Efficient Mapping and Navigation System for Weed Removal Robot in Confined Garden Spaces

by

Mohammed Elkhatib

A Thesis

in

The Department

of

Mechanical, Industrial & Aerospace Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Mechanical Engineering)

at

Concordia University

Montréal, Québec, Canada

February 2025

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Mohammed Elkhatib

Entitled:

Efficient Mapping and Navigation System for Weed Removal Robot in Confined Garden Spaces

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Mechanical Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

Christopher Yee Wong Chair

Chunjiang An Examiner

Christopher Yee Wong Examiner

Wen Fang Xie Supervisor

Approved by

Department of Mechanical, Industrial & Aerospace Engineering

March 2025

Abstract

Efficient Mapping and Navigation System for Weed Removal Robot in Confined Garden Spaces

Mohammed Elkhatib

Autonomous navigation and mapping technologies are reshaping how robots interact with their surroundings, enabling a wide range of applications. This thesis introduces an autonomous mapping and navigation system for a mobile robot tailored for weed control in confined, outdoor garden environments. Unlike indoor robots that often rely on joystick-based manual mapping, the proposed system is fully automated, delivering a seamless, user-friendly setup experience optimized for backyard use. The solution leverages **Google Cartographer** for real-time **SLAM** and **AMCL** for adaptive localization. To optimize exploration coverage, the robot uses a combination of **random exploration** for initial mapping and **structured exploration** to target unexplored areas effectively. The integrated **A* algorithm** ensures efficient path planning and reliable obstacle avoidance throughout navigation. Extensive simulations and real-world testing demonstrate the robot's ability to autonomously map and navigate complex backyard layouts with minimal human intervention. The system shows resilience to dynamic obstacles, sensor limitations, and uneven terrain, confirming its robustness and practical utility. A significant contribution of this research is the development of a **fully autonomous, modular navigation framework** that removes the need for manual setup while ensuring high-accuracy mapping. By simplifying navigation in small-scale, unstructured outdoor environments, this work provides a functional and scalable solution for backyard maintenance and extends the applicability of autonomous robotics beyond controlled indoor settings. This thesis highlights how integrating SLAM, adaptive exploration, and planning can provide effective autonomy for lawn care, contributing to innovation in outdoor robotic systems.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my parents for their unwavering support, endless encouragement, and unconditional love throughout this journey. Their belief in me has been my greatest source of motivation.

I am profoundly grateful to my supervisor, Dr. Wen-Fang Xie, for her invaluable guidance, continuous support, and insightful feedback throughout the course of this research. Her expertise and encouragement have been instrumental in shaping this work.

I am also immensely grateful to my sister, whose guidance and encouragement have been invaluable. Her support has been a pillar of strength throughout this process.

A special thanks to my beloved wife, whose patience, understanding, and constant encouragement have been instrumental in completing this work. Her unwavering support has made this journey smoother and more meaningful.

I would also like to extend my heartfelt appreciation to my dear friends, Mohammed Mahmoud, Islam Bahaa,Lekaa Elaraby and Radwa Etba, for their invaluable friendship, motivation, and support. Whether through direct contributions to this project or through their unwavering spiritual encouragement, they have played a significant role in this journey. Their belief in my abilities and constant encouragement have meant the world to me.

To all those who have supported me in any way throughout this journey, I sincerely thank you.

Contents

List of Figures . . . . . **ix**

List of Tables . . . . . **xi**

List of Listings . . . . . **xii**

List of Abbreviations . . . . . **xiii**

1 Introduction **1**

1.1 Overview 1

1.2 Significance of Mapping and Navigation 1

1.3 Motivation and Objectives 2

1.3.1 Challenges in Backyard Gardens and Lawns 2

1.3.2 Leveraging Research and Innovation 3

1.4 Scope and Contributions 3

1.5 Thesis Structure 4

2 Literature Review **5**

2.1 Overview 5

2.2 Autonomous Navigation Systems 5

2.3 Sensor Selection and System Design 6

2.3.1 Rplidar A3 Sensor 6

2.3.2 System Integration and Real-Time Processing 7

2.4 LiDAR-Based Mapping and Navigation 8

2.4.1 SLAM for Autonomous Navigation 8

2.4.2 SLAM in Unstructured Outdoor Spaces 9

2.5 User-Centric Design in Robotics 10
2.6 Technology and Methodology 11
2.6.1 Automated Navigation for Accessibility 11
2.6.2 LiDAR-Based SLAM for Mapping and Localization . . . 11
2.6.3 The A* Algorithm in Path Planning 12
2.6.4 Comparisons with Alternative Approaches 13
2.7 Conclusion 13

3 System Design and Methodology 14
3.1 Introduction 14
3.1.1 Hardware Setup 15
3.1.2 SLAM Implementation 16
3.1.3 Mapping and Localization 17
3.2 Path Planning Using the A* Algorithm 20
3.2.1 Grid-Based Path Planning with A* 20
3.2.2 Handling Dynamic Obstacles in Path Planning 21
3.2.3 Dynamic Replanning in A* 22
3.3 System Integration 23
3.3.1 Workflow of Autonomous Navigation 23
3.3.2 System Communication Flow 25
3.4 Conclusion 26

