A project report on

EXPLORATION AND NAVIGATION OF GPS DENIED ENVIRONMENTS USING DRONE SWARM

Submitted in partial fulfillment for the award of the degree of

Bachelor of Technology in Computer Science and Engineering with Specialization in Artificial Intelligence and Robotics

by

AADITYA BAGADDEO (20BRS1089)

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

April,2024

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

DECLARATION

I hereby declare that the thesis entitled "EXPLORATION AND MAPPING OF GPS DENIED ENVIRONMENTS USING DRONE SWARM" submitted by me, for the award of the degree of Bachelor of Technology in Computer Science and Engineering with Specialization in Artificial Intelligence and Robotics, Vellore Institute of Technology, Chennai is a record of bonafide work carried out by me under the supervision of Dr Sathian D.

I further declare that the work reported in this thesis has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Chennai

Date: 25/03/2024 Signature of the Candidate

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

School of Computer Science and Engineering

CERTIFICATE

This is to certify that the report entitled **"**EXPLORATION AND MAPPING OF GPS DENIED ENVIRONMENTS USING DRONE SWARM**"** is prepared and submitted by **Aaditya Bagaddeo** (**20BRS1089**) to Vellore Institute of Technology, Chennai, in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology in Computer Science and Engineering with Specialization in Artificial Intelligence and Robotics** programme is a bonafide record carried out under my guidance. The project fulfills the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission. The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certified.

Signature of the Guide:

Name: Dr./Prof. Sathian D
Date: 25/03/2034

Signature of the Internal Examiner	Signature of the External Examiner
Name:	Name:
Date:	Date:

Approved by the Head of Department,
B.Tech. CSE with Specialization in Artificial Intelligence and Robotics

Name: Dr. Harini S

Date: (Seal of SCOPE)

ABSTRACT

Indoor mapping using a swarm of drones is indispensable for a multitude of reasons. Firstly, drones excel at inspecting indoor spaces, navigating through challenging areas inaccessible to humans, and swiftly identifying structural or equipment issues for prompt maintenance. Additionally, these drones contribute significantly to architecture planning by generating precise 3D maps, enabling architects and designers to optimize space utilization for efficient building design. In the context of disaster relief, drones play a pivotal role in search and rescue operations, rapidly assessing indoor structures, locating survivors, and guiding rescue teams to affected areas. For large spaces like warehouses, drone mapping facilitates improved inventory management, optimized storage layouts, and streamlined logistics operations. Moreover, the detailed and exhaustive data collected by a swarm of drones proves invaluable for analytics, decision-making, and enhancing overall operational efficiency across diverse industries. In essence, the utilization of drone swarms for indoor mapping empowers organizations with unparalleled insights, efficiency gains, and informed decision-making in managing and optimizing indoor spaces.

Key Areas of Effect:

- Inspection
- Architecture planning
- Disaster Relief
- Mapping Large spaces like warehouses
- Detailed and Exhaustive Data

ACKNOWLEDGEMENT

It is my pleasure to express with deep sense of gratitude to Dr Sathian D. , Senior Assistant Professor Grade 2, School of Computer Science and Engineering, Vellore Institute of Technology, Chennai, for his constant guidance, continual encouragement, understanding; more than all, he taught me patience in my endeavor. My association with him is not confined to academics only, but it is a great opportunity on my part to work with an intellectual and expert in the field of Drone Automation and Robotics.

It is with gratitude that I would like to extend my thanks to the visionary leader Dr. G. Viswanathan our Honorable Chancellor, Mr. Sankar Viswanathan, Dr. Sekar Viswanathan, Dr. G V Selvam Vice Presidents, Dr. Sandhya Pentareddy, Executive Director, Ms. Kadhambari S. Viswanathan, Assistant Vice-President, Dr. V. S. Kanchana Bhaaskaran Vice-Chancellor, Dr. T. Thyagarajan Pro-Vice Chancellor, VIT Chennai and Dr. P. K. Manoharan, Additional Registrar for providing an exceptional working environment and inspiring all of us during the tenure of the course.

Special mention to Dr. Ganesan R, Dean, Dr. Parvathi R, Associate Dean Academics, Dr. Geetha S, Associate Dean Research, School of Computer Science and Engineering, Vellore Institute of Technology, Chennai for spending their valuable time and efforts in sharing their knowledge and for helping us in every aspect.

In jubilant state, I express ingeniously my whole-hearted thanks to Dr. Harini S, Head of the Department, B.Tech. CSE with Specialization in Artificial Intelligence & Robotics and the Project Coordinators for their valuable support and encouragement to take up and complete the thesis.My sincere thanks to all the faculties and staff at Vellore Institute of Technology, Chennai who helped me acquire the requisite knowledge. I would like to thank my parents for their support. It is indeed a pleasure to thank my friends who encouraged me to take up and complete this task.

Place: Chennai **Aaditya Bagaddeo**

Date: 04/04/2024

CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

Acronym	Expansion
AHRS	Attitude Heading and Reference System
API	Application Programming Interface
BFS	Breadth First Search
C-PRM	Coverage Probabilistic Road Map
CNN	Convolutional Neural Network
DARP	Drone Allocation and Routing Protocol
DFS	Depth First Search
DOF	Degrees of Freedom
FANET	Flying Ad-Hoc Network
GCS	Ground Control Station
GPS	Global Positioning System
IMU	Inertial Measurement Unit
LiDAR	Light Detection and Ranging
MADDPG	Multi-Agent Deep Deterministic Policy Gradient
MCPP	Multi-UAV Coverage Path Planning
MST	Minimum Spanning Tree
OGM	Occupancy Grid Map

RL	Reinforcement Learning
ROS	Robot Operating System
RRT	Rapidly Exploring Random Trees
SC-VRP	Stochastic Combinatorial Vehicle Routing Problem
SLAM	Simultaneous Localisation and Mapping
SOMA	Self Organizing Migrating Algorithm
SSA	Subsumption Architecture
UAV	Unmanned Aerial Vehicle
WFD	Wavefront Frontier Detector

CHAPTER 1

INTRODUCTION

1.1 GENERAL

In our ever-evolving technological landscape, the potential of unmanned aerial vehicles (UAVs), commonly known as drones, has surged to the forefront of innovation. Among their myriad applications, one particularly promising avenue is their utilization in exploring GPS-denied environments, such as indoor spaces where traditional satellite-based navigation systems prove ineffective. In response to this challenge, our project endeavors to implement a cutting-edge algorithm to orchestrate a swarm of drones for efficient exploration of indoor environments.The advent of drone swarm technology presents a paradigm shift in exploration methodologies. By leveraging the collective intelligence and coordination of multiple drones, we aim to overcome the inherent limitations of individual UAVs and navigate the intricate and often unpredictable confines of indoor spaces. This initiative holds immense potential for diverse domains ranging from search and rescue operations in disaster-stricken areas to inspection of hazardous industrial facilities, where human access may be restricted or perilous.

The background of the problem addressed by indoor mapping using a swarm of drones is rooted in the inefficiencies and challenges associated with traditional methods of surveying and inspecting complex indoor environments. Conventional approaches, often reliant on manual efforts or stationary sensors, struggle to comprehensively cover expansive or difficult-to-access areas. In response to this, industries recognize the pressing need for advanced, automated solutions. Challenges range from the intricate inspection of structures and equipment in industrial settings to optimizing architectural planning, responding to disasters, and streamlining operations in large warehouses. Additionally, the limitations of traditional methods have prompted a shift towards leveraging drone technology, equipped with precise sensors and mapping algorithms, for autonomous indoor mapping. This innovative approach not only addresses the shortcomings of existing methods but also empowers organizations with detailed, real-time data for analytics and decision-making, thereby enhancing overall operational efficiency across diverse sectors. Furthermore, it proves invaluable for conducting routine checks on old buildings and structures, where detailed inspections are essential but challenging to perform manually on a regular basis, contributing to the preservation and maintenance of architectural heritage and critical infrastructure.Since all these problems can have very distinct environments of exploration and deployment, simulating the working of the swarm in such environments is very helpful to understand the effect and efficiency of these systems in such environments.Beyond mere exploration, our endeavor seeks to integrate advanced sensors and imaging technologies into the drone swarm, empowering it to gather rich data about the environment. From capturing high-resolution images and generating 3D maps to detecting anomalies or hazards, the drones serve as versatile platforms for comprehensive situational awareness.

1.2 PROBLEM STATEMENT

Indoor environments lack a robust way of navigation given GPS data is very ineffective and inaccurate. Mapping of indoor environments requires precise navigation in unknown environments and thus , techniques like SLAM need to be implemented. Further, given the need for effective implementation of swarms of drones in such diverse indoor environments, it is necessary to have a way to evaluate the effectiveness and usefulness of the algorithm and check the amount of data that can be extracted. This asks for an accurate simulation system that can track the working of these algorithms as well as communication protocols to come up with accurate navigation and mapping in GPS denied environments .

1.3 RESEARCH CHALLENGES

Simulating the algorithm for drone swarm exploration in indoor environments using Robot Operating System (ROS) and AirSim poses several research challenges that demand meticulous attention and innovative solutions.

Firstly, achieving realistic environment modeling within the simulation is crucial. This involves accurately replicating the intricacies of indoor spaces, including varied layouts, obstacles, lighting conditions, and textures, to effectively test the algorithm's performance across diverse scenarios. Secondly, simulating sensors such as cameras, LiDAR, and IMUs within AirSim requires precise modeling to mimic real-world behavior while considering factors like noise, distortion, and occlusions. Additionally, integrating sensor data fusion techniques to optimize information extraction adds complexity to the simulation. Accurately modeling drone dynamics and control is essential for realistic behavior. This encompasses modeling aerodynamics, propulsion systems, and control algorithms to ensure drones respond realistically to inputs, maintain stability in varying conditions, and accurately track trajectories. Furthermore, developing communication protocols and coordination strategies for the drone swarm within ROS is crucial. This involves ensuring seamless communication among drones while minimizing latency and congestion, as well as implementing robust algorithms for task allocation, path planning, and swarm behavior coordination. As simulations scale up in size and complexity, addressing scalability and performance becomes paramount. Optimizing simulation performance to handle large-scale scenarios with numerous drones while maintaining real-time responsiveness requires efficient resource utilization and optimization of computational algorithms.

Further, validating the simulated algorithm's performance against real-world data and benchmarks is essential. This involves developing rigorous validation methodologies and benchmarking criteria to compare simulation results with empirical data collected from physical experiments, ensuring the algorithm's effectiveness and generalizability. Overcoming these challenges demands interdisciplinary collaboration across robotics, computer science, simulation, and control theory, leveraging innovative approaches and cutting-edge technologies to advance the development of drone swarm exploration algorithms within ROS and AirSim, ultimately facilitating safer and more efficient exploration of GPS-denied environments.

1.4 OBJECTIVES

This project aims to simulate the working of a swarm of drones to implement indoor exploration and mapping to depict its working in GPS denied environments. Airsim is a simulator in Unreal Engine which offers multiple APIs to spawn multirotor drones in a custom environment and allows python client based control of the same. Therefore , Airsim is a convenient way to simulate this implementation . However, since Airsim python client doesn't allow the implementation of multiple scripts simultaneously, the communication protocol and the working will be separately simulated using Robotics Operating System (ROS).

The Objectives can be listed as follows:

- Make a Master - Slave drone swarm such that the slave drones can follow the master drone's commands to split up and explore an environment

This includes coming up with the correct arrangement for the swarm such that the swarm can easily cover an environment . Given the absence of a GPS , it is impossible for drones to know where the other drones in the swarm are . Therefore, it is important that the drones communicate their location when there is a necessity for collaboration of the drones. Further, there is a need to use a local coordinate system which is doable thanks to the accurate values an Inertial Measurement Unit can provide.
This objective also includes using a simple Publisher - Subscriber model to establish communication between the drones . Since the communication cannot be simulated in a simple Airsim environment, the communication is separately simulated using a ROS based node system that shows the working of the

- Generate Point Cloud Data from Lidar to implement navigation

Airsim has APIs that allow easy recording of LiDAR as well as other sensors' data. This data can be visualized using python visualization tools as well as can be used to generate maps of the environment . However, the scope of the current implementation is limited to use the lidar data for navigation as mapping requires fusion of multiple 3D point clouds which is another problem that can be solved once the basic algorithm has been tested.

- Implement a simple 2D SLAM on a custom map to simulate the working of the algorithm using Pygame module

Pygame is a python library that is used to build games. This gives us the utility to be able to simulate a simple 2D environment and simulate the motion of our drones in the 2D environment . ROS is used to write separate scripts for every swarm drone and simulate the point of view of every drone separately as well as the communication happening between the drones. This can further be use to show how the drones separately record map data and how fusion of the data can help build a map of the environment.

1.5 SCOPE OF THE PROJECT

The project, titled "EXPLORATION AND MAPPING OF GPS DENIED ENVIRONMENTS USING DRONE SWARM," focuses on addressing the challenges of unmanned aerial vehicle (UAV) navigation within indoor spaces where traditional GPS signals are unreliable or unavailable. This problem is particularly significant in scenarios such as search and rescue missions, surveillance operations, and warehouse logistics, where drones need to navigate accurately despite the absence of GPS guidance.

A pivotal aspect of the project involves the creation of a 3D simulation environment using AirSim, a powerful simulator for drones. Within this simulated environment, the project aims to demonstrate the functionality of the navigation algorithm developed specifically for GPS-denied indoor environments. This simulation will showcase how drone swarms interact with their surroundings, navigate obstacles, and accomplish predefined tasks autonomously, relying on alternative sensor data and advanced algorithms for localization and mapping.

In addition to the 3D simulation, the project includes the development of a 2D simulation environment using Pygame and ROS (Robot Operating System). This simulation will focus on emulating the communication protocols and mapping algorithms employed by the drone swarm. By simulating the exchange of information among drones and their collaborative efforts in mapping their environment, this aspect of the project aims to provide insights into the underlying mechanisms that enable effective navigation in GPS-denied indoor environments.

Furthermore, the project involves recording point clouds generated by sensors such as LiDAR or depth cameras mounted on the drones during simulated flights. These point clouds represent the drones perception of the environment and provide valuable data for navigation and obstacle avoidance. The recorded point clouds will be processed and visualized to illustrate how drones utilize sensor data to build maps of their surroundings and make navigation decisions in real-time.

Overall, by integrating these components, the project seeks to offer a comprehensive understanding of the challenges associated with navigating drone swarms in GPS-denied indoor environments. Furthermore, the simulations developed as part of this project will serve as essential tools for testing and refining navigation algorithms before deployment in real-world scenarios, ultimately contributing to the advancement of autonomous drone technologies and their applications in indoor environments.

CHAPTER 2

SURVEY ON NAVIGATION AND DRONE SWARMS

The current trends in the field of UAV navigation include very heavy utilization of Global Coordinates . Further , they depend a lot on pre-existing maps , especially in indoor environments. Further , drone swarm based explorations are predominantly dependent on global coordinates as well as ground control stations controlling the drones movements to a certain extent. Following is the current research that has been marking the advancements in Drone swarm navigation as well as mapping and exploration .

Setup and configuration of an autonomous UAV swarm for indoor coverage path planning [1] proposed an algorithm to utilize cellular decomposition and coverage path planning techniques to optimize the exploration of GPS-denied indoor environments by drone swarms. The process begins with the construction of a grid map representing the space, which is then decomposed into cells. These cells are linked to form a graph, enabling the application of traversal algorithms for coverage optimization.The Multi-UAV Coverage Path Planning (mCPP) algorithm, described in the referenced paper, leverages a full map of the environment to coordinate multiple drones efficiently. By employing cellular decomposition and depth-first search (DFS) to construct a minimum spanning tree (MST), the algorithm divides cells into sub-cells and orchestrates a counterclockwise trajectory around the MST for comprehensive coverage. This approach ensures efficient area coverage and energy usage, with uniform distribution of energy consumption among drones.

Additionally, the proposed algorithm DARP (Drone Allocation and Routing Protocol) allocates specific areas for each drone based on the proximity to neighboring drones within the grid. By considering the nearest drone for each cell, DARP optimizes coverage and minimizes redundant exploration. This method effectively utilizes drone resources while maintaining uniform energy consumption across the swarm.

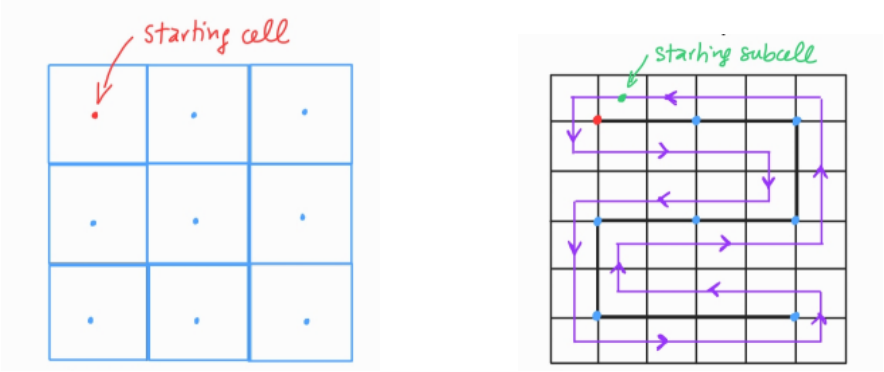


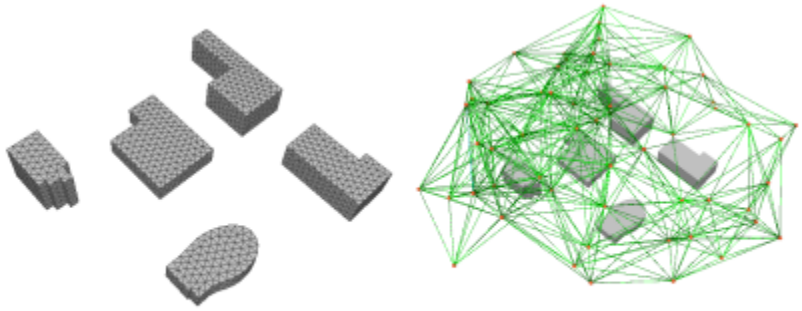
Figure 1: Using DARP to segregate drone target areas

Overall, the combination of cellular decomposition, traversal algorithms, and dynamic allocation strategies enhances the efficiency and effectiveness of drone

swarm exploration in GPS-denied indoor environments. These approaches enable comprehensive coverage while optimizing energy usage and ensuring equitable distribution of tasks among drones, thus advancing the capabilities of autonomous drone technologies in complex scenarios.

Multi-UAV Coverage Path Planning for the Inspection of Large and Complex Structures[2] further proposed an inspection path planning framework that addresses the Multi-Agent Coverage Path Planning (MACPP) problem, specifically applied to 3D visual inspection tasks. The framework begins by generating waypoints and path primitives through incremental sampling, creating a C-PRM (Coverage Probabilistic Roadmap) graph that encapsulates information on topology, coverage, and path length. The MACPP problem is then formulated as a min-max Stochastic Combinatorial Vehicle Routing Problem (SC-VRP) and solved using Biased Random-Key Genetic Algorithm (BRKGA).

For Coverage Sampling with Path-Primitives, the framework utilizes a Coverage Sampling Problem (CSP) approach to generate and select via-points and path-primitives to ensure thorough coverage of the target structures. A path-primitive sampling method is employed, alongside a dual sampling technique to enhance efficiency and coverage. Binary dilation on voxels is utilized to create an efficient sampling space. The dual path-primitive sampling algorithm iteratively samples via-points and path-primitives with a bias towards unseen surface patches, improving sampling efficiency. Visibility evaluation is conducted through ray-tracing, assessing the visibility of viewpoints and path-primitives. Collision-free local planning is employed to find collision-free local paths, while DualSampleVP biases via-point sampling towards uncovered surface patches.



(a) Target structure (b) Target structure with C-PRM

Figure 2 : Probabilistic Road Map around the target with MCPP

Overall, the framework employs incremental sampling and visibility evaluation techniques to generate waypoints and path primitives, ensuring comprehensive coverage of target structures in 3D visual inspection tasks. The utilization of CSP, dual sampling methods, and collision-free local planning enhances sampling efficiency and

path planning accuracy, contributing to effective inspection path planning in complex environments.

