysia), pp. 85–90, IEEE, Nov. 2015.

[11] J. Leonard and H. Durrant-Whyte, "Mobile robot localization by tracking geometric beacons," *IEEE Transactions on Robotics and Automation*, vol. 7, pp. 376–382, June 1991.

[12] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE Robotics Automation Magazine*, 2006.

[13] R. Smith, M. Self, and P. Cheeseman, *Estimating Uncertain Spatial Relationships in Robotics*, pp. 167–193. New York, NY: Springer New York, 1990.

[14] H. Durrant-Whyte, D. Rye, and E. Nebot, "Localization of autonomous guided vehicles," in *Robotics Research* (G. Giralt and G. Hirzinger, eds.), (London), pp. 613–625, Springer London, 1996.

[15] Y. Chen, Y. Zhou, Q. Lv, and K. Deveerasetty, "A review of v-slam *," pp. 603–608, 08 2018.

[16] C. Debeunne and D. Vivet, "A review of visual-lidar fusion based simultaneous localization and mapping," *Sensors*, vol. 20, no. 7, 2020.

[17] X. Gao and T. Zhang, *Introduction to Visual SLAM: From Theory to Practice*. Singapore: Springer Singapore, 2021.

[18] D. Scaramuzza and F. Fraundorfer, "Visual Odometry [Tutorial]," *IEEE Robotics & Automation Magazine*, vol. 18, pp. 80–92, Dec. 2011.

[19] K. Beevers and W. Huang, "Slam with sparse sensing," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pp. 2285–2290, 2006.

[20] L. M. Paz, J. D. TardÓs, and J. Neira, "Divide and conquer: Ekf slam in $o(n)$," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1107–1120, 2008.

[21] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "Orb-slam: A versatile and accurate monocular slam system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[22] Z. Yuan, "Advances in monocular orb-slam system: a review," *Theoretical and Natural Science*, vol. 41, pp. 75–80, 11 2024.

[23] K. Wang, L. Kooistra, R. Pan, W. Wang, and J. Valente, "UAV-based simultaneous localization and mapping in outdoor environments: A systematic scoping review," *Journal of Field Robotics*, p. rob.22325, Apr. 2024.

[24] M. Labbé and F. Michaud, "Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *Journal of Field Robotics*, vol. 36, p. 416–446, Oct. 2018.

[25] Z. Kong and Q. Lu, "A brief review of simultaneous localization and mapping," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pp. 5517–5522, 2017.

[26] R. Catania, "Politecnico di Torino Master's Degree in Mechatronic Engineering," 2023.

[27] J.-X. Wu and Y.-P. Hsu, "The Indoor Localization of a Vision-Based Unmanned Aerial Vehicle," *International Journal of iRobotics*, vol. 4, pp. 1–6, Apr. 2021.

[28] X. Chen, X. Zhu, and C. Liu, "Real-Time 3D Reconstruction of UAV Acquisition System for the Urban Pipe Based on RTAB-Map," *Applied Sciences*, vol. 13, p. 13182, Jan. 2023. Number: 24 Publisher: Multidisciplinary Digital Publishing Institute.

[29] Matlabbe Contributors, "Rtab-map drone example." `https://github.com/matlabbe/rtabmap_drone_example`, 2025. Accessed: 2025-03-05.

[30] S. Bortoff, "Path planning for UAVs," in *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No.00CH36334)*, (Chicago, IL, USA), pp. 364–368 vol.1, IEEE, 2000.

[31] D. Rathbun, S. Kragelund, A. Pongpunwattana, and B. Capozzi, "An evolution based path planning algorithm for autonomous motion of a UAV through uncertain environments," in *Proceedings. The 21st Digital Avionics Systems Conference*, vol. 2, (Irvine, CA, USA), pp. 8D2–1–8D2–12, IEEE, 2002.

[32] J. Canny, " A new algebraic method for robot motion planning and real geometry ," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, (Los Alamitos, CA, USA), pp. 39–48, IEEE Computer Society, Oct. 1987.

[33] P. Bhattacharya and M. L. Gavrilova, " Voronoi diagram in optimal path planning ," in *2007 4th International Symposium on Voronoi Diagrams in Science and Engineering*, (Los Alamitos, CA, USA), pp. 38–47, IEEE Computer Society, July 2007.