4 Implementation 27
4.1 Introduction 27
4.2 Initialization 27
4.2.1 Process 28
4.2.2 Explanation of the Initialization Phase 31
4.3 Mapping Phase 35
4.3.1 Real-Time Map Generation with Cartographer 36
4.3.2 Map Output and Analysis 37

4.3.3 Mapping Workflow and Applications 39
4.3.4 Final Mapping Stage 40
4.4 Discovering Phase 41
4.4.1 Scouting (Structured Discovering) 42
4.4.2 Explanation and Advantages of Discovering Strategy . . 52
4.5 Navigation Phase 54
4.5.1 Introduction to Navigation 54
4.5.2 Using the A* Package for Navigation 55
4.5.3 Navigation Explanation: Path Planning, Real-Time Updates, and Obstacle Avoidance 57
4.6 Obstacle Avoidance Using LiDAR Data 59
4.6.1 Global and Local Planners in Navigation 60
4.6.2 Obstacle Avoidance Workflow 62
4.7 Challenges and Optimizations in Implementation 63
4.8 Conclusion 65

5 Testing and Results 66
5.1 Simulation Results 66
5.1.1 Scouting and Mapping 66
5.1.2 Testing in a Second Map 67
5.2 Real-World Testing 68
5.2.1 Indoor Laboratory Experiments 68
5.2.2 Outdoor Garden Experiment 69
5.2.3 Metrics for Comparison 71
5.2.4 Simulation vs. Real-World Mapping 71
5.2.5 Overall Comparison 73
5.2.6 Insights and Observations 73
5.2.7 Key Findings and Advantages 76
5.2.8 Final Performance Summary 77
5.3 Conclusion 77

6 Conclusion and Future Work 78

6.1 Conclusion 78

6.2 Future Work 79

A Scouting Functionality 88

B Navigation and Obstacle Avoidance 96

C Watchtower Node for Scouting Progress Monitoring 102

D Launch Description for Mapping and Visualization 106

E Manager Node for Navigation and Scouting 110

F Launch File for Navigating and Scouting Nodes 118

List of Figures

List of Figures

2.1 Rplidar A3 Sensor used for mapping and navigation. 7

3.1 The autonomous robot 15
3.2 A* Pathfinding Visualization 22
3.3 Overview of System Communication Flow 26

4.1 System initialization workflow. 31
4.2 Workflow incorporating the Mapping Phase. 35
4.3 Initial mapping stage. 40
4.4 Mid-mapping stage. 40
4.5 Final mapping stage. 41
4.6 Structured Discovering strategy. 52
4.7 Structured Discovering Strategy. 54
4.8 Obstacle avoidance process: (a) initial detection, (b) path adjustment, and (c) execution of the new trajectory. 59
4.9 Navigation Phase Flowchart. 62
4.10 Obstacle avoidance workflow. 63

5.1 Robot's path during the mapping process. 67

5.2 Occupancy grid map from the first simulation. 67
5.3 Robot navigating a complex environment. 68
5.4 Final occupancy grid map from the second simulation. 68
5.5 Robot scouting in the laboratory. 69
5.6 Occupancy grid map from the laboratory experiment. 69
5.7 Comparison between the SLAM-generated occupancy grid and the actual aerial view of the backyard garden. 70
5.8 Comparison of mapping accuracy over multiple test iterations. . 75
5.9 Comparison of path efficiency in terms of deviation from the optimal trajectory. 75
5.10 Comparison of localization error between Santos et al. (2015) and the proposed system. 76

List of Tables

List of Tables

2.1 Comparison of LiDAR Options 7
2.2 Comparison of SLAM vs. Alternatives 12
2.3 Comparison of A* with Alternative Path Planning Algorithms . 13
2.4 Comparison of AMCL with Alternative Localization Methods . 13

3.1 Key Hardware Components and Specifications 15

5.1 General Mapping and Localization Performance Across Environments 73
5.2 Environmental Impact and System Efficiency Metrics 73