Mapping of unknown environments with an autonomous drone swarm[3] shows a simplified version of Hector SLAM. Hector SLAM is unique in that it doesn't require odometry and treats the robot as a 6 Degree of Freedom (6DOF) rigid body. This method considers both translational and rotational motions, leading to more accurate 3D maps. Hector SLAM utilizes Occupancy Grid Mapping, representing the map as a grid of cells with logarithmic probabilities of being occupied, unexplored, or free. During scan matching, new scans are matched to previous scans iteratively, estimating pose changes and updating the map accordingly. Multi-resolution maps using different grid sizes are generated to aid in path planning calculations. Implementation of this SLAM method was done using the Robot Operating System (ROS) package, hector_mapping, which condensed the calculations into relevant functions. The algorithm was simplified and ported to Arduino for use as a programming library. The drone utilized one IMU and six ToF (Time-of-Flight) sensors for mapping. ToF sensors provided depth-based information, while the IMU controlled drone movement to maintain a constant angular velocity.

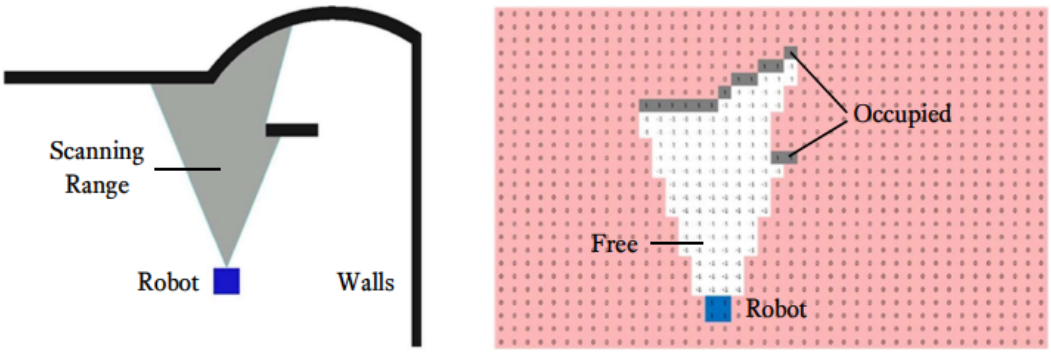


Figure 3 : Frontier based Hector SLAM

The solution design for implementing the drone swarm system was optimized to ensure simplicity, scalability, and independence of operation. Assembly of the drones was straightforward due to standardized design, and sensor calibration and movement calibration were quick processes. The RF module facilitated communication between drones, supporting connections with up to six other modules and enabling tree topology networks with a theoretical maximum of 3125 modules. Coordination within the swarm system was streamlined by assuming drones were initially placed close to each other. The first drone's position served as the origin of the map, and subsequent drones were activated sequentially. Each drone compared its initial scan with the first drone's to determine its position, and this data was added to the map. Drones then autonomously navigated to the closest frontier outside a certain radius of the first drone, avoiding collisions by staying clear of other drones.

This implementation maintained each drone's independence of operation, as it did not rely directly on other drones for function. This design ensured ease of scalability, allowing for the addition or removal of drones as needed, and guaranteed completion

of the mapping task even if certain drones malfunctioned. Overall, the solution design prioritized simplicity, scalability, and robustness in the implementation of the drone swarm system for mapping tasks.UAV swarm communication and control architectures[4] introduces and compares two prominent architectures for coordinating communication within UAV swarms: infrastructure-based swarm architecture and Flying Ad-Hoc Network (FANET) architecture.

The infrastructure-based swarm architecture relies on a Ground Control Station (GCS) for telemetry reception and command transmission to individual drones. It operates semi-autonomously, depending on centralized control. Advantages include real-time optimization by the GCS's high-performance computer, eliminating the need for direct drone networking, and reduced payload requirements. However, drawbacks include GCS dependency, lack of system redundancy, and the requirement for UAVs to stay within GCS propagation range, posing susceptibility to interference. FANET architecture involves UAVs communicating autonomously without an access point, finding applications in military, civil, wildfire management, and disaster monitoring. Challenges include size, weight, power considerations, communication distance limitations, and dynamic reconfiguration issues leading to packet loss. The proposed hybrid architecture integrates ad-hoc elements with infrastructure support, utilizing cellular networks for UAV-to-UAV communication. Decisions are distributed among UAVs, ensuring high autonomy. Strengths include unlimited communication range with cellular coverage, redundancy, reliability, 5G advancements, and lightweight hardware. In summary, the hybrid architecture combines infrastructure-based and ad-hoc principles, leveraging cellular networks for communication while enabling distributed decision-making and UAV autonomy.

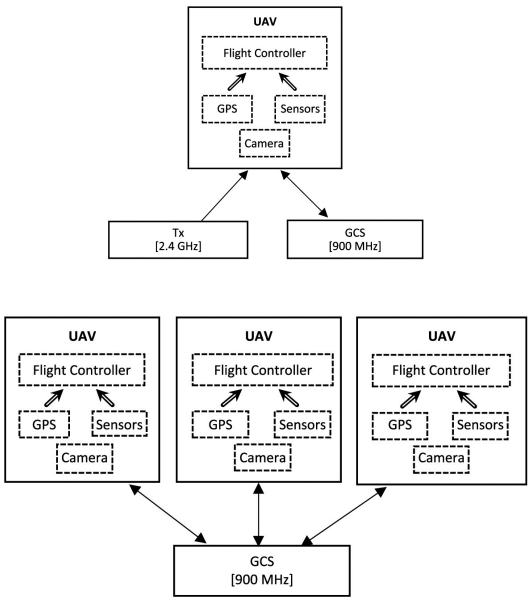


Fig. 4. Communication architecture of UAV swarms based on FANET.

Autonomous Drone Swarm Navigation and Multi-target Tracking in 3D Environments with Dynamic Obstacles [5] discusses the application of deep reinforcement learning and swarm intelligence in drone swarm navigation and target tracking tasks. It highlights the importance and versatility of swarm intelligence across various disciplines, including telecommunications, surveillance, and search and rescue missions. The research aims to address the challenges of developing an end-to-end model for detecting targets in complex environments and autonomously navigating drone swarms towards them while avoiding obstacles and maintaining stable formations. Key contributions of the work include proposing a policy-based deep reinforcement learning strategy for swarm navigation, creating realistic 3D environments with dynamic obstacles, introducing mechanisms for converting sensory input to high-level commands and incorporating memory to aid swarm navigation. Novel reward functions are introduced to enable obstacle avoidance, target identification, and efficient path traversal while maintaining stable swarm structures. The research also introduces multi-target tracking capabilities and the concept of multi-swarms, where a swarm can divide into sub-swarms to track multiple distant targets simultaneously.

The proposed framework models five key swarm behaviors: swarm formation and organization, dynamic obstacle avoidance, target detection, navigation towards the target while sustaining swarm formation, and tracking multiple targets by dividing the swarm into sub-swarms. The research utilizes Unity 3D for training swarm agents, aiming to simulate real-world scenarios in complex environments. Overall, the research addresses the need for optimizing swarm navigation and target tracking in unseen environments, contributing to advancements in autonomous drone technologies. The research aims to address challenges in artificial swarm systems, focusing on end-to-end models for detecting targets, autonomous navigation, and obstacle avoidance. The contributions of this work include a policy-based deep reinforcement learning strategy for autonomous drone swarm navigation in complex 3D environments with dynamic obstacles. The introduced mechanisms involve converting sensory input to high-level commands, incorporating memory for path recall, and introducing novel reward functions for barrier avoidance and target identification.

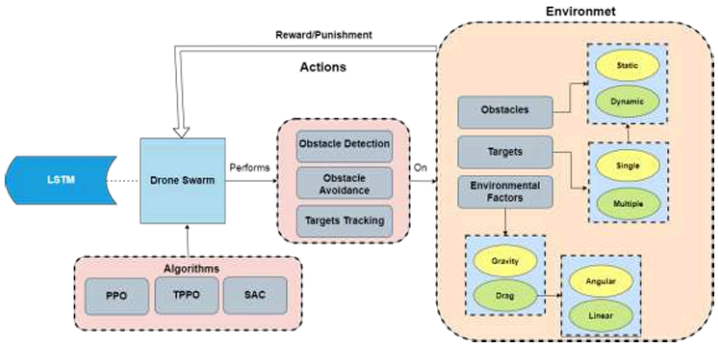


Figure 5 : Proposed Architecture for RL Model

A Multi Agent Reinforcement Learning Method for swarm robots for collaborative space exploration [6] covers the implementation of a multi-agent Actor-Critic reinforcement learning model and the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) algorithm on a self-exploring autonomous swarm of bots designed for space exploration. It represents a significant advancement in robotics and space technology. This innovative approach enables the swarm of bots to navigate and explore unknown environments autonomously while optimizing action sequences for efficient exploration. The utilization of a multi-agent Actor-Critic reinforcement learning model allows each bot in the swarm to learn and adapt its behavior based on feedback received from the environment. By leveraging reinforcement learning techniques, the bots can explore and navigate through complex and unfamiliar terrain, continuously refining their strategies to achieve optimal performance. This adaptive learning capability is crucial for space exploration missions where environments are often unpredictable and dynamic.

Furthermore, the implementation of the MADDPG algorithm enhances swarm collaboration by enabling the bots to coordinate their actions effectively. By optimizing action sequences through deep deterministic policy gradients, the swarm can execute synchronized maneuvers to overcome obstacles, avoid collisions, and efficiently explore vast areas of space. This collaborative approach maximizes the exploration capabilities of the swarm, allowing it to cover more ground and gather valuable data for scientific research and discovery.One of the key advantages of this approach is its energy-efficient navigation system. By leveraging intelligent algorithms and collaborative strategies, the swarm can conserve energy while exploring distant and challenging environments. This is particularly important for space exploration missions where energy resources are limited and must be carefully managed to ensure the success of the mission. Overall, the implementation of multi-agent reinforcement learning models and advanced optimization algorithms represents a significant advancement in autonomous swarm technology for space exploration. By enabling autonomous and efficient exploration of unknown environments, this technology has the potential to revolutionize our understanding of the universe and unlock new frontiers in space exploration.

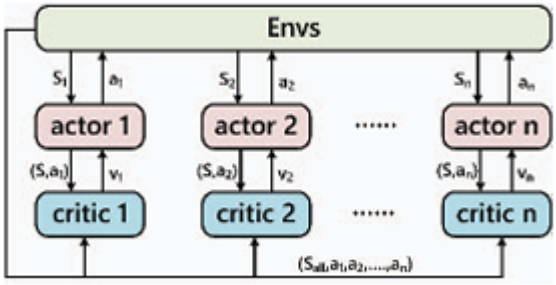


Figure 6 : Multi-Agent Actor Critic Architecture

Consciousness [7] aims to facilitate autonomous swarm navigation by integrating navigation, object detection, coordinate calculation, and adaptive autonomous modes using translational coordinates. The top-level algorithm consists of two feedback-based modules: one managing followers and the other managing adaptive autonomous modes with collision avoidance capabilities. In the Follower module, drones receive coordinates from their leader and obstacles, transmitted by the leader. They perform calculations to translate these coordinates based on their own positions. If there is no feedback from the leader within a defined timeout, indicating potential communication loss, the drone temporarily assumes a leadership role as a fail-safe measure, resuming navigation by activating its sensors.

The algorithm in the paper presents a detailed procedure for the navigation and object detection tasks performed by the leader agent within a swarm. Operating within an infinite loop, the leader continuously monitors its surroundings for obstacles. Upon detecting an obstacle, it calculates the distance and angle to accurately assess the threat. Subsequently, the leader communicates relevant information, including obstacle data, to each follower agent within the swarm. This communication facilitates coordinated movement and ensures that all agents are aware of potential obstacles. Additionally, the algorithm incorporates a cross-checking mechanism to verify the accuracy of received data at the follower's end. If significant disparities are detected, indicating potential errors, the affected follower is flagged as dynamic, prompting the need for adjustment. Furthermore, collision avoidance procedures are executed to prevent potential collisions among agents.

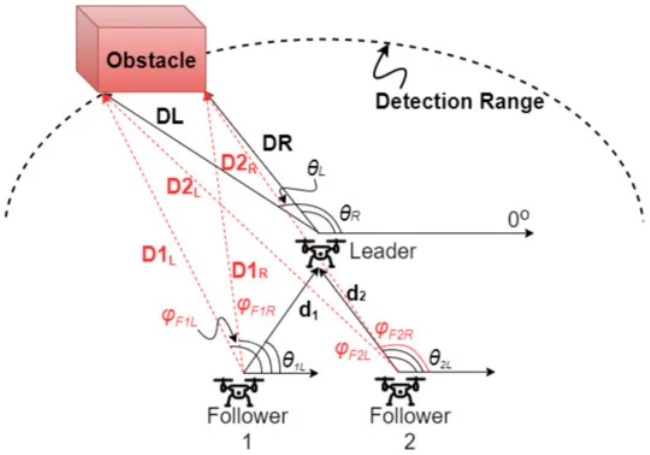


Figure 7 : Swarm moves together based on angle and distance measurements

Frontier Based Exploration for Autonomous Robot[8] proposes the WFD

algorithm, an enhanced version of the original frontier-based exploration method, utilizes two nested Breadth-First Searches (BFS) to improve efficiency. It selectively scans known regions of the occupancy grid, reducing time complexity by avoiding scanning the entire grid in each algorithm run. The algorithm's pseudocode involves initializing a queue data structure (queuem) to find frontier points in the occupancy grid. Points are enqueued and dequeued, marked as Map-Open-List and Map-Close-List, respectively. The inclusion of points with at least one open-space neighbor in queue is constrained, ensuring selective scanning of known regions.

The algorithm begins by enqueuing the robot's current pose into queuem and executing a BFS until encountering a frontier point. A new queue (queuef) is then initialized to extract the frontier of this point. The inner BFS marks points as Frontier-Open-List, and frontier points found are stored in a list marked as Frontier-Close-List. A frontier point is added to the frontier list only if it is not part of Frontier-Open-List, Frontier-Close-List, or Map-Close-List, guaranteeing unique assignments in each algorithm run.

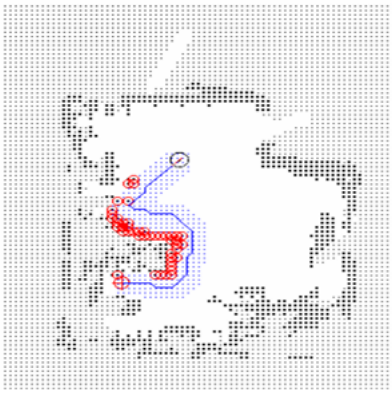


Figure 8: Frontier based path planning and updating the occupancy grid

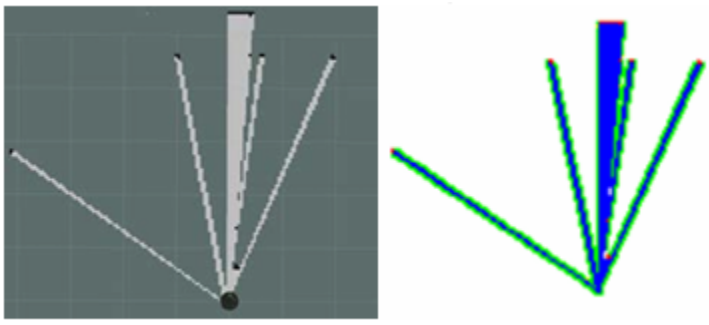


Figure 9: Detection of frontier points at every iteration

Environments [9] proposes the Subsumption Architecture (SSA), which is applied to model the behavior of swarm robots, specifically addressing target detection problems. The SSA consists of three main layers: transmission, obstacle avoidance, and target exploration, with an additional layer for outdoor environments. Each layer comprises interconnected modules that contribute to the overall behavior of the swarm robots. In the transmission layer, the robot detects targets, triggering the transmission of messages to the base station and stopping the robot's motors when sensor inputs exceed a threshold. The obstacle avoidance layer directs the robot to turn right or left based on sensory inputs, aiding in avoiding obstacles.

This SSA implementation allows for diverse and adaptive movements in swarm robots, facilitating target detection in both indoor and outdoor environments. The incremental evolution of SSA enables the design of collective behavior, demonstrating flexibility and effectiveness in various settings.

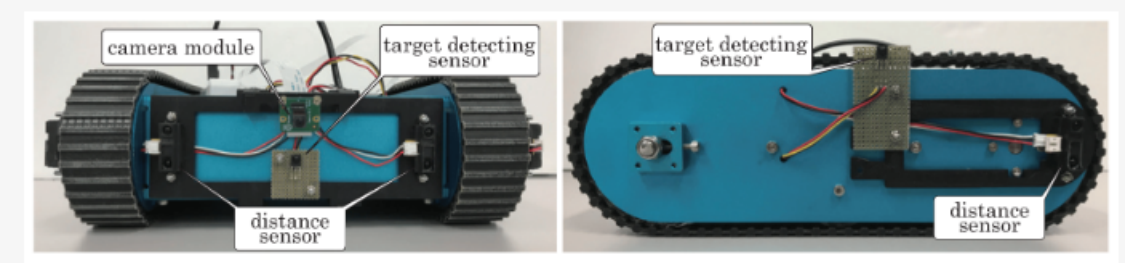


Figure 10: Swarm Robot Setup

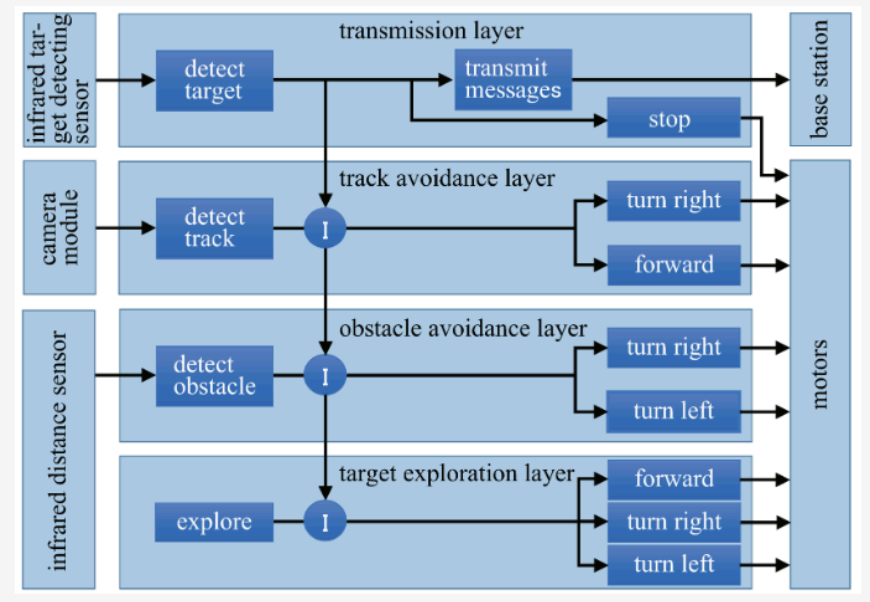


Figure 11: Layered SSA Architecture

Obstacle Avoidance for Swarm Robot Based on Self-Organizing Migrating Algorithm [10] presents an obstacle avoidance algorithm for swarm robot in unknown environment based on self-organizing migrating algorithm in which the trajectory is

divided into a set of points created by the SOMA that robot must pass through. During the move, obstacles will be detected by sensors on the robot, it only knows the target position without knowing the obstacles location until detecting them. The fitness function for a robot navigating towards a target while avoiding obstacles consists of two components: an attractive element towards the target and a repulsive element from obstacles. The number of obstacles.

The influential coefficient c determines how far the robot will avoid obstacles and navigate through gaps between obstacles. This coefficient is adjusted dynamically, especially in cases where the robot is trapped between obstacles, to prevent local minima. The change in the fitness function is depicted through contour figures. Initially, when obstacles are detected, the repulsive force component is added to the fitness function. If the gaps between obstacles are too small for the robot to navigate through, the influential coefficient is reduced, leading to the robot being trapped between obstacles. However, by further reducing c, the robot can escape from the hold and move away from the obstacles. The fitness function is based on the principle of attraction from the target and the repulsion of the obstacles, which helps robot find the trajectory for moving, and the ways leaving it to move safety away from the trapped area.