[34] P. Marin-Plaza, A. Hussein, D. Martin, and A. d. l. Escalera, "Global and local path planning study in a ros-based research platform for autonomous vehicles," *Journal of Advanced Transportation*, vol. 2018, no. 1, p. 6392697, 2018.

[35] M. Hwangbo, J. Kuffner, and T. Kanade, "Efficient two-phase 3d motion planning for small fixed-wing uavs," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 1035–1041, 2007.

[36] D. Rachmawati and L. Gustin, "Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem," *Journal of Physics: Conference Series*, vol. 1566, p. 012061, June 2020.

[37] Y. Y. Win and H. S. Hlaing, "Shortest Path Analysis Based on Dijkstra's Algorithm in Myanmar Road Network," vol. 06, no. 10, 2019.

[38] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[39] P. Marin-Plaza, A. Hussein, D. Martin, and A. D. L. Escalera, "Global and local path planning study in a ROS-based research platform for autonomous vehicles," vol. 2018, pp. 1–10.

[40] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics & Automation Magazine*, vol. 4, pp. 23–33, Mar. 1997.

[41] C. Rösmann, F. Hoffmann, and T. Bertram, "Kinodynamic trajectory optimization and control for car-like robots," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5681–5686, 2017.

[42] S. Macenski, S. Singh, F. Martin, and J. Gines, "Regulated Pure Pursuit for Robot Path Tracking," May 2023.

[43] A. Gibiansky, "Quadcopter dynamics, simulation, and control," 2013. Accessed:

2025-03-05.

[44] N. Q. Vo, “Nonlinear geometric control of a quadrotor with a cable-suspended load,” 2017.

[45] G. V. Raffo, M. G. Ortega, and F. R. Rubio, “An integral predictive/nonlinear H control structure for a quadrotor helicopter,” *Automatica*, vol. 46, pp. 29–39, Jan. 2010.

[46] S. Bhattarai, K. Poudel, N. Bhatta, S. Mahat, S. Bhattarai, and K. S. Thapa Magar, “Modeling and Development of Baseline Guidance Navigation and Control System for Medical Delivery UAV,” in *2018 AIAA Information Systems-AIAA Infotech @ Aerospace*, (Kissimmee, Florida), American Institute of Aeronautics and Astronautics, Jan. 2018.

[47] F. A. C. MOLINA, “Graphslam algorithm: Implementation for solving simultaneous localization and mapping,” *Repositorio Académico de la Universidad de Chile*, n.d. Available: https://repositorio.uchile.cl/bitstream/handle/2250/139093/Graphslam-algorithm-implementation-for-solving-simultaneous-localization-and-mapping.pdf?sequence=1isAllowed=y, Accessed: Dec. 18, 2024.

[48] D. Scaramuzza and Z. Zhang, “Visual-inertial odometry of aerial robots,” *CoRR*, vol. abs/1906.03289, 2019.

[49] S. Ghambari, M. Golabi, L. Jourdan, J. Lepagnot, and L. Idoumghar, “UAV path planning techniques: A survey,” *RAIRO - Operations Research*, vol. 58, pp. 2951–2989, July 2024.

[50] S. Biswas, *Real-Time Path Planning for a Swarm of Autonomous Systems*. PhD thesis, UNSW Sydney, 2019.

[51] M. Šeda, “Roadmap Methods vs. Cell Decomposition in Robot Motion Planning,”

[52] S. Francis, *Cooperative Path Planning for Autonomous Ground Vehicles Using 3D Sensor in Cluttered Environment*. PhD thesis, UNSW Sydney, 2013.

[53] A. Swingler, “A Cell Decomposition Approach to Robotic Trajectory Planning via Disjunctive Programming,” 2012.

[54] A. Abbadi and V. Prenosil, "SAFE PATH PLANNING USING CELL DECOMPOSITION APPROXIMATION,"

[55] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *The International Journal of Robotics Research*, vol. 5, no. 1, pp. 90–98, 1986.

[56] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, and Y. Xia, "Survey of Robot 3D Path Planning Algorithms," *Journal of Control Science and Engineering*, vol. 2016, pp. 1–22, 2016.