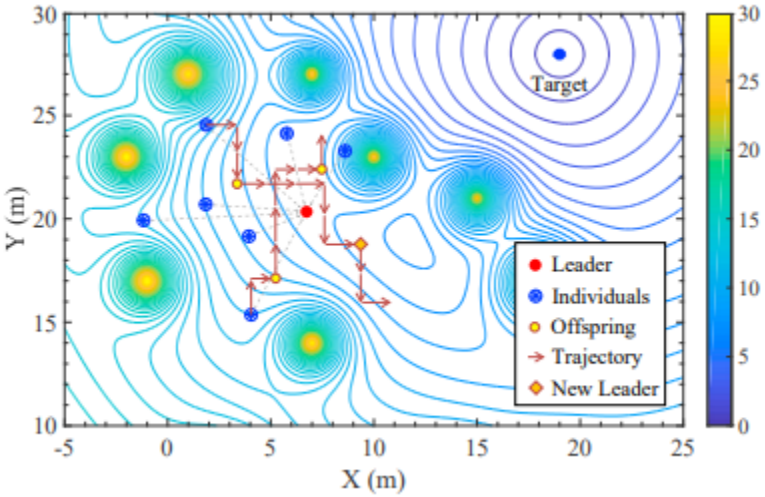


Figure 12 : The Principle of SOMA

CHAPTER 3

IMPLEMENTATION

3.1 GENERAL

The implementation in AirSim for the navigation of drone swarms in GPS-denied indoor environments involves several key components. Airsim is used to set up a simulation in a custom indoor environment. Firstly, the setup is defined in a settings.json file, configuring the initial arrangement of drones along with the sensors and cameras mounted on them. This setup is crucial as it establishes the baseline for the simulation environment.The core of the implementation lies within Python scripts utilizing AirSim APIs. These scripts orchestrate the synchronized movement of drones, collect data from sensors such as LiDAR and cameras, and implement decision-making logic based on this data. One notable feature of the algorithm is the integration of a simple OpenCV-based door detection mechanism. When a door is detected within the drone's field of view, it triggers a response wherein the drone halts its movement. This pause allows for further exploration beyond the door. Upon detecting a door, the master drone initializes the separation of slave drones to explore the area behind it. Each slave drone is tasked with a specific exploration mission. This collaborative approach enables efficient exploration of the indoor environment. While the slave drones carry out their tasks, the master drone waits for their completion. Once all tasks are fulfilled and the slave drones return, they rejoin the master drone. This synchronization ensures cohesive teamwork among the drone swarm.

Further, a pygame and ros setup is used to simulate communication and mapping. The Pygame setup creates a simplified 2D grid environment, defining the positions of obstacles and drones. This environment provides a visual representation of the indoor space, allowing for easy visualization of drone movements and interactions with obstacles. ROS plays a pivotal role in maintaining the behavior of the drone swarm. A master drone node is responsible for simulating the entire swarm and making decisions such as when slave drones should separate for exploration. Each slave drone node subscribes to the master node, receiving commands to separate and explore specific areas.When a slave drone receives the command to separate, it opens a separate Pygame window showing its perspective of the environment. This visual representation allows for real-time monitoring of the exploration process. During exploration, each slave drone autonomously navigates its assigned area, backtracking upon completion to close the loop. Once done, it waits for the master drone to send its local coordinates. Based on this information, the slave drone calculates the optimal path to return to the master drone's location.Upon reaching the master drone, the slave drones publish map data to a mapping drone node. This mapping drone aggregates the received map data from all slave drones to generate a fused map of the environment. This fused map provides a comprehensive representation of the indoor space, aiding in further navigation and decision-making processes .

Following the completion of tasks and reassembly of the swarm, the algorithm seamlessly transitions to the next phase, repeating the exploration process. This

iterative approach allows for thorough coverage of the indoor environment while leveraging the collective capabilities of the drone swarm.

In summary, the algorithm implemented in AirSim orchestrates the synchronized movement of drone swarms, integrates sensor data for decision-making, and employs collaborative exploration strategies enabled by simple door detection mechanisms. This approach facilitates efficient navigation in GPS-denied indoor environments, showcasing the potential of autonomous drone technologies in complex scenarios. Further the Pygame and ROS implementation facilitates communication and coordination among drone swarm nodes, enabling collaborative exploration and mapping of GPS-denied indoor environments. This approach demonstrates the potential of autonomous drone technologies in complex scenarios and lays the foundation for future advancements in indoor navigation systems.

3.2 METHODOLOGY

The project involves simulating drone object modules within a simulated environment, utilizing hardware components such as the Quadrotor Drone, Lidar Sensor for obstacle avoidance and mapping, and Camera for image processing and perception. Various software modules are employed including Airsim, Unreal Engine, PyGame, Robot Operating System (ROS), Tensorflow, Keras, and OpenCV. Algorithms such as image segmentation, IMU-based motion tracking, LiDAR SLAM, and convolutional neural networks (CNN) for image classification are utilized for drone navigation and perception tasks. The protocol for implementation includes creating a world in Unreal Engine, establishing a database for door locations, defining agents and sensors for drones, setting up visual odometry-based path planning for drones, and integrating algorithms for data processing and analysis. Lidar and IMU data are collected and graphed to assess the effectiveness of the implemented algorithms and sensor fusion techniques in navigation and perception tasks within the simulated environment.

The proposed exploration algorithm is designed to efficiently map multiple floors of a building using a coordinated swarm of drones, with a master drone orchestrating the exploration process. Each floor is assigned a set of drones, and the master drone receives instructions on which floor to explore next. To ensure precise navigation and mapping, the algorithm leverages an Attitude and Heading Reference System (AHRS) to monitor the position and orientation of the drones as they move through the floor space.

As the drones explore, they autonomously navigate through doors and unexplored areas using advanced algorithms. The master drone dynamically adjusts the formation of the swarm based on the environment and the progress of exploration. This flexibility allows for optimal coverage of the floor space while adapting to obstacles and changing conditions.A crucial aspect of the algorithm is the monitoring of battery levels and safety considerations. If the battery levels of any drone drop below a predefined threshold or if the distance from the starting point becomes unsafe, the swarm returns to a designated charging station for recharging. This proactive approach ensures uninterrupted exploration while preserving the safety of the drones.

The slave drones in the swarm are equipped with SLAM (Simultaneous Localization and Mapping) techniques, enabling them to efficiently explore doors and unexplored areas. This capability enhances the mapping accuracy and completeness of the floor space, providing valuable insights for various applications such as search and rescue operations, building inspection, or environmental monitoring. In summary, the proposed exploration algorithm represents a comprehensive approach to mapping multiple floors of a building using a coordinated swarm of drones. By combining advanced navigation techniques, dynamic swarm coordination, and intelligent battery management, the algorithm enables efficient and thorough exploration of complex indoor environments.

3.3 ALGORITHM

Since the swarm follows a master-slave architecture , the algorithm can be divided into two parts, one that applies to the master drone and one that applies to the slave drones.

Swarm Arrangement :

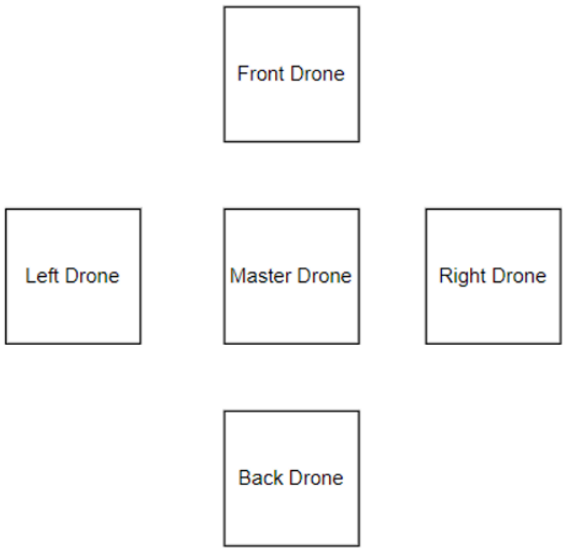


Fig 13 : Arrangement of Drones in the swarm

The **Master Drone** algorithm orchestrates the systematic exploration of multiple floors within a building, treating each floor as a separate case. It organizes a swarm of five drones, with a master drone at the center and four slave drones aligned around it in a plus pattern. Each set of drones is assigned to explore a specific floor, with the floor number provided to the master drone to determine the destination. Upon deployment on each floor, the drones move forward while monitoring their position in three-dimensional space using inertial measurement unit (IMU) data. As they progress, the drones autonomously detect doors or unexplored areas using a combination of sensors such as lidar, and if detected, a slave drone separates from the swarm to investigate further.

The algorithm prioritizes safety and battery management. If all slave drones

maintain battery levels above 30% and are at a safe distance from the starting point, they continue exploration. However, if a slave drone's battery falls below 30% or if it reaches a critical battery level (e.g., below 40%) and is in an unsafe location, the swarm prioritizes safety and returns to the starting point for charging. In such scenarios, if other drones have sufficient battery and are in a safe position, they may continue exploration. However, if multiple drones face battery issues or are in unsafe locations, the entire swarm returns for recharging.

Advantages of the Master Drone algorithm include its systematic approach to floor exploration, efficient use of resources through swarm coordination, and prioritization of safety and battery management. By treating each floor as a separate case and dynamically adjusting the swarm formation based on environmental conditions, the algorithm ensures thorough exploration while minimizing risks to the drones. Additionally, the ability to backtrack or trace previous paths enhances efficiency and coverage, especially in complex indoor environments where navigation may be challenging.

Algorithm _Master drone

- split cases - (consider floors separate, work on each floor separate)
- 5 drones in a swarm - every set of drones will have a master at the center with the other four aligned in a plus pattern around the master
- set of 5 drones assigned for each floor
- floor number fed to master drone (say 5 floors in a building, then 5 sets of swarms , each master will be told which floor to reach(so basically how many stairs to climb)
- on each floor - LOOP
  - deploy a set of 5 drones - center one being master
  - enter and move straight (monitor x, y and z using IMU)
  - every time a door is detected / lidar range doesn't detect a wall, separate one drone towards that side
  - if (all slave drones battery > 30 %) and dist_from_spawn == safe:
    - if no of drones in swarm == 1 : stop
      - `if number ≠ 5 and number == 2 : backtrack`
    - when number == 5
    - trace the previous path till last checkpoint
  - if any one drone battery < 30% or (battery < 40% and dist_from_spawn == unsafe):
    - if battery of all others > 60 % and dist_from_start == safe:
      - continue()
    - else another drone battery < 60% or and (dist_from_spawn == unsafe):
      - whole swarm return back for charging

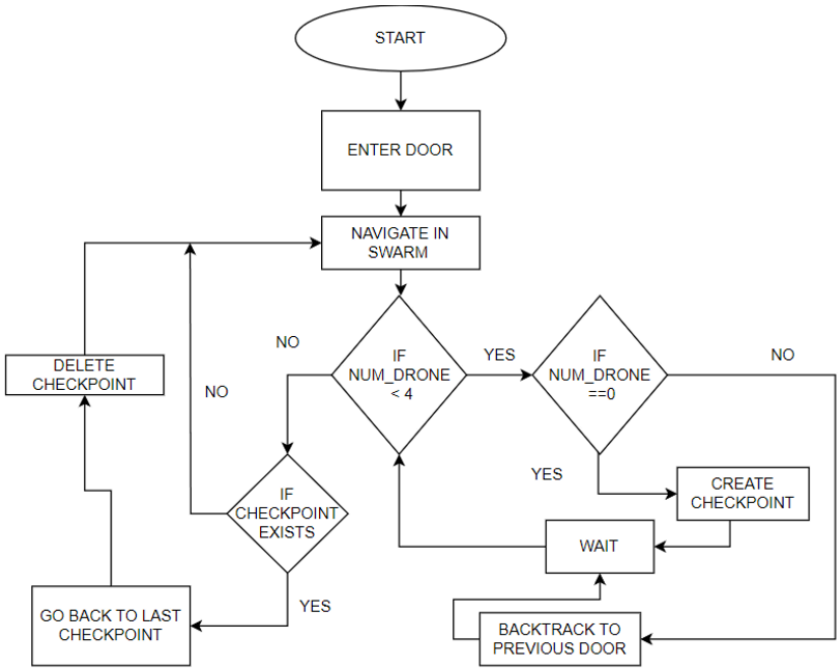


Figure 14: Flowchart for Master Dorne Algorithm

The **Slave Drone** algorithm focuses on the exploration and mapping of doors and unexplored areas detected by the master drone. It operates within the swarm under the guidance of the master drone, ensuring coordinated exploration while maximizing coverage. When the drone's battery level is above 30% and it is in a safe position, it autonomously separates from the master drone and moves towards detected doors or unexplored areas. Upon reaching a door, the drone enters and explores the area, using visual slam techniques to map the surroundings and identify potential paths for further exploration.

The algorithm enables the drone to explore multiple doors sequentially, with a limit set on the number of temporary doors to prevent exhaustive exploration. After exploring the designated areas, the drone returns to the door and awaits further instructions from the master drone. However, if the drone's battery level falls below 30% or it enters an unsafe location, it prioritizes safety and returns to the starting point for recharging.

Advantages of the Slave Drone algorithm include its ability to autonomously explore doors and unexplored areas, efficiently mapping the environment while coordinating with other drones in the swarm. By employing visual slam techniques, the drone can navigate through complex indoor spaces and contribute to comprehensive floor mapping. Additionally, the algorithm's focus on safety and battery management ensures the drone's longevity and reliability during exploration missions.

Algorithm _Slave Drone

- If drone battery > 30 % and dist_from_spawn == safe:
  - if door detected / lidar didn't detect wall , separate from master and move towards door/ unexplored area
  - function EXPLORE()
  - `while temp_door_count ≤ 3 :`
  - temporary_door_count +=1
  - enter the door
  - if another door detected
  - call Explore()
  - else
  - perform Visual Slam
  - Return to door
  - wait for master (land / hover based on how far master drone has moved)
- else if drone battery < 30 % or (battery < 40% and dist_from_spawn == unsafe):
  - return to start for charging

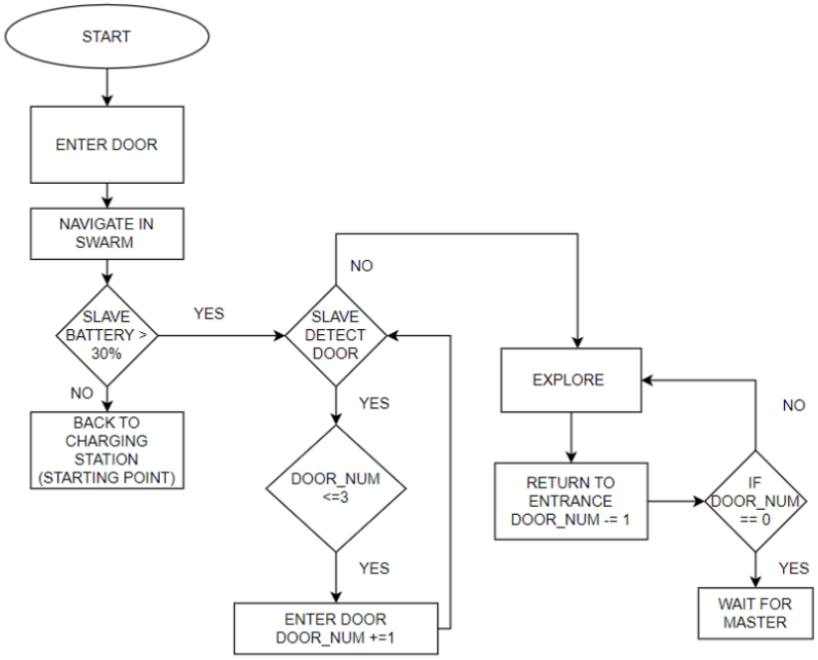


Figure 15: Flowchart for Slave Dorne Algorithm

Finally , the **Door Detection** Algorithm can be seen as follows -

Door Detection Algorithm (specific to door environment in Airsim)

- segment the image to obtain a binary image
- for every row of pixels in the image :

- num → no. of rows that belong to door
- k = no. of pixels that are of pixel intensity 0 (black)
- if k % 4 == 0 (K is a multiple of 4)
    - the row belongs to door
    - num += 1
- if num > threshhold :
    - Door Detected

- If Door Detected :
- Centre Point = average of all points in rows belonging to door
- if centre point = width of image / 2 (centre of image laterally)
    - the door is in the centre of the image , i.e drone can now seperate from the swarm

3.4 COMMUNICATION

The communication within the drone swarm utilizes a Publisher-Subscriber model, facilitated by the Robot Operating System (ROS), to exchange information between drones. ROS provides a robust framework for developing and simulating robotic systems, offering a communication infrastructure that allows nodes to publish messages on specific topics, which other nodes can then subscribe to and receive.

In this model, each drone acts as both a publisher and a subscriber. The master drone publishes relevant information, such as floor assignments, exploration progress, and safety alerts, on designated topics. Meanwhile, the slave drones subscribe to these topics to receive instructions and updates from the master drone.ROS enables seamless communication between drones in the swarm, allowing for real-time coordination and synchronization of actions. For example, when the master drone detects a door or an unexplored area, it publishes a message indicating the location and instructs nearby slave drones to investigate. The slave drones, subscribed to this topic, receive the instructions and autonomously navigate towards the designated areas for exploration.Moreover, ROS facilitates the exchange of feedback and status updates between drones, enabling them to adapt their behavior dynamically based on changing conditions. For instance, if a slave drone encounters an obstacle or experiences a battery issue, it can publish a message indicating the situation. Other drones in the swarm, subscribed to this topic, can receive the update and adjust their actions accordingly, ensuring collective safety and efficiency.

To simulate the communication between drones in the current scenario , there is no better tool than ROS . ROS allows definition of publisher and subscriber nodes and allows them to send data to respective nodes through a rosmaster .

Communication Sequence -

- Swarm will navigate through the environment until a door is detected
- based on which side the door is detected , the master drone will send signal to respective side slave drone to separate from the swarm and being exploration towards that door
- It also publishes the current local coordinate of the drone i.e the point of separation
- All slave drones are in subscribing mode till they receive the command to separate
- When all drones receive the separation command, the master drone will stop and publish its current location to all the slave drones
- Once the slave drone is done exploring, it will look for the master drone’s published data about its current position
- Based on the detaching position and current position of the master , the slave drone will move towards the master’s location by calculating the position

difference and the kind of motions to make to reach the master

- Once all four slave drones are back in swarm, they send their collected obstacle data to a separate Mapping node that will fuse these 2D map matrices to complete mapping of that area of the environment

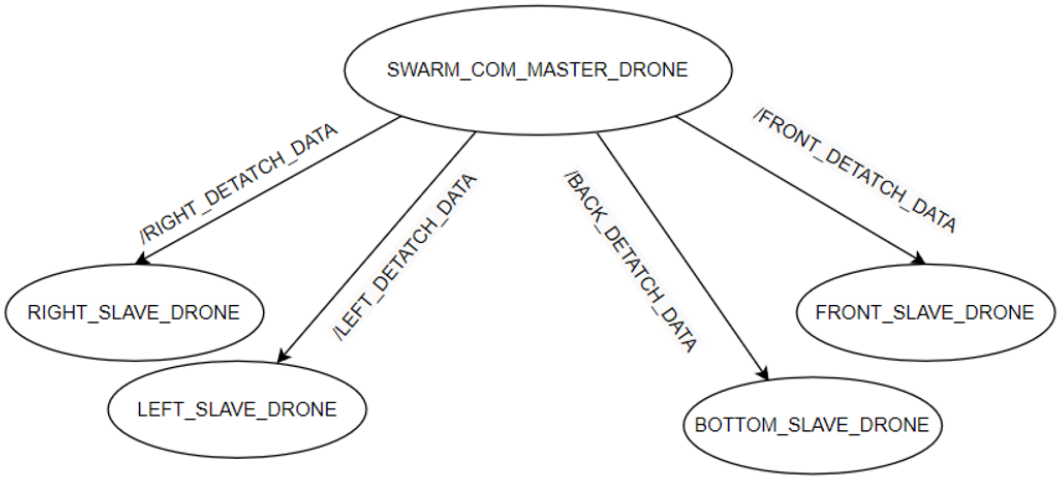


Figure 16 : RQT Graph depicting nodes and topics active while navigation is taking place

3.5 SIMULATION ENVIRONMENT

3.5.1 AIRSIM

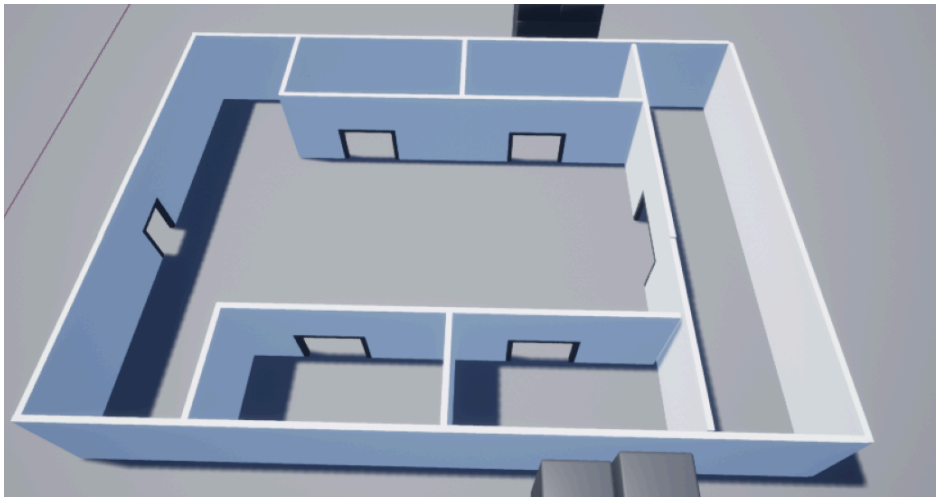


Figure 17 : Top View of Indoor Simulation Environment setup in Airsim

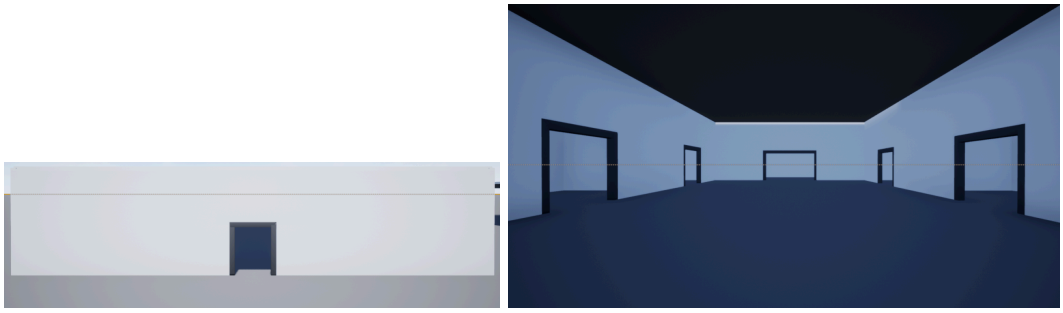


Figure 18 : Front View of Environment from outside and inside

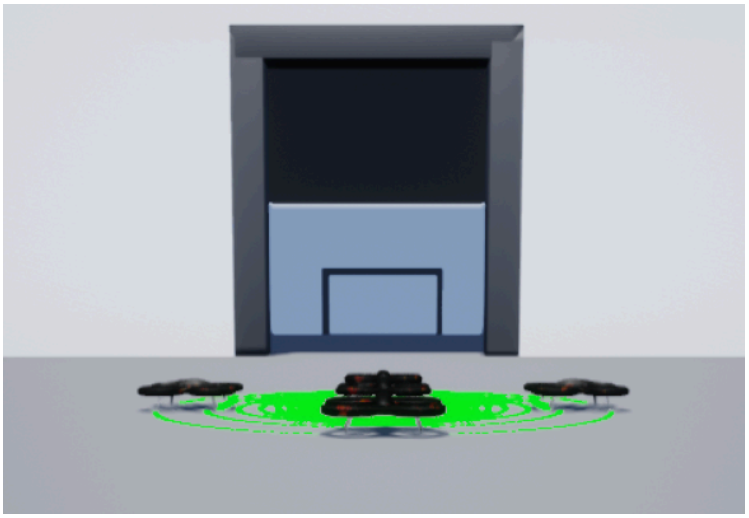


Figure 19: Swarm Point of View when Starting Simulation

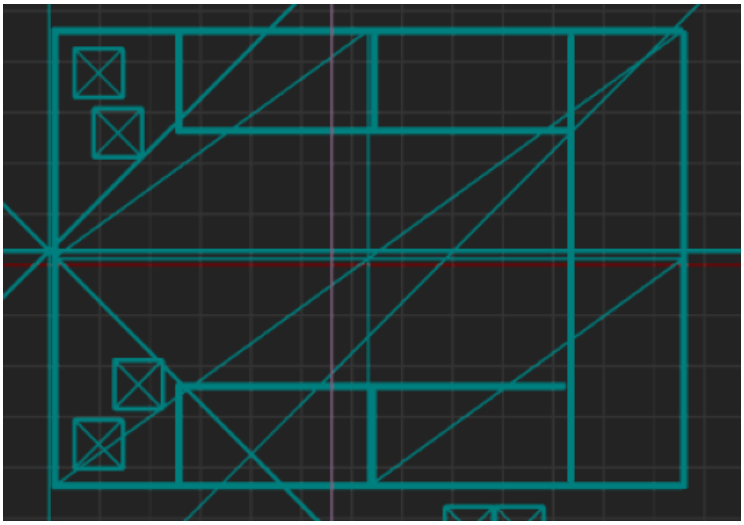


Figure 20 : Top Perspective Blueprint of Simulation Environment

3.5.2 PYGAME WITH ROS

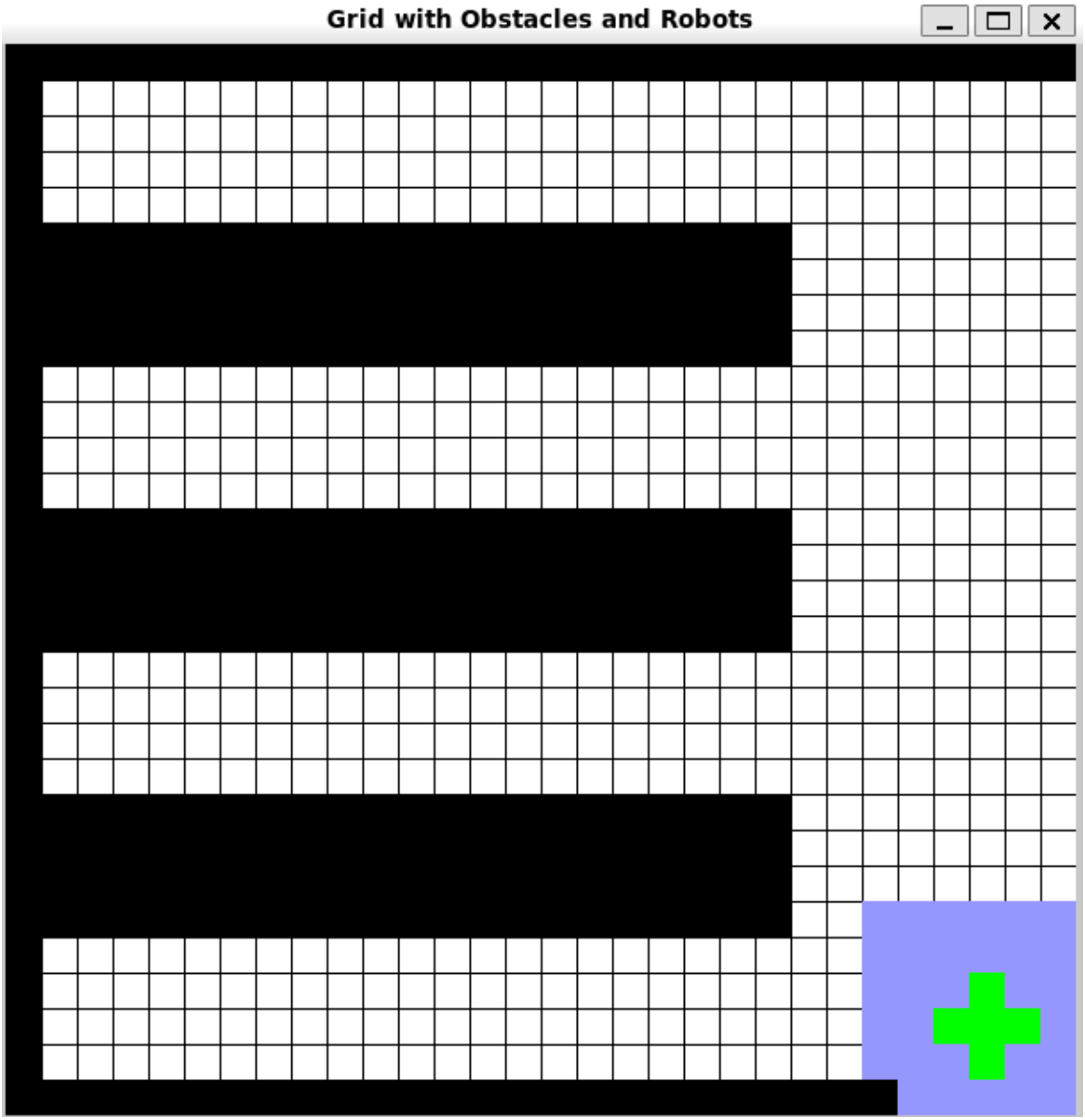


Figure 21 : Occupancy Grid Map for environment with swarm and lidar scan area highlighted

Pygame APIs can be used to generate grid , defIning cell size as well as grid size. Further , the occupancy map can be completed by defining an obstacle array and marking all cells of those coordinates with black. This pygame window can be further updated continuously which allows easy implementation and simulation of navigation algorithms.

3.6 RESULTS

3.6.1 Airim

- Swarm Simulation for Navigation In Indoor Environment

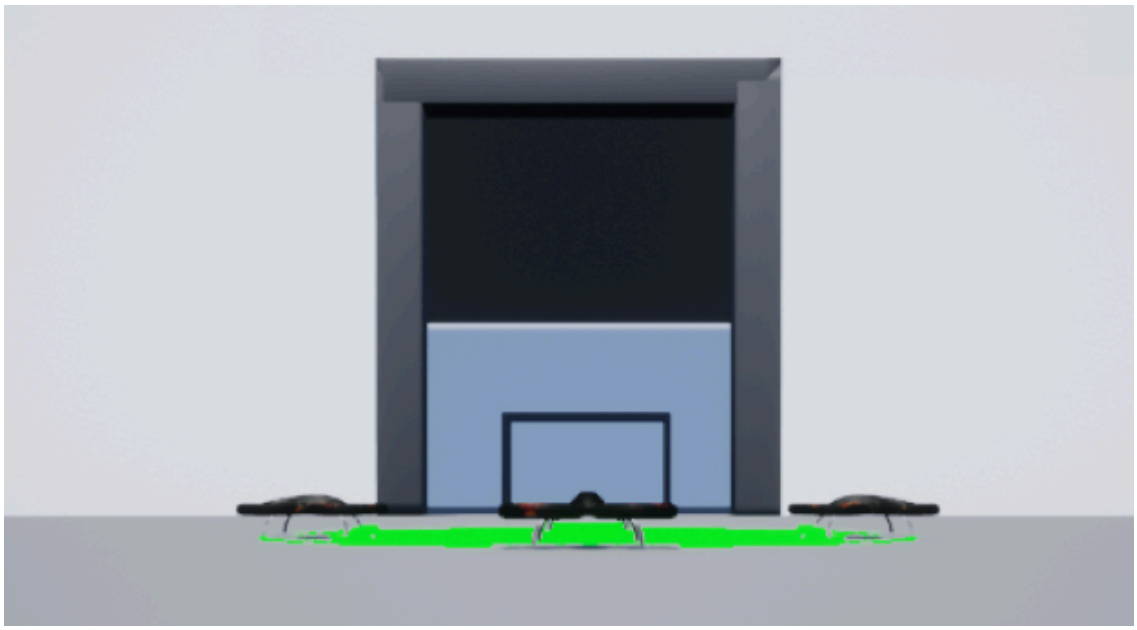


Figure 22 : Swarm Ready to Take off , LIDAR Enabled

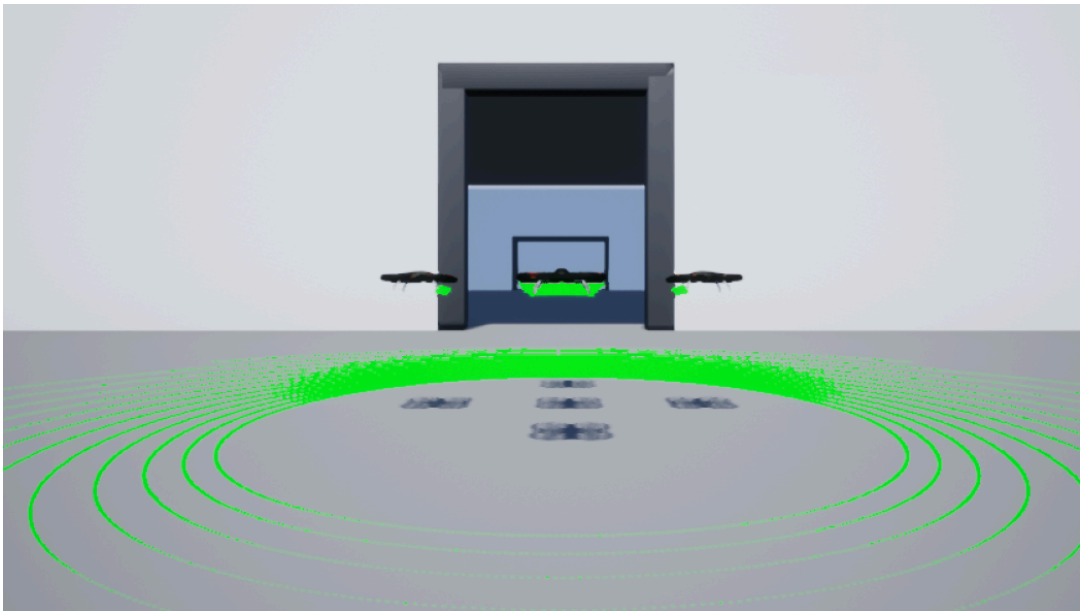


Figure 23 : Swarm took off and moving towards entry

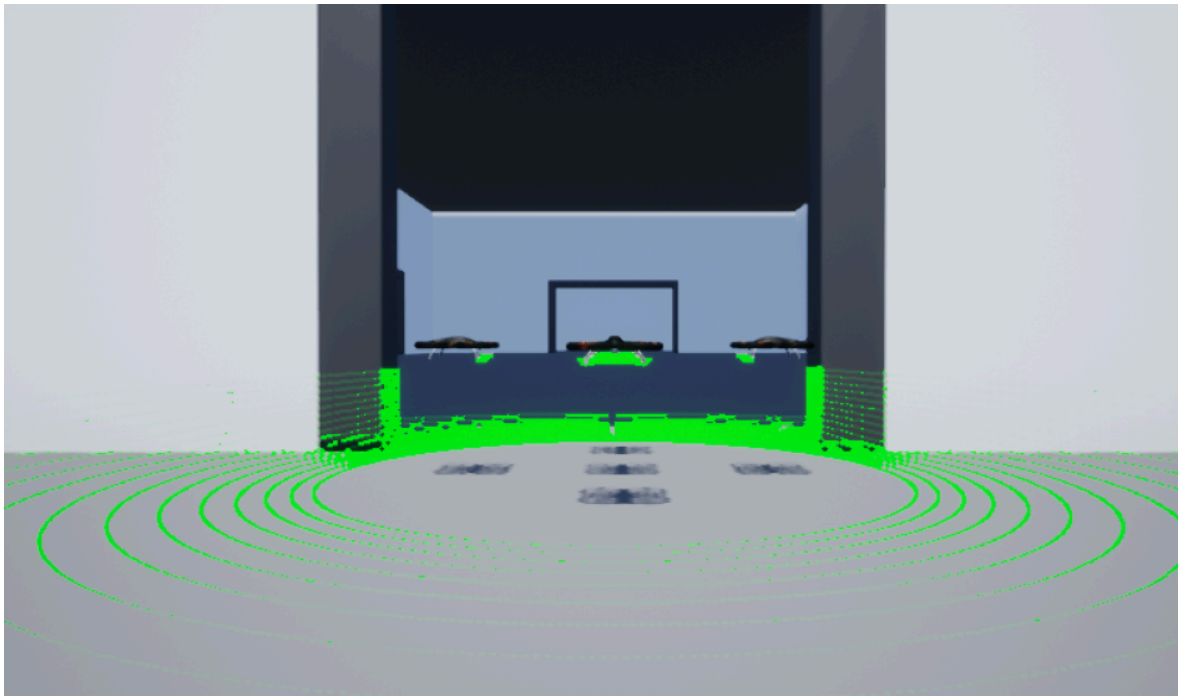


Figure 24 : Swarm entering through the door

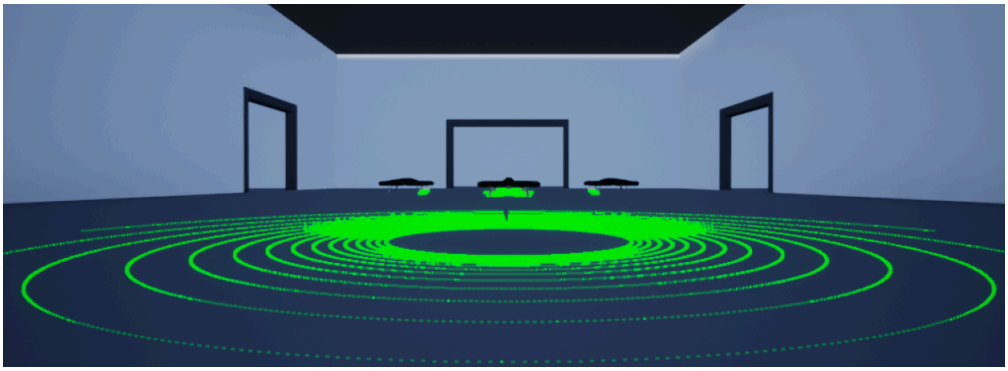


Figure 25 : Swarm Stops when Door Detected

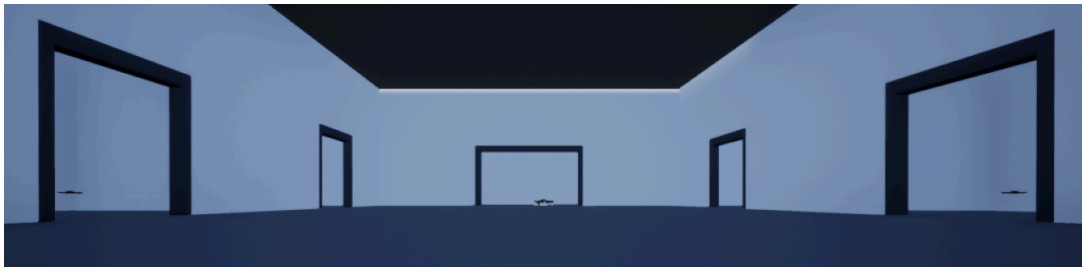


Figure 26 : Slave Drones separate to enter room for exploration

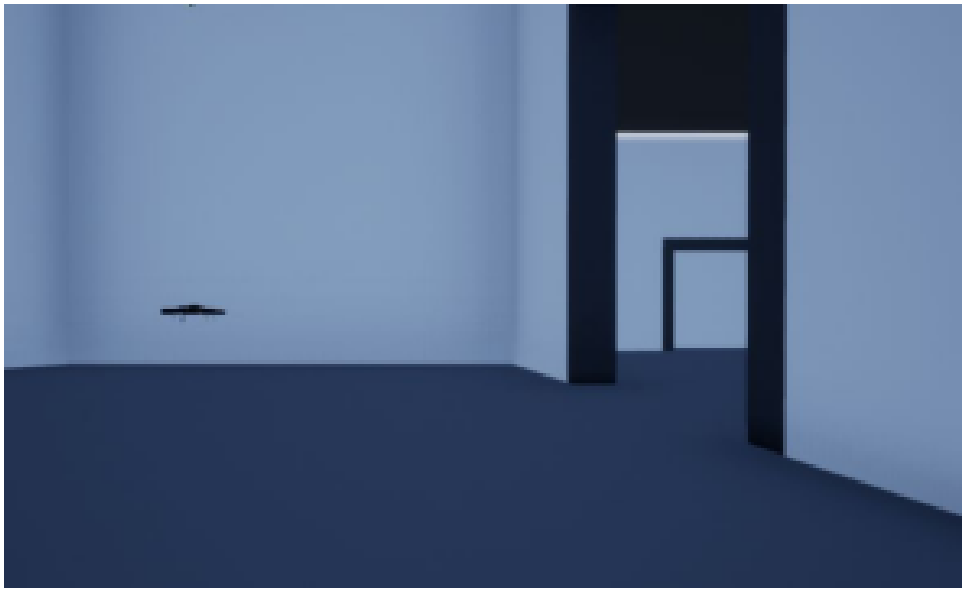


Figure 27 : Drone Enters room

- Door Detection
  DOOR DETECTOR CLASS

```python
class Door_Detector():

  def __init__(self,thresh_low,thresh_high,image,vehicle_name):

    self.tl = thresh_low
    self.th = thresh_high
    self.image = image
    self.image_size = image.shape
    self.vehicle_name = vehicle_name

    self.top_half = image.shape
    self.gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)

    self.edge = cv2.Canny(image, thresh_low, thresh_high)



    def get_point_clusters():
      pts = []
      lat=[]
      lon=[]
      for i in range(20,self.image_size[0]//2):
        for j in range(self.image_size[1]):
          if self.edge[i][j] == 255:
            pts.append([i,j])
```

```
            lat.append(i)
            lon.append(j)

      return [pts,lat,lon]

    self.pts,self.lats,self.lons = get_point_clusters()

    def split_image():
      img_size = self.image.shape
      tp = img_size[0]//2
      top_half = self.image[::tp]
      centre_x = img_size[1]//2
      top_left =[]
      top_right = []
      for i in range(tp):
        for j in range(centre_x):
          if self.image[i][j] == 255:
            top_left.append([i,j])
      for i in range(tp):
        for j in range(centre_x,2*centre_x):
          if self.image[i][j] == 255:
            top_right.append([i,j])

      return [top_left,top_right]

  def check_door(self):
    lat_to_lon = defaultdict(int)
    for x,y in self.pts:
      lat_to_lon[x] += 1
    result_dict = dict(lat_to_lon)
    lst = list(result_dict.values())

    quadrat_y= []
    for i in self.pts:
      if result_dict[i[0]] == 4:
        quadrat_y.append(i[1])
    set_y = list(set(quadrat_y))
    print(lst.count(4))
    print(np.mean(set_y))
    if lst.count(4) > 28 and np.mean(set_y) < 150 and np.mean(set_y)> 125:
      return True
    else:
      return False
```