[57] M. Jones, S. Djahel, and K. Welsh, "Path-Planning for Unmanned Aerial Vehicles with Environment Complexity Considerations: A Survey," *ACM Computing Surveys*, vol. 55, pp. 1–39, Nov. 2023.

[58] David V. Lue, "Navfn - ros wiki." `https://wiki.ros.org/navfn`, 2025. Accessed: 2025-03-05.

[59] L. Liu, X. Wang, X. Yang, H. Liu, J. Li, and P. Wang, "Path planning techniques for mobile robots: Review and prospect," *Expert Systems with Applications*, vol. 227, p. 120254, Oct. 2023.

[60] Eitan Marder-Eppstein, "dwa_local_planner ros wiki." `http://wiki.ros.org/dwa_local_planner`, 2025. Accessed: 2025-03-05.

[61] ROS Navigation2 Contributors, "Navigation2 regulated pure pursuit controller." `https://github.com/ros-navigation/navigation2/blob/main/nav2_regulated_pure_pursuit_controller/README.md`, 2025. Accessed: 2025-03-05.

[62] Eitan Marder-Eppstein, "move_base - ros wiki." `https://wiki.ros.org/move_base`, 2025. Accessed: 2025-03-05.

APPENDICES

APPENDIX A: C++ Code for operating quadcopter in Offboard Mode

Listing 6.1: Offboard_node.cpp

```
/**
 * @file offb_node.cpp
*/

#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/Joy.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/CommandLong.h>
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>
#include <mavros_msgs/PositionTarget.h>
#include <tf/transform_listener.h>
#include <sensor_msgs/Range.h>

#define VELOCITY2D_CONTROL 0b011111000011
#define VELOCITY_CONTROL 0b011111000111
#define POSITION_CONTROL 0b101111111000
unsigned short velocity_mask = VELOCITY2D_CONTROL;

float lidar_distance = 0.5;  // Default value, set it to a safe
    height
float initial_lidar_altitude = 0.5;  // Save initial altitude

mavros_msgs::PositionTarget current_goal;
ros::Time lastTwistReceived;

mavros_msgs::State current_state;
void state_cb(const mavros_msgs::State::ConstPtr& msg) {
    current_state = *msg;
}

void lidar_cb(const sensor_msgs::Range::ConstPtr& msg) {
    lidar_distance = msg->range;  // Update the Z-coordinate based on
        LIDAR data
    ROS_INFO(\texttt{"LIDAR_Distance:%f"}, lidar_distance);
}

void twist_cb(const geometry_msgs::Twist::ConstPtr& msg) {
    if (current_goal.type_mask == POSITION_CONTROL) {
```

```
40          ROS_INFO("Switch_to_velocity_control");
41      }
42      current_goal.coordinate_frame = mavros_msgs::PositionTarget::
            FRAME_BODY_NED;
43      current_goal.type_mask = velocity_mask;
44      current_goal.velocity.x = msg->linear.x;
45      current_goal.velocity.y = msg->linear.y;
46      current_goal.velocity.z = (velocity_mask == VELOCITY2D_CONTROL ?
            0 : msg->linear.z);
47      current_goal.position.z = lidar_distance;
48      current_goal.yaw_rate = msg->angular.z;
49      lastTwistReceived = ros::Time::now();
50  }
51
52  void joy_cb(const sensor_msgs::Joy::ConstPtr& msg) {
53      if (msg->buttons[5] == 1) {
54          // When holding right trigger, accept velocity in Z
55          velocity_mask = VELOCITY_CONTROL;
56      } else {
57          velocity_mask = VELOCITY2D_CONTROL;
58      }
59  }
60
61  int main(int argc, char** argv) {
62      ros::init(argc, argv, "lidar");
63      ros::NodeHandle nh;
64
65      ros::Subscriber state_sub = nh.subscribe<mavros_msgs::State>("
            mavros/state", 10, state_cb);
66      ros::Publisher local_pos_pub = nh.advertise<mavros_msgs::
            PositionTarget>("mavros/setpoint_raw/local", 1);
67      ros::Publisher vision_pos_pub = nh.advertise<geometry_msgs::
            PoseStamped>("mavros/vision_pose/pose", 1);
68      ros::ServiceClient arming_client = nh.serviceClient<mavros_msgs::
            CommandBool>("mavros/cmd/arming");
69      ros::ServiceClient command_client = nh.serviceClient<mavros_msgs
            ::CommandLong>("mavros/cmd/command");
70      ros::ServiceClient set_mode_client = nh.serviceClient<mavros_msgs
            ::SetMode>("mavros/set_mode");
71      ros::Subscriber twist_sub = nh.subscribe<geometry_msgs::Twist>("/
            cmd_vel", 1, twist_cb);
72      ros::Subscriber joy_sub = nh.subscribe<sensor_msgs::Joy>("/joy",
            1, joy_cb);