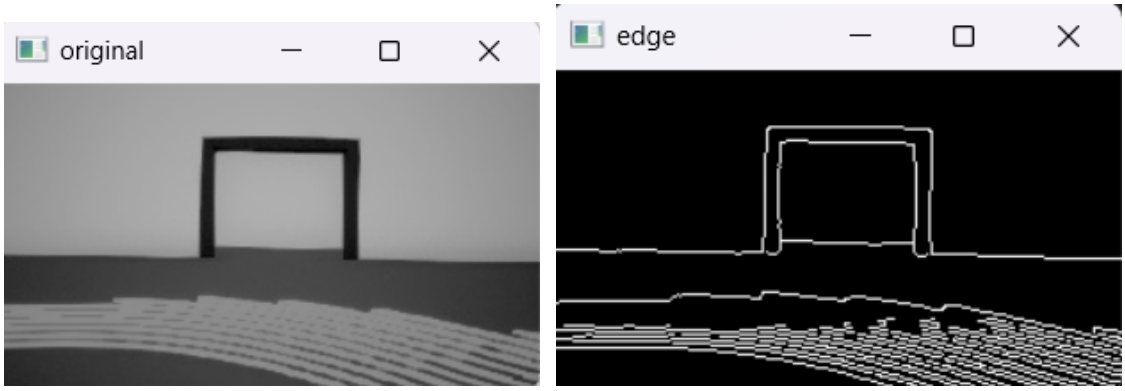


Figure 28 : Original image from drone camera and edge detection output

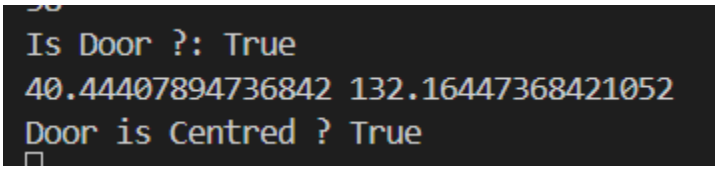


Figure 29 : Code detects door as well as its position in the frame

3.6.2 ROS and Pygame

- Master Drone Node

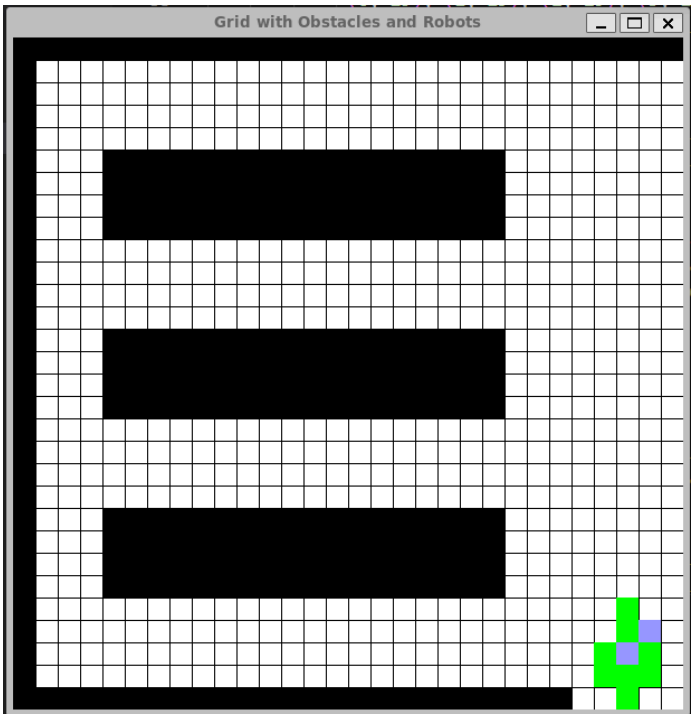


Figure 30 : Master Node initialized , Swarm Starts Navigation

SWARM CLASS

```python
class Swarm:
  def __init__(self,init_pos,screen):
    self.init_pos = init_pos
    self.vel_x = 0
    self.vel_y = 0
    self.current_pose_x = self.init_pos[0]
    self.current_pose_y = self.init_pos[1]
    self.drone_count = 5
    self.drone_list = ["left","right","master","front","back"]
    self.curr_drone_list = ["left","right","master","front","back"]
    self.Path_Data = []
    self.detatch_data = [0,0,0,0,0]
    self.screen = screen


                                                    self.motion_pub       =
rospy.Publisher('Motion_Data',Int32MultiArray,queue_size = 1)
                                                     self.det_pub_l       =
rospy.Publisher('Left_Detatch_Data',Int32MultiArray,queue_size = 10)
                                                     self.det_pub_r       =
rospy.Publisher('Right_Detatch_Data',Int32MultiArray,queue_size = 10)
                                                     self.det_pub_f       =
rospy.Publisher('Front_Detatch_Data',Int32MultiArray,queue_size = 10)
                                                     self.det_pub_b       =
rospy.Publisher('Back_Detatch_Data',Int32MultiArray,queue_size = 10)
    self.Motion_Data = Int32MultiArray()
    self.Motion_Data.data = self.Path_Data
    self.Detatch_Poses = Int32MultiArray()
    self.Detatch_Poses.layout.dim.append(MultiArrayDimension())
    self.Detatch_Poses.layout.dim[0].label = 'point_x'
    self.Detatch_Poses.layout.dim[0].size = 5
    self.Detatch_Poses.layout.dim[0].stride = 5


    self.Detatch_Poses.data = self.detatch_data

  def detatch(self,drone_name):
    self.curr_drone_list.remove(drone_name)
    self.drone_count-=1
    if drone_name == "left":
      i=0
      while i < 3:
        print("publishing: ",self.detatch_data)
        self.det_pub_l.publish(self.Detatch_Poses)
        i+=1
    elif drone_name == "right":
      i=0
      while i < 3:
        print("publishing: ",self.detatch_data)
```

```
      self.det_pub_r.publish(self.Detatch_Poses)
      i+=1
    elif drone_name == "front":
      i=0
      while i < 3:
        print("publishing: ",self.detatch_data)
        self.det_pub_f.publish(self.Detatch_Poses)
        i+=1
    elif drone_name == "back":
      i=0
      while i < 3:
        print("publishing: ",self.detatch_data)
        self.det_pub_b.publish(self.Detatch_Poses)
        i+=1
    return


  def navigate(self):
    d = door_detector("Master")
    self.current_pose_x = self.init_pos[0]
    self.current_pose_y = self.init_pos[1]
    door_detected = 0

    self.motion_pub.publish(self.Motion_Data)

    while self.drone_count != 1:
      self.vel_y = 1
                  while door_detected == 0 and self.current_pose_x >0 and
self.current_pose_y > 0:
        new_pose_x = self.current_pose_x - self.vel_x
        new_pose_y = self.current_pose_y - self.vel_y
        print(self.current_pose_x,self.current_pose_y)
                                                                  cam_scan    =
d.get_cam_scan(grid,self.current_pose_x,self.current_pose_y)
        source_col = 0
        for i in cam_scan:
          if 20 in i:
            source_col= list(i).index(20)
        print(cam_scan)
                                                                      pat     =
d.detect_pattern(cam_scan,[self.current_pose_x,self.current_pose_y],sourc
e_col)
        print(pat)
          pygame.draw.rect(screen, RED, (new_pose_x * CELL_SIZE, new_pose_y *
CELL_SIZE, CELL_SIZE, CELL_SIZE))
        if self.drone_count == 5:
          for i, j in [(-1,0),(0, -1), (1, 0), (0, 1)]:
                      pygame.draw.rect(screen, RED, ((S.current_pose_x + i) *
CELL_SIZE, (S.current_pose_y + j) * CELL_SIZE, CELL_SIZE, CELL_SIZE))
        if self.drone_count == 4:
          for i, j in [(0, -1), (1, 0), (0, 1)]:
```

```
            pygame.draw.rect(screen, RED, (new_pose_x * CELL_SIZE, new_pose_y
* CELL_SIZE, CELL_SIZE, CELL_SIZE))
                    pygame.draw.rect(screen, BLUE, ((S.current_pose_x + i) *
CELL_SIZE, (S.current_pose_y + j) * CELL_SIZE, CELL_SIZE, CELL_SIZE))

           pygame.display.flip()
           time.sleep(0.5)
       elif self.drone_count == 3:
           for i, j in [(1, 0), (0, 1)]:
                    pygame.draw.rect(screen, RED, (new_pose_x * CELL_SIZE,
new_pose_y * CELL_SIZE, CELL_SIZE, CELL_SIZE))
                    pygame.draw.rect(screen, RED, ((S.current_pose_x + i) *
CELL_SIZE, (S.current_pose_y + j) * CELL_SIZE, CELL_SIZE, CELL_SIZE))

            pygame.display.flip()
            time.sleep(0.5)
       elif self.drone_count == 2:
            pygame.draw.rect(screen, RED, (new_pose_x * CELL_SIZE, new_pose_y
* CELL_SIZE, CELL_SIZE, CELL_SIZE))
           for i, j in [(0, 1)]:
                    pygame.draw.rect(screen, BLUE, ((S.current_pose_x + i) *
CELL_SIZE, (S.current_pose_y + j) * CELL_SIZE, CELL_SIZE, CELL_SIZE))

            pygame.display.flip()
            time.sleep(0.5)



       self.Path_Data.append(self.current_pose_x)
       self.Path_Data.append(self.current_pose_y)

       self.current_pose_x = new_pose_x
       self.current_pose_y = new_pose_y
       if pat[0] == True and pat[1] == True:
         print(self.current_pose_x,self.current_pose_y)
         print("door_detected")
         door_detected = 1
         if self.drone_count == 5:
           self.vel_y = 0
           self.detatch_data[0] = self.current_pose_x
           self.detatch_data[1] = self.current_pose_y
           self.detatch_data[-1] = d.door_dist
           self.detatch("left")
           self.vel_y = 1
           self.vel_y = 1
         elif self.drone_count == 4:
           self.vel_y = 0
           self.detatch_data[0] = self.current_pose_x
           self.detatch_data[1] = self.current_pose_y
           self.detatch_data[-1] = d.door_dist
           self.detatch("back")
```

```
            self.vel_y = 1
            self.vel_y = 1
          elif self.drone_count == 3:
            self.vel_y = 0

            self.detatch_data[0] = self.current_pose_x
            self.detatch_data[1] = self.current_pose_y
            self.detatch_data[-1] = d.door_dist
            self.detatch("front")
            self.vel_y = 1
            self.vel_y = 1          elif self.drone_count == 2:
            self.vel_y = 0

            self.detatch_data[0] = self.current_pose_x
            self.detatch_data[1] = self.current_pose_y
            self.detatch_data[-1] = d.door_dist
            self.detatch("right")
            self.vel_y = 1
            self.vel_y = 1
          print("Detatched",self.drone_count)
          door_detected = 0

      if self.drone_count == 1:
        self.detatch_data[2] = self.current_pose_x
        self.detatch_data[3] = self.current_pose_y

        i=0

        while i < 100:
          self.det_pub_f.publish(self.Detatch_Poses)
          self.det_pub_b.publish(self.Detatch_Poses)
          self.det_pub_r.publish(self.Detatch_Poses)
          self.det_pub_l.publish(self.Detatch_Poses)
          i+=1
          time.sleep(0.25)
```

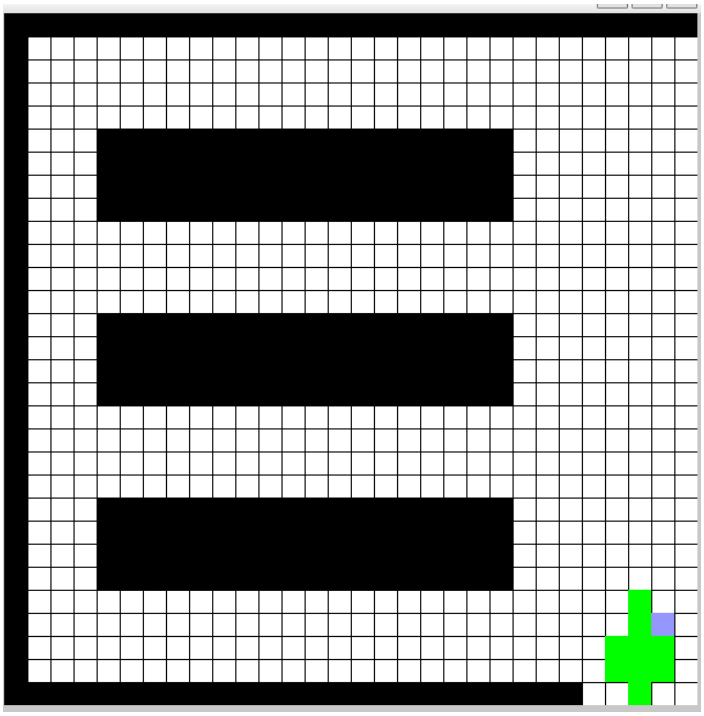


Figure 31 : Door Detected on left , Left Drone instructed to separate

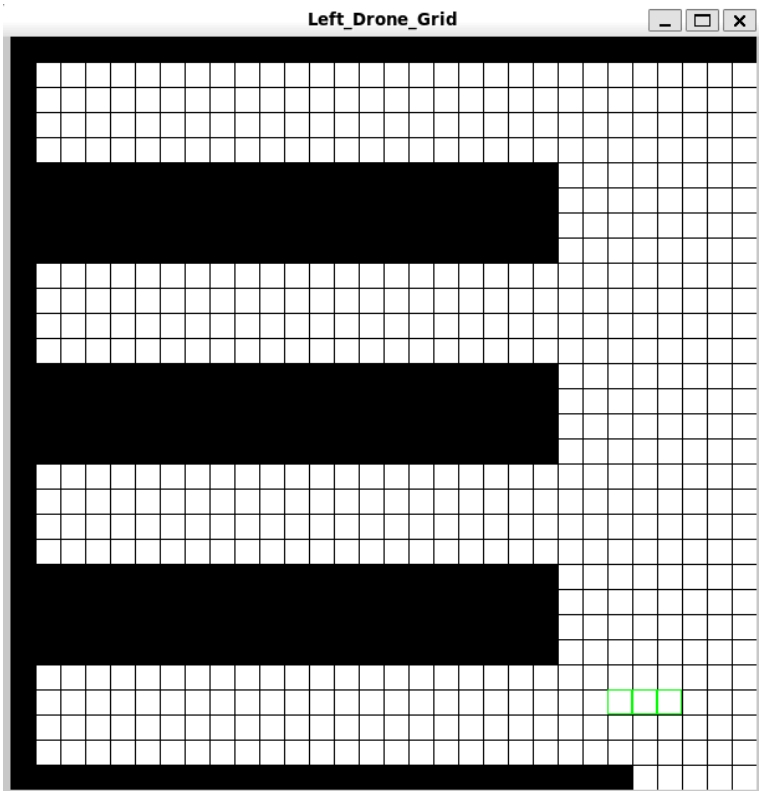


Figure 32 : Left Drone node subscribes to master and separates for exploration

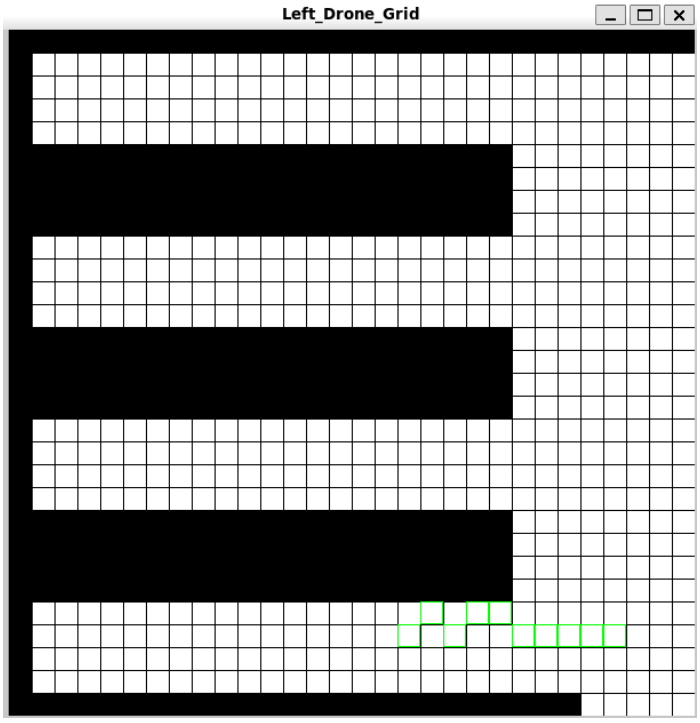


Figure 33 : Left Drone enters door and starts exploration

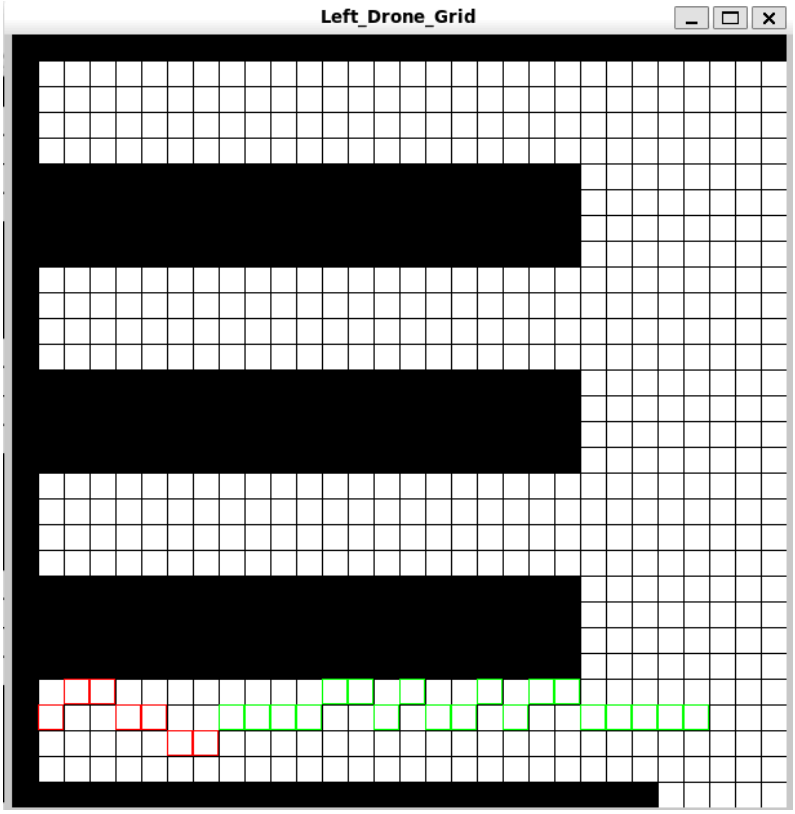


Figure 34 : Drone Reached Dead End , backtracking to close loop

At this stage , the left slave drone has finished its temporary task , it will now wait for the master drone to publish a message telling the drones its position . While this is happening , simultaneously , the rest of the swarm is still moving looking for other doors so that the other slave drones can be dispatched for exploration.

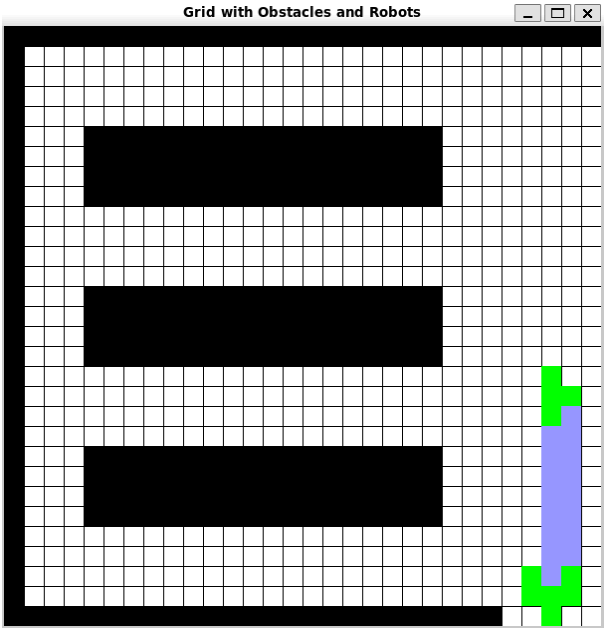


Figure 35 : The swarm reaches second door , back drone separates

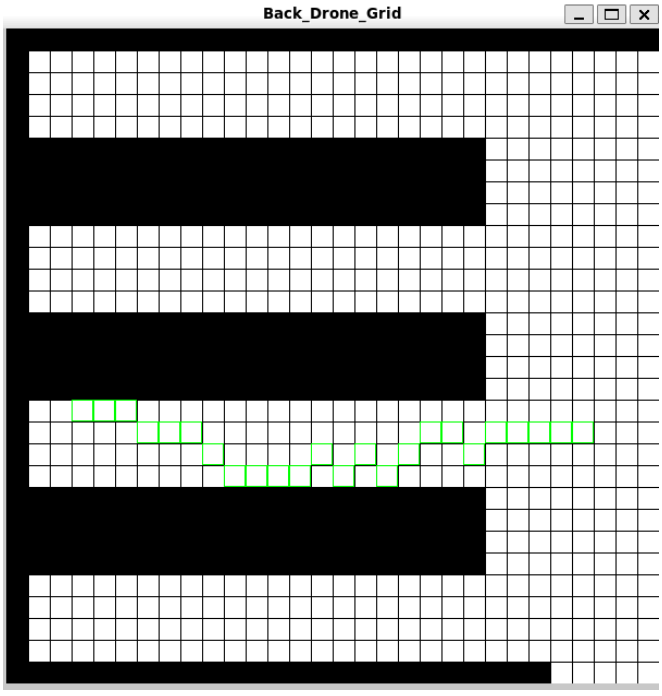


Figure 36 : Back Drone separates from swarm and starts exploration

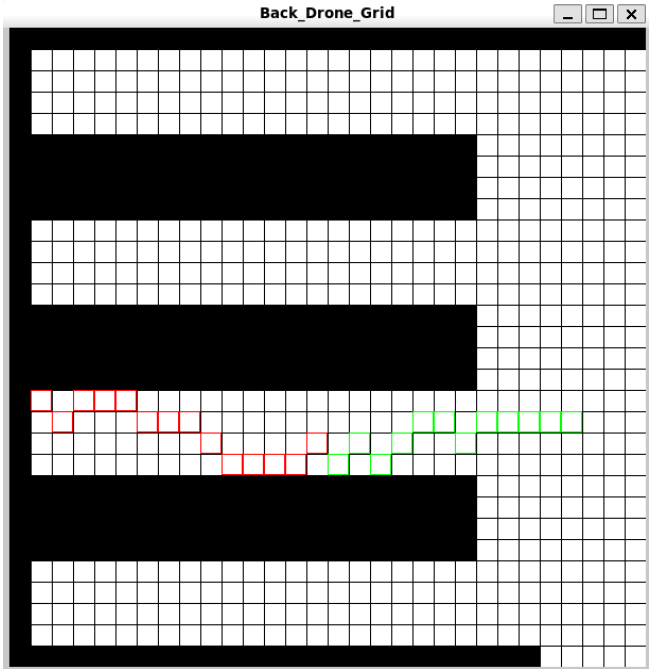


Figure 37 : Drone Reached Dead End , backtracking to close loop

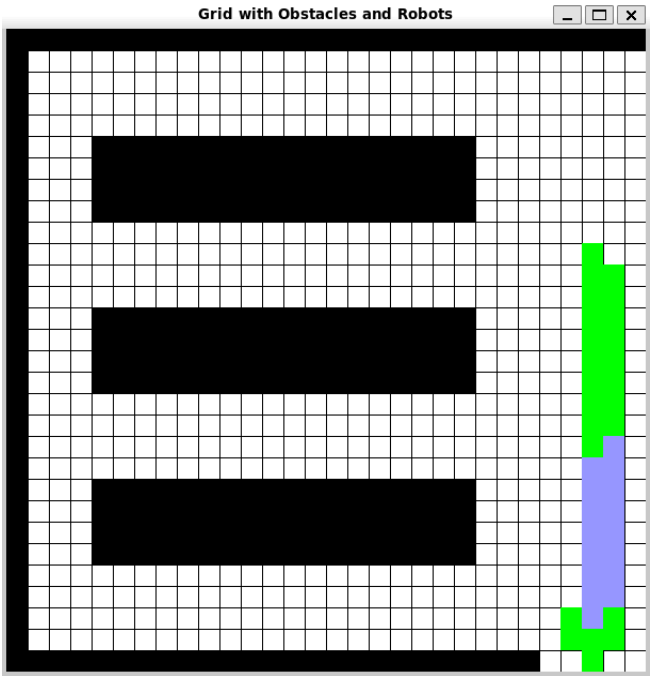


Figure 38 : The swarm reaches third door , front drone separates

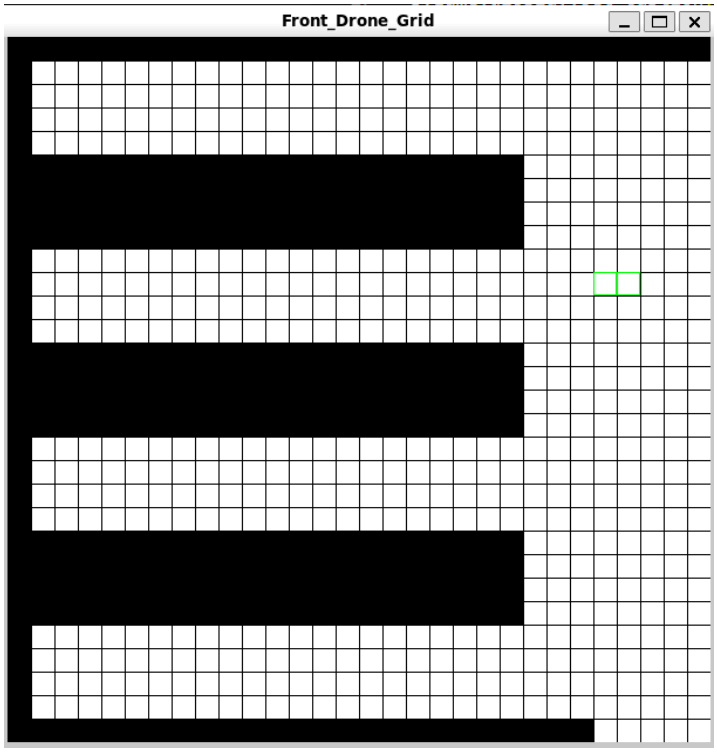


Figure 39 : Front Drone separates from swarm

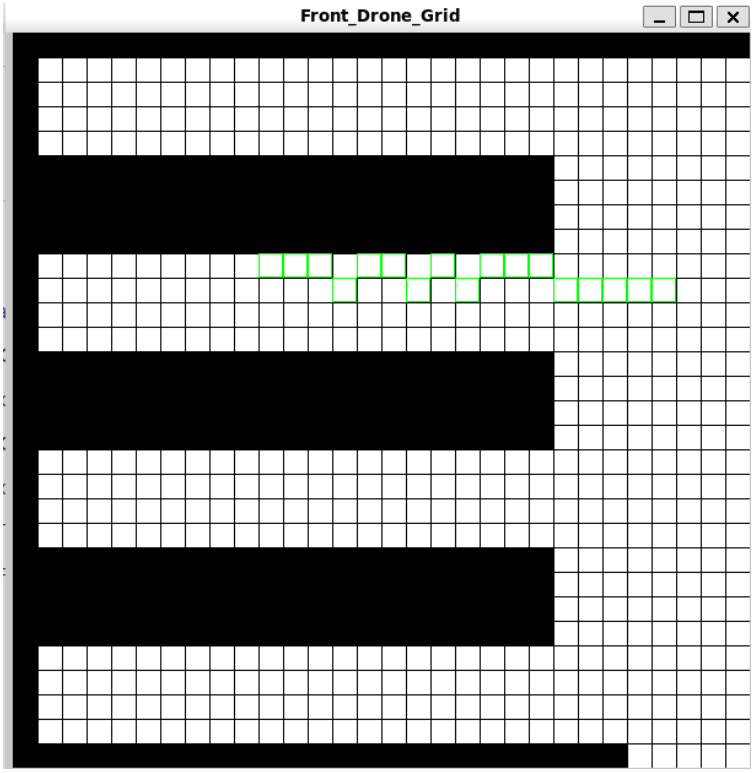


Figure 40 : Front Drone starts exploration

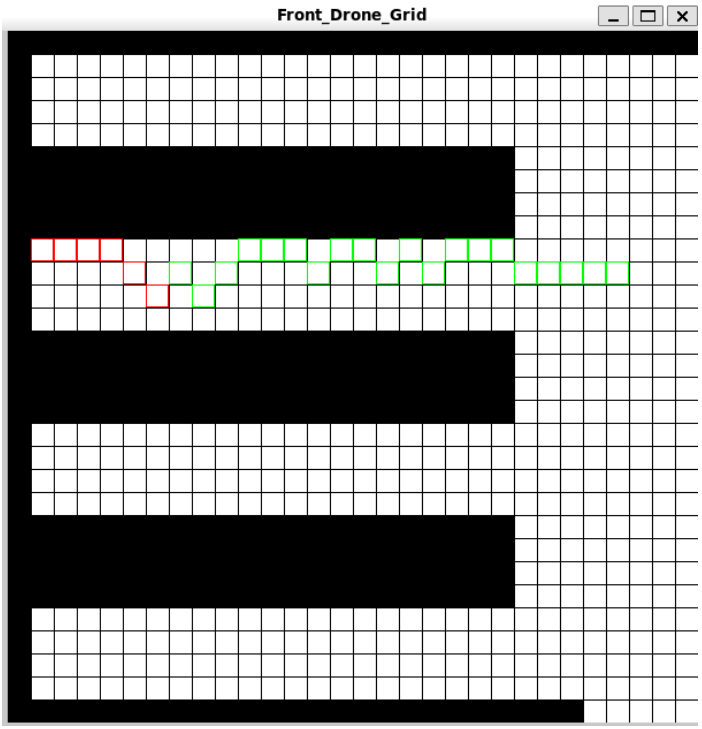


Figure 41 : Drone Reached Dead End , backtracking to close loop

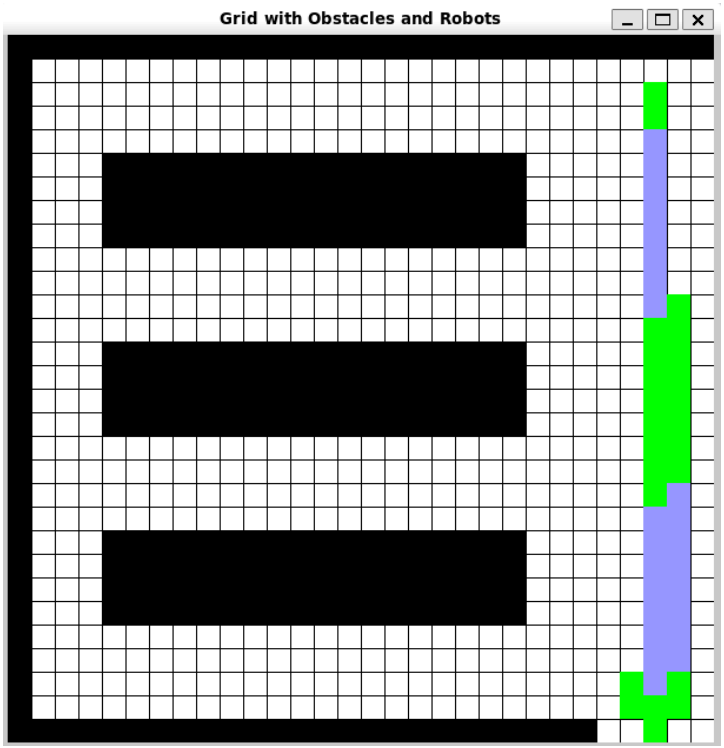


Figure 42 : The swarm reaches the last door , right drone separates

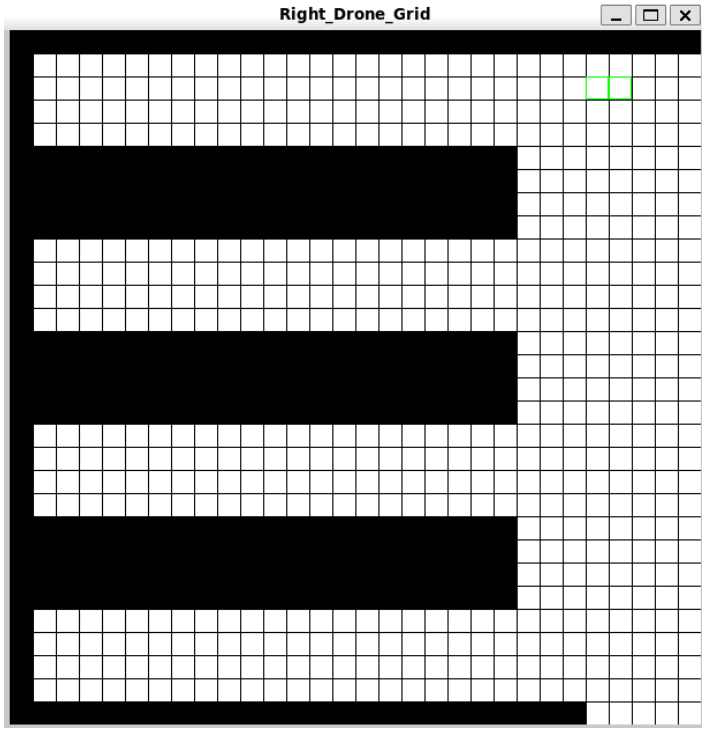


Figure 43 : Right Drone separates from swarm

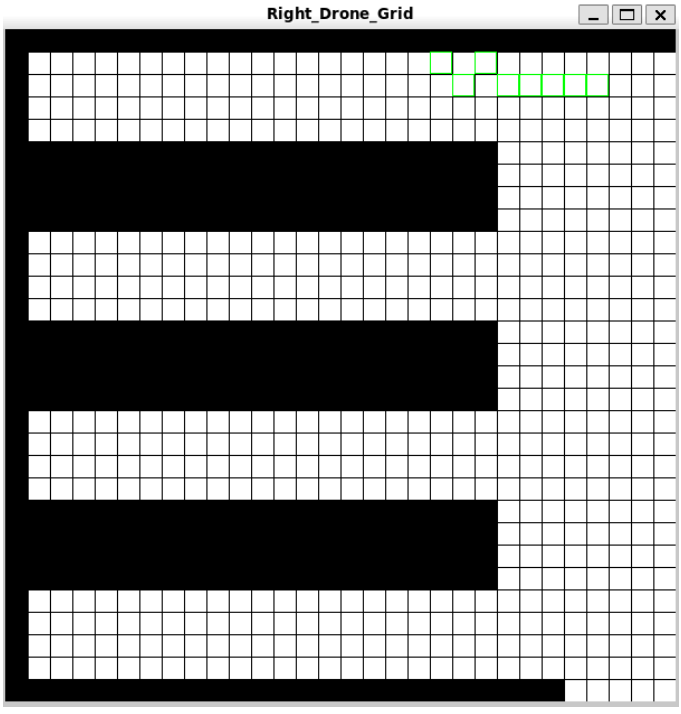


Figure 44 : Right Drone starts exploration

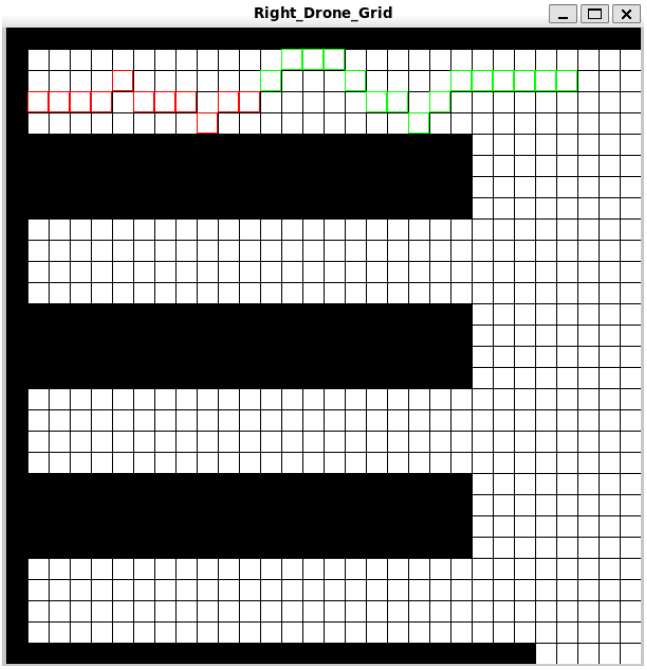


Figure 45 : Drone Reached Dead End , backtracking to close loop

Now , the swarm has separated from all 4 drones , as such , the master node publishes its current position to the slave drones , allowing them to calculate their destination point based on the point where they detached. Now , all four drones will start moving towards the master drone to complete the swarm.