73      ros::Subscriber lidar_sub = nh.subscribe<sensor_msgs::Range>("
            mavros/distance_sensor/hrlv_ez4_pub", 5, lidar_cb);
74
75      ros::Rate rate(50.0);
```

```
76
77      while (ros::ok() && !current_state.connected) {
78          ros::spinOnce();
79          rate.sleep();
80      }
81
82      mavros_msgs::SetMode offb_set_mode;
83      offb_set_mode.request.custom_mode = "OFFBOARD";
84
85      mavros_msgs::CommandBool arm_cmd;
86      arm_cmd.request.value = true;
87
88      mavros_msgs::CommandLong disarm_cmd;
89      disarm_cmd.request.broadcast = false;
90      disarm_cmd.request.command = 400;
91
92      ros::Time last_request = ros::Time::now();
93      lastTwistReceived = ros::Time::now();
94
95      tf::TransformListener listener;
96
97      ROS_INFO("Setting_offboard_mode...(5sec)");
98      ros::spinOnce();
99      if (!listener.waitForTransform("/map", "/base_link", ros::Time(0)
            , ros::Duration(5))) {
100         ROS_ERROR("Cannot_get_current_position_between_/map_and_/
                base_link");
101         return -1;
102     }
103
104     try {
105         tf::StampedTransform visionPoseTf;
106         listener.lookupTransform("/map", "/base_link", ros::Time(0),
                visionPoseTf);
107
108         current_goal.coordinate_frame = mavros_msgs::PositionTarget::
                FRAME_LOCAL_NED;
109         current_goal.type_mask = POSITION_CONTROL;
110         current_goal.position.x = visionPoseTf.getOrigin().x();
111         current_goal.position.y = visionPoseTf.getOrigin().y();
112         current_goal.position.z = initial_lidar_altitude;  // Use
                initial altitude
113         current_goal.yaw = tf::getYaw(visionPoseTf.getRotation());
114         current_goal.velocity.x = 0;
115         current_goal.velocity.y = 0;
116         current_goal.velocity.z = 0;
117         current_goal.yaw_rate = 0;
```

```
118         current_goal.acceleration_or_force.x = 0;
119         current_goal.acceleration_or_force.y = 0;
120         current_goal.acceleration_or_force.z = 0;
121         ROS_INFO("Initial_position=(%f,%f,%f)_yaw=%f",
122                  current_goal.position.x,
123                  current_goal.position.y,
124                  current_goal.position.z,
125                  current_goal.yaw);
126     } catch (tf::TransformException& ex) {
127         ROS_ERROR("%s", ex.what());
128         return -1;
129     }
130 
131     for (int i = 100; ros::ok() && i > 0; --i) {
132         local_pos_pub.publish(current_goal);
133         ros::spinOnce();
134         rate.sleep();
135     }
136 
137     geometry_msgs::PoseStamped current_pose;
138     current_pose.header.frame_id = "map";
139 
140     while (ros::ok()) {
141         tf::StampedTransform visionPoseTf;
142         try {
143             listener.lookupTransform("/map", "/base_link", ros::Time
                    (0), visionPoseTf);
144 
145             current_pose.pose.position.x = visionPoseTf.getOrigin().x
                    ();
146             current_pose.pose.position.y = visionPoseTf.getOrigin().y
                    ();
147             current_pose.pose.position.z = lidar_distance;
148             current_pose.pose.orientation.x = visionPoseTf.
                    getRotation().x();
149             current_pose.pose.orientation.y = visionPoseTf.
                    getRotation().y();
150             current_pose.pose.orientation.z = visionPoseTf.
                    getRotation().z();
151             current_pose.pose.orientation.w = visionPoseTf.
                    getRotation().w();
152         } catch (tf::TransformException& ex) {
153             ROS_ERROR("%s", ex.what());
154         }
155 
156         if (current_state.mode != "OFFBOARD" &&
157             (ros::Time::now() - last_request > ros::Duration(5.0))) {
```

```
158             if (set_mode_client.call(offb_set_mode) && offb_set_mode.
                    response.mode_sent) {
159                 ROS_INFO("Offboard_enabled");
160                 ROS_INFO("Vehicle_arming...(5sec)");
161             }
162             last_request = ros::Time::now();
163         } else {