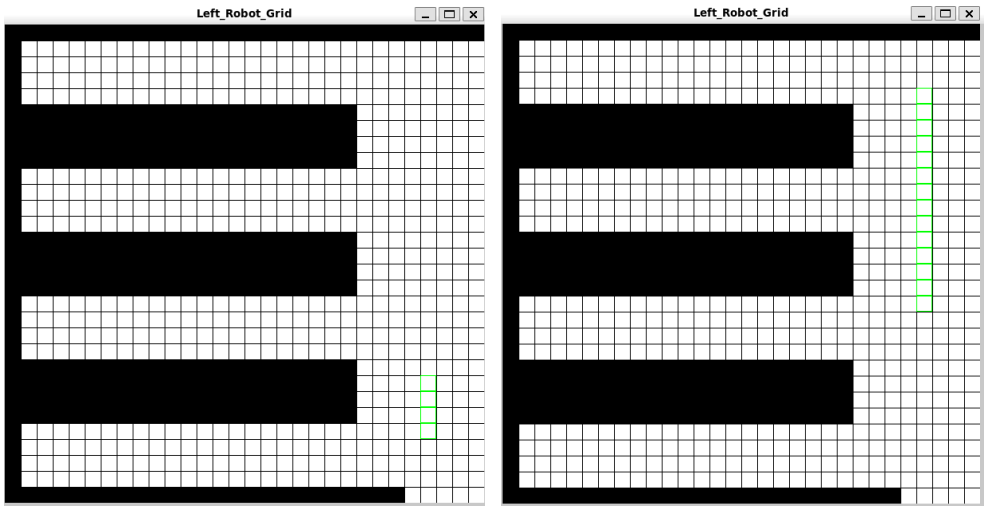


Figure 46 : Left Drone moving back to master

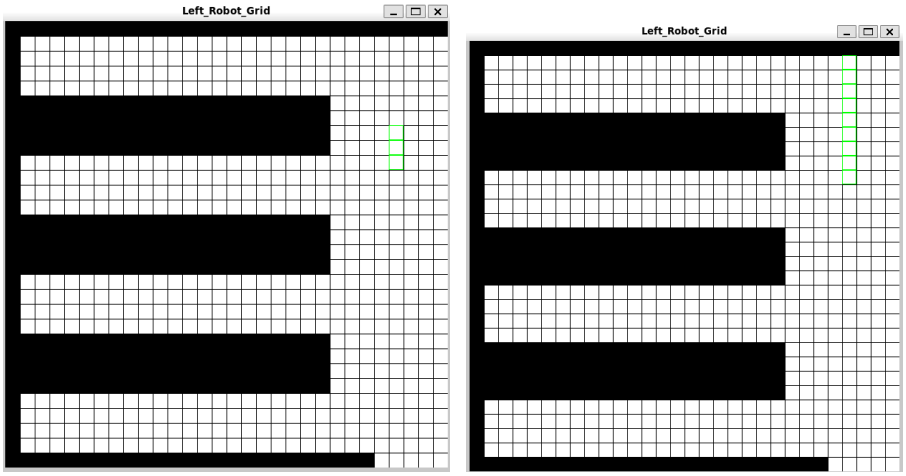


Figure 47 : Front Drone moving back to master

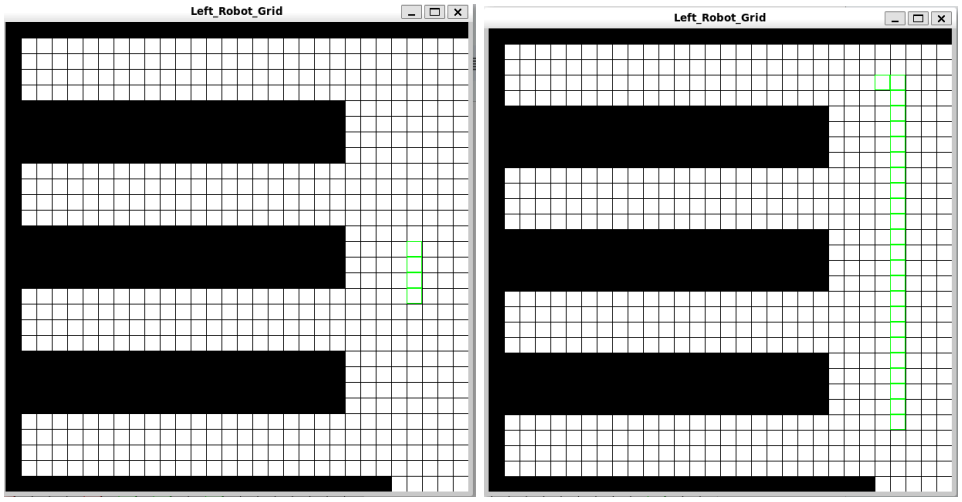


Figure 48 : Back Drone moving back to master

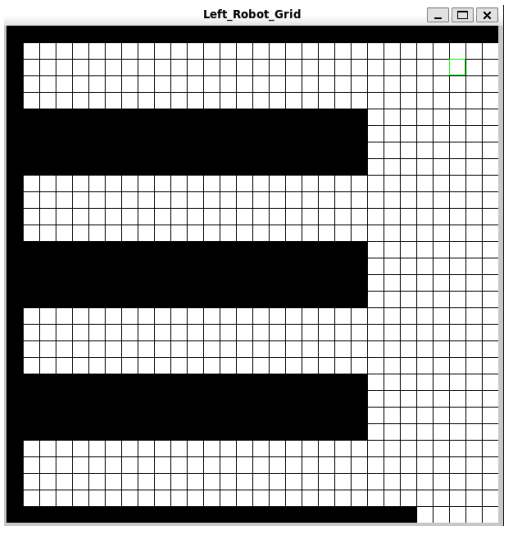


Figure 49 : As the right drone loop closes, it has already reached master

SLAVE DRONE CLASS

Python

```
class Slave():
  def __init__(self,name,screen):
    self.name = name
    self.vel_x = -1
    self.vel_y = 0
    self.det_data = None
    self.mast_loc = None
    self.screen = screen
    self.x = 0
    self.y = 0
    self.map = np.zeros((30,30))
    self.obstacles = []
    self.map_data_points = []
    self.checkpoint = [0,0]
    self.grid = grid
                                                    self.map_pub      =
rospy.Publisher("Left_Data",Int32MultiArray,queue_size = 1)

    self.map_data = Int32MultiArray()
    self.map_data.layout.dim.append(MultiArrayDimension())
    self.map_data.layout.dim[0].label = "len"



  def subscriber_callback(self,data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
    self.det_data = data.data
    print(data.data)

  def subscribe_init(self):
    rospy.init_node("Slave_Drone",anonymous=True)
    print("Subscribing")
    print(self.det_data)
    rate = rospy.Rate(3)
    while self.det_data == None:
                                                        slave_sub     =
rospy.Subscriber('Left_Detatch_Data',Int32MultiArray,self.subscriber_c
allback)
      rate.sleep()

  def subscriber_mast_callback(self,data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
    self.mast_loc = data.data
    print(data.data)

  def subscribe_mast_init(self):
    rospy.init_node("Slave_Drone",anonymous=True)
    print("Subscribing _ Master")
```

```
    print(self.mast_loc)
    rate = rospy.Rate(3)
    while self.mast_loc == None:
                                                            slave_sub     =
rospy.Subscriber('Left_Detatch_Data',Int32MultiArray,self.subscriber_mast
_callback)
      rate.sleep()


  def get_lidar_scan(matrix, x, y):
    lidar_scan = []
    rows, cols = len(matrix), len(matrix[0])

    # Calculate starting and ending indices for the lidar_scan
    start_row = max(0, x - 3)
    end_row = min(rows - 1, x + 3)
    start_col = max(0, y - 3)
    end_col = min(cols - 1, y + 3)

    # Extract the lidar_scan
    for i in range(start_row, end_row + 1):
      row = []
      for j in range(start_col, end_col + 1):
        row.append(matrix[i][j])
      lidar_scan.append(row)

    return lidar_scan

  def path_to_master(self):
    x_dist = abs(self.mast_loc[1] - self.checkpoint[1])
    y_dist = abs(self.mast_loc[0] - self.checkpoint[0])
    print(self.checkpoint)
    print("dist:",x_dist,y_dist,self.x,self.y)

    for i in range(x_dist-1):
      self.y-=1
            pygame.draw.rect(screen, RED, ((self.x) * CELL_SIZE, (self.y) *
CELL_SIZE, CELL_SIZE, CELL_SIZE),1)
      pygame.display.flip()
      time.sleep(0.5)
    for i in range(y_dist):
      self.x+=1
            pygame.draw.rect(screen, RED, ((self.x) * CELL_SIZE, (self.y) *
CELL_SIZE, CELL_SIZE, CELL_SIZE),1)
      pygame.display.flip()
      time.sleep(0.5)
    self.x-= 1
    pygame.draw.rect(screen, RED, ((self.x) * CELL_SIZE, (self.y) * CELL_SIZE,
CELL_SIZE, CELL_SIZE),1)
    pygame.display.flip()
    time.sleep(0.5)
```

```
    print("Master Found")

  def get_safe_points(matrix, x, y):
    neighbor_indices = []
    if matrix[x, y - 1] != 1:
      neighbor_indices.append((x, y - 1))
    if matrix[x - 1, y - 1] != 1:
      neighbor_indices.append((x - 1, y - 1))
    if matrix[x + 1, y - 1] != 1:
      neighbor_indices.append((x + 1, y - 1))
    if matrix[x - 1, y] != 1:
      neighbor_indices.append((x - 1, y))
    if matrix[x + 1, y] != 1:
      neighbor_indices.append((x + 1, y))
    return neighbor_indices

  def slam(self, door_dist,init_point):
    path_forward = []
    break_flag = 0
    self.x = init_point[0]
    self.y = init_point[1]
    self.checkpoint = [self.x,self.y]
    for i in range(door_dist-1):
        pygame.draw.rect(screen, RED, ((self.x + i) * CELL_SIZE, (self.y + j) *
CELL_SIZE, CELL_SIZE, CELL_SIZE),1)
      time.sleep(0.5)
      path_forward.append((self.y + self.vel_y,self.x+self.vel_x))
      self.x += self.vel_x
      self.y += self.vel_y
      pygame.display.flip()

    print(self.checkpoint)
    print(self.x,self.y)
    while break_flag != 1:
      print(break_flag)
      print(self.x,self,y)
                  padded_matrix = np.pad(self.grid, 3, mode='constant',
constant_values=1)
      extracted_array = padded_matrix[self.y:self.y+7, self.x:self.x+7]
      lidar_scan = extracted_array
      print(lidar_scan)


      neighbor_indices = []
      if lidar_scan[3, 2] != 1:
        neighbor_indices.append((-1,0))
      if lidar_scan[2, 2] != 1:
        neighbor_indices.append((- 1,- 1))
      if lidar_scan[4, 2] != 1:
        neighbor_indices.append((-1, 1))
      print(neighbor_indices)
```

```
     if len(neighbor_indices) > 0:
                                                                choice    =
neighbor_indices[random.randint(0,len(neighbor_indices)-1)]

       self.vel_x = choice[0]
       self.vel_y = choice[1]
       print(self.x+self.vel_x,self.vel_y+self.y)
       obstacles = []
       for i in range(len(lidar_scan)):
         for j in range(len(lidar_scan[0])):
           if lidar_scan[i][j] == 1:
             if (self.y+i-3,self.x+j-3) not in obstacles:
               obstacles.append((self.y+i-3,self.x+j-3))
       print(obstacles)
       self.obstacles.append(obstacles)
       path_forward.append((self.y + self.vel_y,self.x+self.vel_x))

       if ((self.y + self.vel_y,self.x+self.vel_x) not in obstacles and
         self.x + self.vel_x <29 and self.y + self.vel_y <29 and
         self.x + self.vel_x >0 and self.y + self.vel_y >0 ) :
         self.x += self.vel_x
         self.y += self.vel_y
         time.sleep(0.5)
                 pygame.draw.rect(screen, RED, (self.x * CELL_SIZE, self.y *
CELL_SIZE, CELL_SIZE, CELL_SIZE),1)
         pygame.display.flip()
     else:
       print("Obstacles Everywhere, Backtracking")
       backtrack = path_forward[::-1]
       for i,j in backtrack:
         time.sleep(0.5)
         self.x = j
         self.y = i
         print(self.x+1,self.y,self.checkpoint)
              pygame.draw.rect(screen, GREEN, (j * CELL_SIZE, i * CELL_SIZE,
CELL_SIZE, CELL_SIZE),1)
         pygame.display.flip()
                        if (self.x+1 == self.checkpoint[0]) and (self.y ==
self.checkpoint[1]):
           break_flag = 1
```

Figure 50 : Map Generated by combining obstacle data form all slave drones

MAP GENERATOR NODE

Python

```
class Map_Generator:

  def __init__(self,size):
    self.map = []
    self.left_data = None
    self.right_data = None
    self.front_data = None
    self.back_data = None
    self.obstacles = []

  def left_callback(self,data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
    self.left_data = data.data
    print(data.data)
```

```
    # rospy.signal_shutdown("Message Received")

  def left_init(self):
    rospy.init_node("Map_Generator",anonymous=True)
    print("Subscribing")
    rate = rospy.Rate(3)
    while self.left_data == None:
                                                        slave_sub     =
rospy.Subscriber('Left_Data',Int32MultiArray,self.left_callback)
      rate.sleep()

  def right_callback(self,data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
    self.right_data = data.data
    print(data.data)
    # rospy.signal_shutdown("Message Received")

  def right_init(self):
    rospy.init_node("Map_Generator",anonymous=True)
    print("Subscribing")
    rate = rospy.Rate(3)
    while self.right_data == None:
                                                        slave_sub     =
rospy.Subscriber('Right_Data',Int32MultiArray,self.right_callback)
      rate.sleep()

  def front_callback(self,data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
    self.front_data = data.data
    print(data.data)
    # rospy.signal_shutdown("Message Received")

  def front_init(self):
    rospy.init_node("Map_Generator",anonymous=True)
    print("Subscribing")
    rate = rospy.Rate(3)
    while self.front_data == None:
                                                        slave_sub     =
rospy.Subscriber('Front_Data',Int32MultiArray,self.front_callback)
      rate.sleep()

  def back_callback(self,data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
    self.back_data = data.data
    print(data.data)
    # rospy.signal_shutdown("Message Received")

  def back_init(self):
    rospy.init_node("Map_Generator",anonymous=True)
    print("Subscribing")
    rate = rospy.Rate(3)
```

```
    while self.back_data == None:
                                                                slave_sub     =
rospy.Subscriber('Back_Data',Int32MultiArray,self.back_callback)
      rate.sleep()

def transpose_array(arr):
  # Use list comprehension to transpose the array
    return [[arr[j][i] for j in range(len(arr))] for i in range(len(arr[0]))]

def merge_arrays(arr1,arr2,row_col_choice):
  if row_col_choice == 1: #merge_by_row
    for i in arr1:
      if i in arr2:
        k1 = arr1.index(i)
        k2 = arr2.index(i)
        print(k1,k2)
        if k1 == len(arr1)-1 and k2 == 0:
          arr1.extend(arr2[1:])
          return arr1
        elif k1 == 0 and k2 == len(arr2[1:])-1:
          arr2.extend(arr1)
          return arr2
      else:
        print("Cannot be merged")
  elif row_col_choice == 0:
    arr1 = transpose_array(arr1)
    print(arr1)
    arr2 = transpose_array(arr2)
    print(arr2)
    for i in arr1:
      if i in arr2:
        k1 = arr1.index(i)
        k2 = arr2.index(i)
        print(k1,k2)
        if k1 == len(arr1)-1 and k2 == 0:
          arr1.extend(arr2[1:])
          return transpose_array(arr1)
        elif k1 == 0 and k2 == len(arr2[1:])-1:
          arr2.extend(arr1)
          return transpose_array(arr2)
      else:
        print("Cannot be merged")
```

CHAPTER 4

RESULTS AND ANALYSIS

The working of the algorithm includes comparing three different SLAM / Navigation algorithms for exploration , tested over four different complexities of environment. The complexities can be seen as follows :

Level 1:

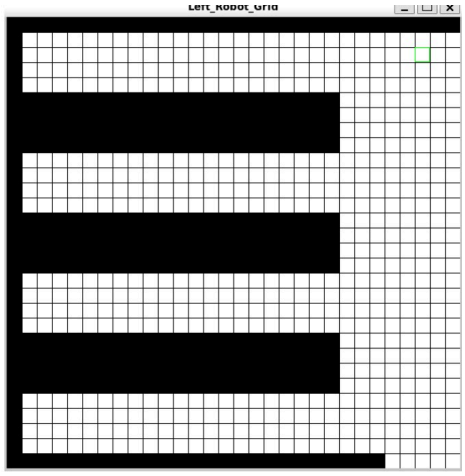


Figure 51 : Map Complexity Level 1

Level 2:

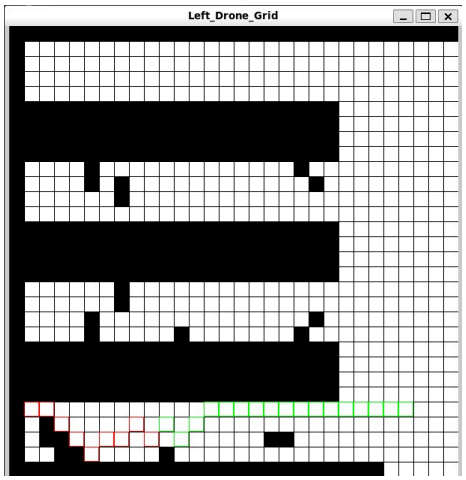


Figure 52 : Map Complexity Level 2

Level 3:

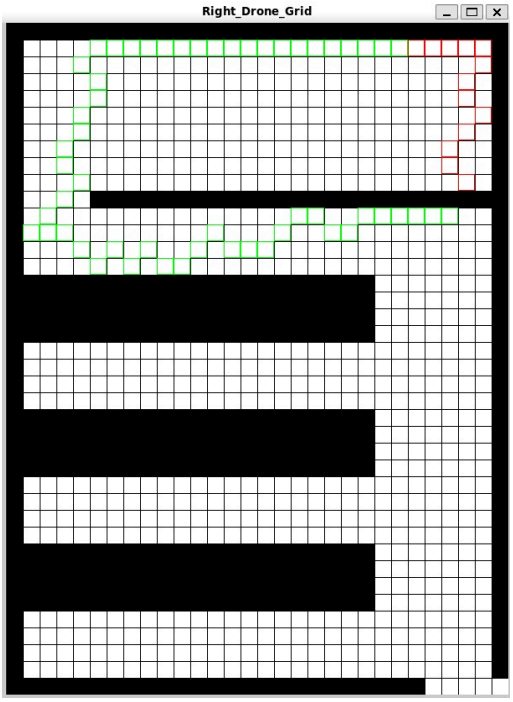


Figure 53 : Map Complexity Level 3

Level 4:

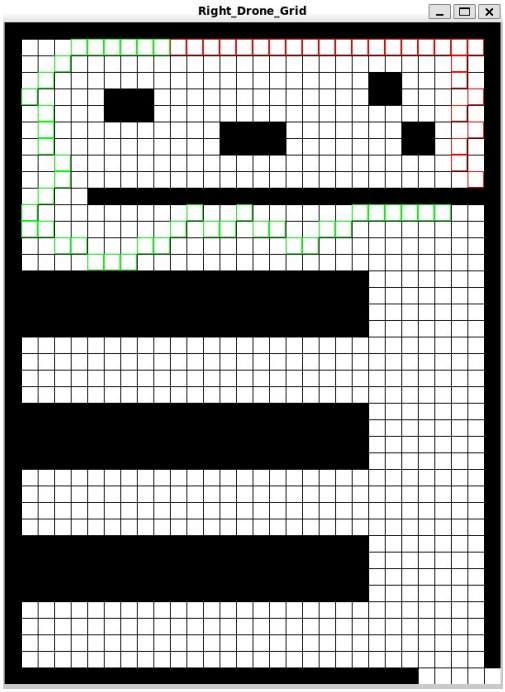


Figure 54: Map Complexity Level 4

For these different levels of environment , 3 Algorithms , namely Exploratory RRT ,

Frontier Based SLAM and Potential Field Based Navigation .

The following graphs shows the data recorded when the algorithms were run on the maps during test runs , comparing the distance traveled as well as the time taken for exploration during the runs :

1) Distance Comparisons

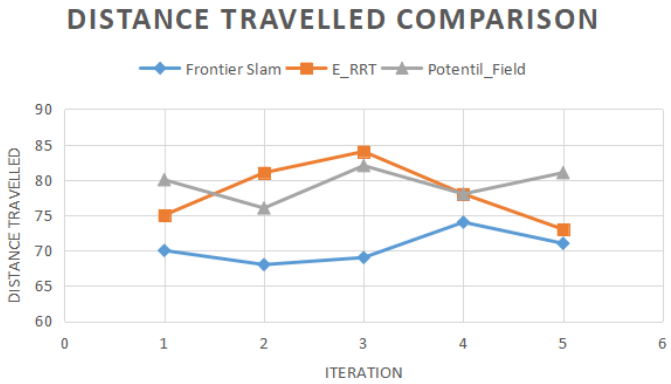


Figure 55 : Distance Covered Comparison for First Complexity Environment

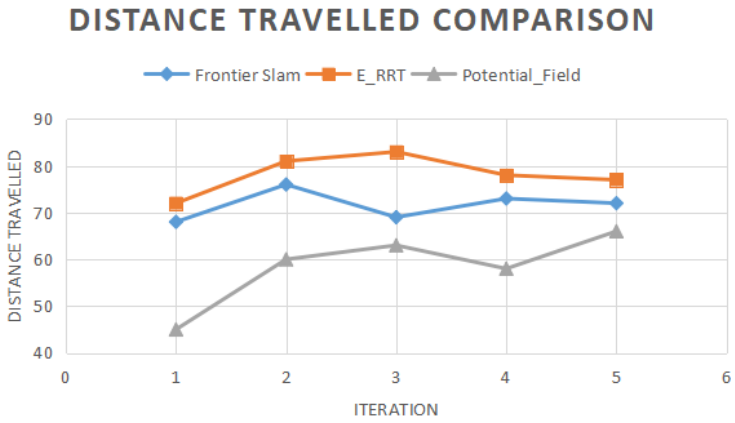


Figure 56 : Distance Covered Comparison for Second Complexity Environment

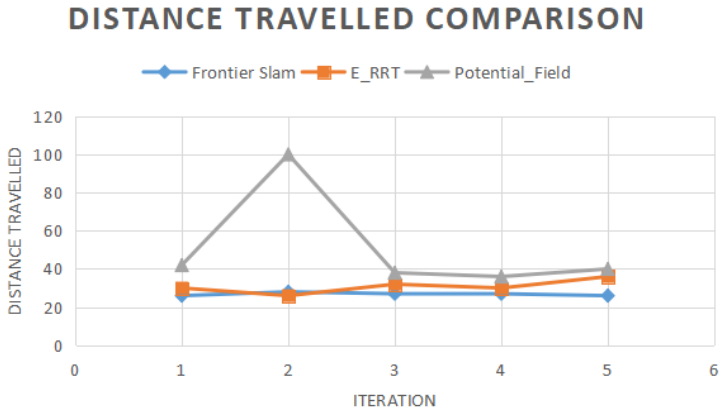


Figure 57 : Distance Covered Comparison for Third Complexity Environmen

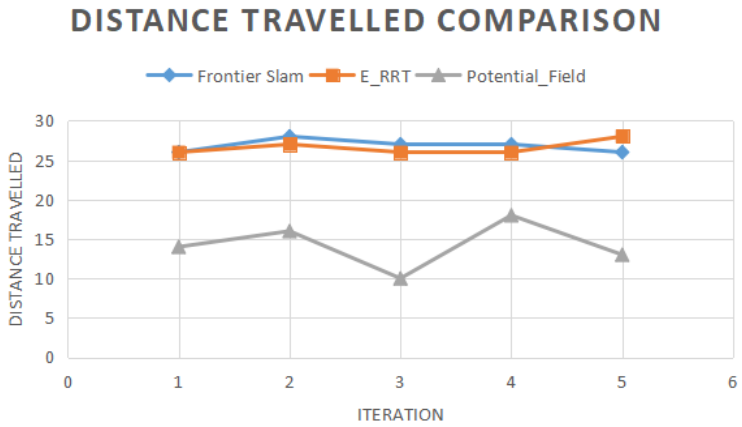


Figure 58 : Distance Covered Comparison for FourthComplexity Environment

2) Time Comparisons

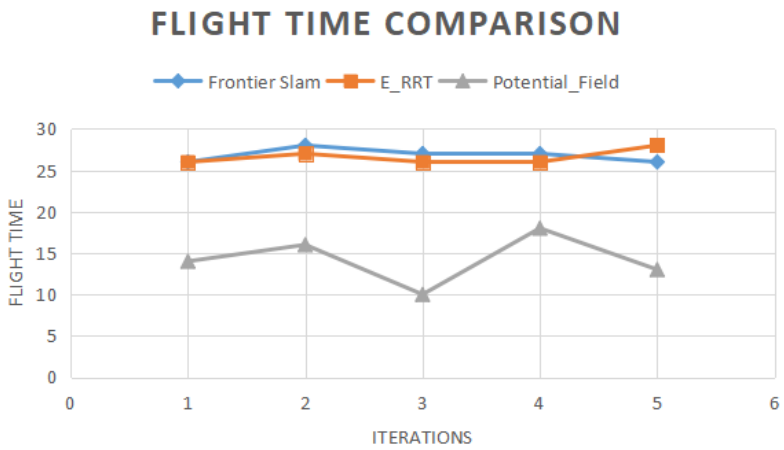


Figure 59: Flight Time Comparison for First Complexity Environment

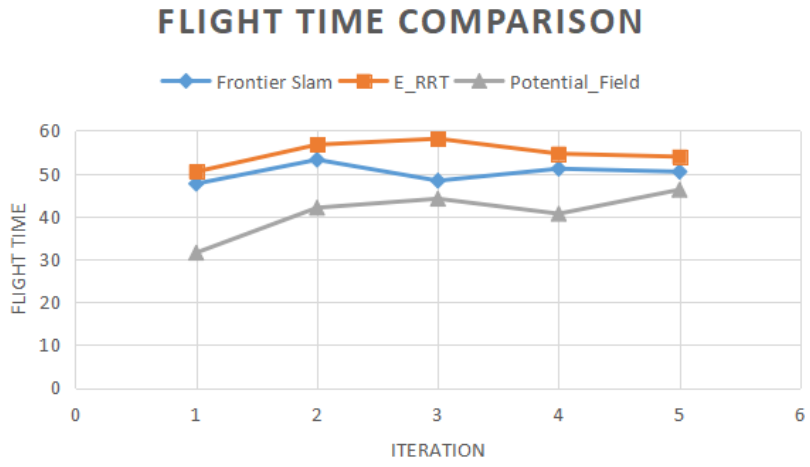


Figure 60 : Flight Time Comparison for Second Complexity Environment

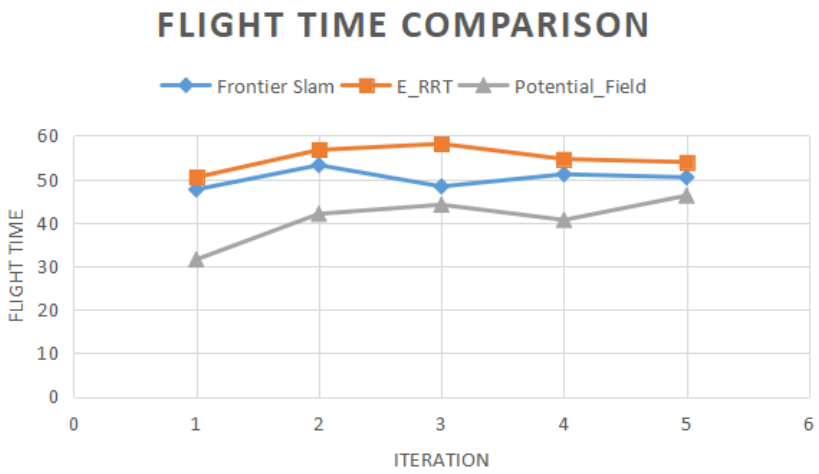


Figure 61 :Flight Time Comparison for Third Complexity Environment

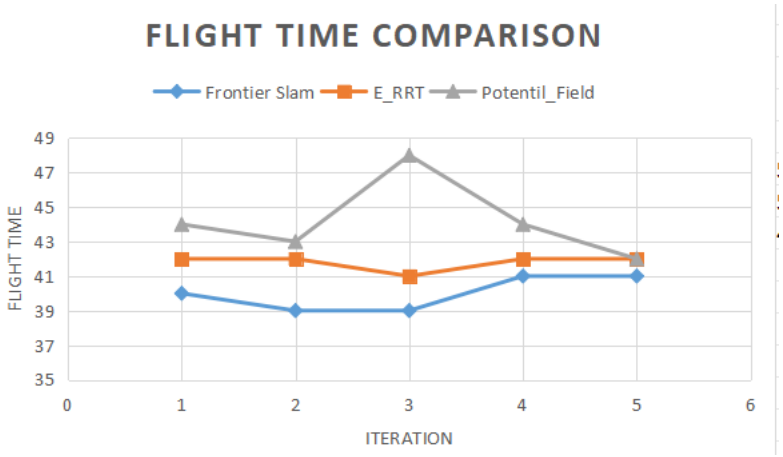


Figure 62 : Flight Time Comparison for Fourth Complexity Environment

MAP GENERATED FOR ALL COMPLEXITIES :

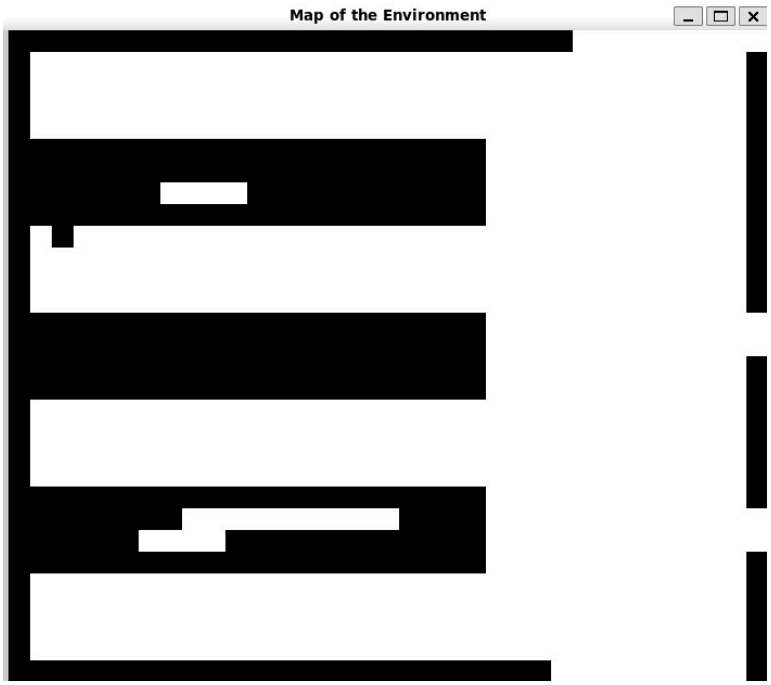


Figure 63 : Map Generated by Exploration for Complexity 1

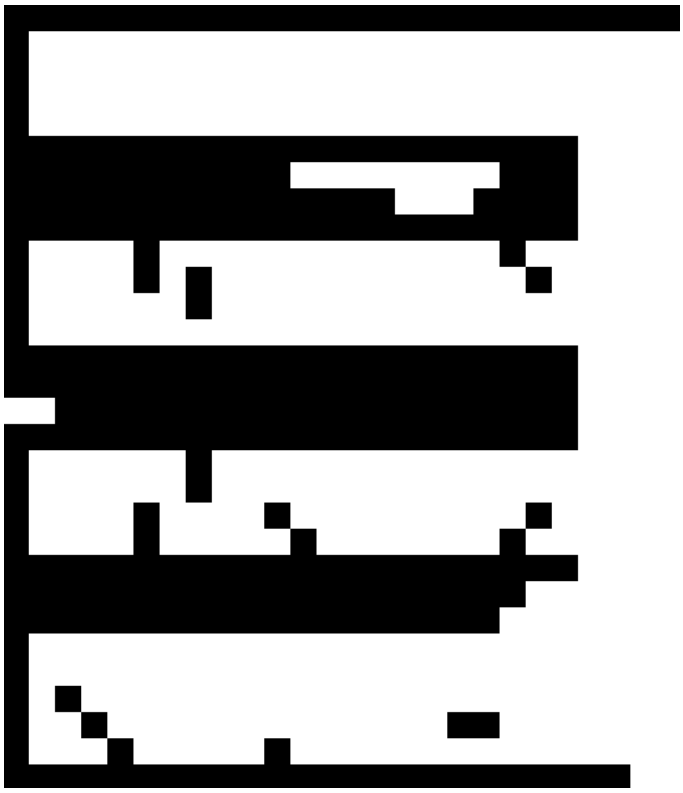


Figure 64 : Map Generated by Exploration for Complexity 2

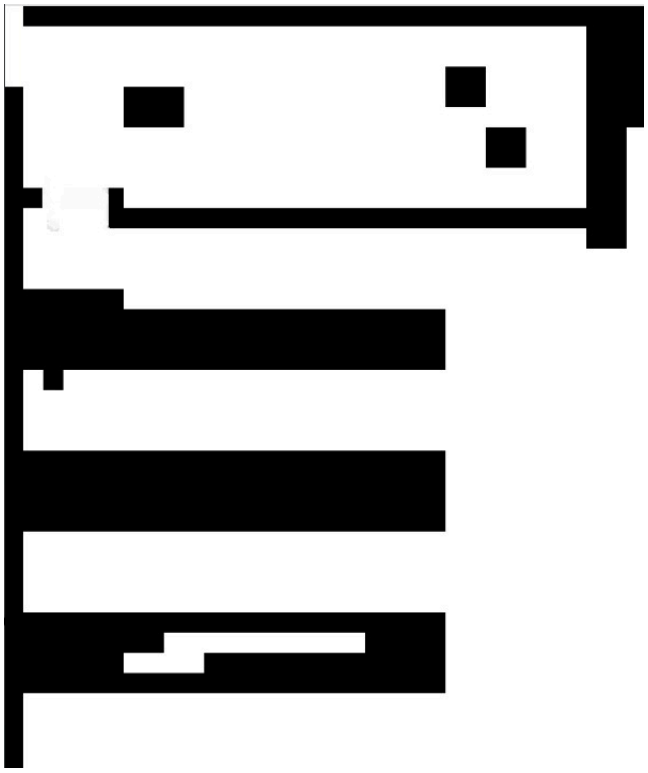


Figure 65 : Map Generated by Exploration for Complexity 3

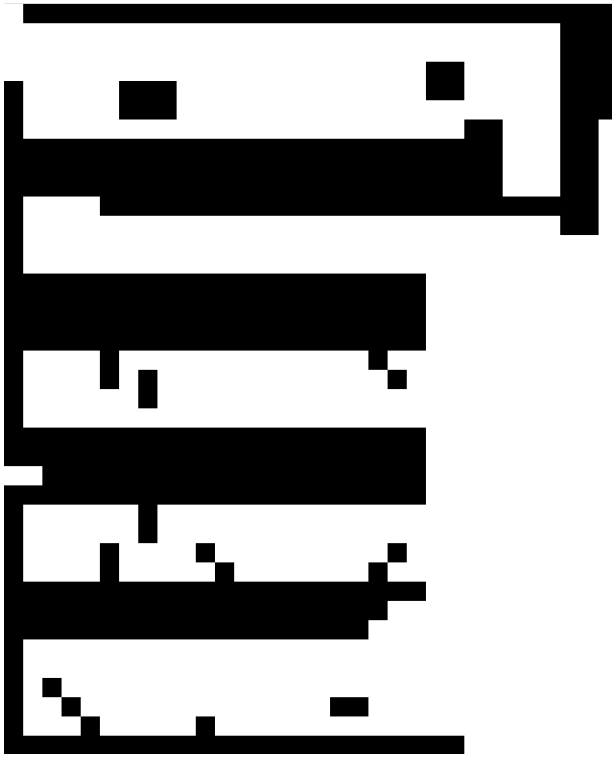


Figure 66 : Map Generated by Exploration for Complexity 4

CHAPTER 5
CONCLUSION AND FUTURE SCOPE

There is a strong path forward for future research in the area of drone swarm mapping and navigation in GPS-denied interior situations, with the goal of improving scalability, testing in various circumstances, and expanding capabilities to outdoor settings. In light of the growing significance of drones in a range of sectors, including construction, infrastructure inspection, and disaster relief, it is critical to tackle these obstacles in order to ensure their widespread implementation and efficiency.

First and foremost, scalability is an important factor to take into account as drone swarms get bigger and more complicated. There are many obstacles in the way of scaling the system to support more drones while still ensuring effective coordination, communication, and resource use. Subsequent research endeavors in this field may concentrate on enhancing algorithms for swarm behavior, distributing load among numerous drones, and decentralized decision-making procedures. Furthermore, by guaranteeing effective resource usage and resilience to faults within the swarm, investigating strategies like hierarchical structure and dynamic task allocation can aid in scaling.

Second, in order to guarantee the resilience and flexibility of the drone swarm navigation system, testing in various conditions is crucial. While early development might take place in controlled indoor settings, real-world deployment frequently involves erratic weather and a variety of terrains. Testing across a range of interior environments, including factories, warehouses, and office buildings, enables the validation of algorithms and methods in many contexts. Moreover, expanding testing to outside settings brings new difficulties such as unpredictable weather, intricate terrain, and GPS errors. Through the use of simulation settings and field testing, scientists may evaluate how well the system performs in a variety of scenarios and adjust algorithms as necessary.

Both fascinating potential and challenging situations arise when drone navigation is extended to outside locations. Drones can navigate autonomously in areas where GPS signals are unstable or unavailable by implementing GPS-less navigation techniques like optical odometry and simultaneous localization and mapping (SLAM). In particular, visual SLAM approaches have the ability to attain the level of accuracy and precision necessary for effective and safe outdoor navigation. Drones may create and update real-time maps of their surroundings by incorporating sensor data from cameras, inertial measurement units (IMUs), and other sensors. This allows for reliable navigation even in difficult outdoor conditions.

As a summary, future research in the field of drone swarm navigation and mapping in GPS-denied indoor environments will focus on scalability enhancements, testing in various environments, extending the application to outdoor scenarios, integrating real 3D point clouds for map generation, and applying visual SLAM techniques. Researchers can improve drone swarm capabilities and open the door for their broad use in a variety of applications, from infrastructure inspection to search and rescue operations, by tackling these obstacles.

REFERENCES

[1]. Amala Arokia Nathan, R.J., Kurmi, I. & Bimber, O. Drone swarm strategy for the detection and tracking of occluded targets in complex environments. *Commun Eng* 2, 55 (2023).

[2].MitchCampion, PrakashRanganathan, and SalehFaruque. 2019. UAV swarm communication and control architectures: a review. *Journal of Unmanned Vehicle Systems*. 7(2): 93-106.

[3]. Y. Huang, S. Wu, Z. Mu, X. Long, S. Chu and G. Zhao, "A Multi-agent Reinforcement Learning Method for Swarm Robots in Space Collaborative Exploration," *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*, Singapore, 2020, pp. 139-144, doi: 10.1109/ICCAR49639.2020.9107997.

[4]. Yasin JN, Mahboob H, Haghbayan M-H, Yasin MM, Plosila J. Energy-Efficient Navigation of an Autonomous Swarm with Adaptive Consciousness. *Remote Sensing*. 2021; 13(6):1059. https://doi.org/10.3390/rs13061059

[5]. Katada Y, Hasegawa S, Yamashita K, Okazaki N, Ohkura K. Swarm Crawler Robots Using Lévy Flight for Targets Exploration in Large Environments. *Robotics*. 2022; 11(4):76. https://doi.org/10.3390/robotics11040076

[6] H. D. K. Motlagh, F. Lotfi, H. D. Taghirad and S. B. Germi, "Position Estimation for Drones based on Visual SLAM and IMU in GPS-denied Environment," *2019 7th International Conference on Robotics and Mechatronics (ICRoM)*, Tehran, Iran, 2019, pp. 120-124, doi: 10.1109/ICRoM48714.2019.9071826.

[7] Amala Arokia Nathan, R.J., Kurmi, I. & Bimber, O. Drone swarm strategy for the detection and tracking of occluded targets in complex environments. *Commun Eng* 2, 55 (2023). https://doi.org/10.1038/s44172-023-00104-0

[8] Oršulić, J., Miklić, D., & Kovačić, Z. (2019). Efficient Dense Frontier Detection for 2-D Graph SLAM Based on Occupancy Grid Submaps. *IEEE Robotics and Automation Letters*, *4*(4), 3569–3576. https://doi.org/10.1109/LRA.2019.2928203

[9] V, D., Sharma K, R., & Kumar T, G. (2013). Comparative Study of Algorithms for Frontier based Area Exploration and Slam for Mobile Robots. *International Journal of Computer Applications*, *77*(8), 37–42. https://doi.org/10.5120/13417-1086

[10] Harik, E. H. C., & Korsaeth, A. (2018). Combining hector SLAM and artificial potential field for autonomous navigation inside a greenhouse. *Robotics*, *7*(2). https://doi.org/10.3390/robotics7020022