164             if (!current_state.armed &&
165                 !(current_goal.velocity.z < -0.4 && current_goal.
                        yaw_rate < -0.4) &&
166                 (ros::Time::now() - last_request > ros::Duration(5.0)
                        )) {
167                 if (arming_client.call(arm_cmd) && arm_cmd.response.
                        success) {
168                     ROS_INFO("Vehicle_armed");
169                     ROS_INFO("Take_off_to_position=(%f,%f,%f)_yaw=%f"
                            ,
170                             current_goal.position.x,
171                             current_goal.position.y,
172                             current_goal.position.z,
173                             current_goal.yaw);
174                 }
175                 last_request = ros::Time::now();
176             } else if (current_goal.velocity.z < -0.4 && current_goal
                    .yaw_rate < -0.4 &&
177                     (ros::Time::now() - last_request > ros::
                            Duration(5.0))) {
178                 if (command_client.call(disarm_cmd) && disarm_cmd.
                        response.success) {
179                     ROS_INFO("Vehicle_disarmed");
180                     ros::shutdown();
181                 } else {
182                     ROS_INFO("Disarming_failed!_Still_in_flight?");
183                 }
184                 last_request = ros::Time::now();
185             }
186         }
187 
188         current_goal.header.stamp = ros::Time::now();
189 
190         if (current_goal.header.stamp.toSec() - lastTwistReceived.
                toSec() > 1 &&
191             current_goal.type_mask != POSITION_CONTROL) {
192             current_goal.coordinate_frame = mavros_msgs::
                    PositionTarget::FRAME_LOCAL_NED;
193             current_goal.type_mask = POSITION_CONTROL;
194             current_goal.position.x = current_pose.pose.position.x;
```

```
195             current_goal.position.y = current_pose.pose.position.y;
196             current_goal.position.z = initial_lidar_altitude;  //
                    Maintain initial altitude
197             tfScalar yaw, pitch, roll;
198             tf::Matrix3x3 mat(tf::Quaternion(current_pose.pose.
                    orientation.x,
199                                              current_pose.pose.
                                                    orientation.y,
200                                              current_pose.pose.
                                                    orientation.z,
201                                              current_pose.pose.
                                                    orientation.w));
202             mat.getEulerYPR(yaw, pitch, roll);
203             current_goal.yaw = yaw;
204             ROS_INFO("Switch_to_position_control(x=%f,y=%f,z=%f,yaw=%
                    f)",
205                     current_goal.position.x,
206                     current_goal.position.y,
207                     current_goal.position.z,
208                     current_goal.yaw);
209         }
210
211         current_pose.header.stamp = current_goal.header.stamp;
212         local_pos_pub.publish(current_goal);
213         vision_pos_pub.publish(current_pose);
214
215         ros::spinOnce();
216         rate.sleep();
217     }
218
219     return 0;
220 }
```

APPENDIX B: Source code repository for the project

This repository is hosted on GitHub and contains all the launch files, urdf, and autonomous launch code required for both simulation and real implementation of the project.

Here is the GitHub repository link:

https://github.com/77bas-SLAMdrone/indoor-slam-drone

Feedbacks and suggestion by all is highly encouraged. Authors feel that if anybody is likely to continue this project, they can follow this repository and continue from that. It is encouraged to browse the file, submit issues, and even contribute to the project if they are interested. Any input from the community will be valued and acknowledged.

APPENDIX C: Real World Environments

Figure 6.2.1: Senior Classroom D2 Block

Figure 6.2.2: Manufacturing Lab of Aerospace Department