Listings

4.1 Watchtower Map Exploration Logic 30

4.2 XML Model Configuration Example 32

4.3 Real Robot Configuration Example 33

4.4 Rplidar Node Initialization 34

4.5 Jetson Nano SSH and ROS Launch Commands 34

4.6 Cartographer Node Configuration 37

4.7 Watchtower Map Discovering Logic 38

4.8 Frontier Detection Algorithm (Simplified) 43

4.9 Frontier Clustering with DBSCAN 45

4.10 Example Frontier Clustering Output 46

4.11 Listener Callback Function 50

A.1 Scouting Server Implementation 88

B.1 Navigation and Obstacle Avoidance Logic 96

C.1 Modified Watchtower Node Implementation for Monitoring Scouting Progress 102

D.1 Modified Launch File for Cartographer and RViz Integration . . 106

E.1 Modified Manager Node for Autonomous Navigation and Scouting 110

F.1 Modified Launch File for Navigating and Scouting 118

List of Abbreviations

Abbreviation	Definition
A*	A-Star (Pathfinding Algorithm)
AI	Artificial Intelligence
AMCL	Adaptive Monte Carlo Localization
BMS	Battery Management System
CPU	Central Processing Unit
DWA	Dynamic Window Approach
GPS	Global Positioning System
GPU	Graphics Processing Unit
ICP	Iterative Closest Point (Algorithm for Scan Matching)
IMU	Inertial Measurement Unit
LiDAR	Light Detection and Ranging
MAE	Mean Absolute Error
ML	Machine Learning
RRT	Rapidly-exploring Random Tree
RMSE	Root Mean Square Error
ROS	Robot Operating System
Rplidar	Rotating Planar LiDAR
SD	Standard Deviation
SLAM	Simultaneous Localization and Mapping
NVIDIA	Company producing Jetson Nano hardware

Chapter 1

Introduction

1.1 Overview

Autonomous systems have transformed routine tasks, but many still require complex initial setups that deter user adoption. This project addresses that gap by developing a user-friendly robot capable of autonomous mapping and navigation in backyard gardens and lawns, minimizing user input while maximizing efficiency. Unlike traditional systems that often depend on manual navigation for environment mapping, this approach emphasizes simplicity and accessibility, aligning with modern trends in user-centered robotics. The system leverages LiDAR-based SLAM (Simultaneous Localization and Mapping) to efficiently map confined spaces like gardens and lawns—environments that pose unique challenges due to their irregular layouts and natural obstacles.

1.2 Significance of Mapping and Navigation

Effective mapping and navigation are crucial for autonomous systems, enabling robots to perceive their environment, avoid obstacles, and plan efficient paths. In confined and irregular spaces like backyard gardens, these capabilities are even more critical due to challenges such as uneven terrain, dense vegetation, and dynamic obstacles [17].

This project utilizes the Rplidar A3 to support 2D SLAM through Google Cartographer, focusing on the SLAM process rather than the LiDAR hardware itself. While 2D SLAM effectively handles flat terrains, its lack of vertical data can impact localization stability in complex outdoor environments [4]. This limitation becomes especially significant in unstructured settings, where natural obstacles and terrain irregularities affect mapping accuracy.

To address these challenges, the system integrates Cartographer SLAM with adaptive navigation strategies. The robot employs the Scouter method for targeted mapping of unexplored regions. This strategy is complemented by the A* path planning algorithm, which optimizes route selection and obstacle avoidance. This combination enables the robot to navigate confined and dynamic environments more effectively, pushing the boundaries of traditional 2D SLAM applications.

1.3 Motivation and Objectives

This project addresses the need for user-friendly autonomous mobile robots by developing a system that can independently map and navigate backyard gardens and lawns without requiring complex setup or technical expertise. While existing robots, like robotic vacuum cleaners and smart assistants, offer navigation features, they often demand time-consuming initial mapping, deterring non-technical users. The goal is to create an autonomous robot that simplifies user interaction while assisting with tasks such as identifying and removing weeds, making it both practical and accessible for everyday use.

1.3.1 Challenges in Backyard Gardens and Lawns

Backyard gardens, with their small spaces, uneven surfaces, and unique layouts, present specific challenges for robots. Unlike open or structured environments, gardens and lawns need precise mapping, the ability to deal with unexpected obstacles, and smooth operation without constant human input. To tackle this,

the project focuses on using LiDAR technology, particularly the Rplidar A3, which specializes in accurate, real-time 2D mapping. This makes it a perfect fit for small, confined areas like gardens, ensuring that the robot can adapt and navigate effectively.

1.3.2 Leveraging Research and Innovation

While LiDAR assists in generating 2D maps, the primary focus of this research is on the SLAM algorithm, particularly Cartographer, which enables real-time mapping and localization in confined outdoor environments [22]. The emphasis is placed on the mapping and localization process itself, rather than the LiDAR hardware, highlighting the critical role of SLAM in achieving accurate and efficient navigation.

Although this study employs a LiDAR-based SLAM approach, alternative methods like visual SLAM, such as ORB-SLAM, have also demonstrated strong performance in complex environments by utilizing camera data for feature-rich mapping and localization. These methods can offer advantages in scenarios where rich visual information is available.

While techniques like GPS and LiDAR SLAM have been widely used for large-scale outdoor navigation [5], this project adapts these principles for smaller, more confined environments like backyard gardens. By tailoring the SLAM process and integrating efficient exploration and navigation strategies, the system showcases the potential of robotics to simplify everyday tasks and improve daily life.

1.4 Scope and Contributions

This project presents an autonomous robot that uses the Rplidar A3 and Cartographer SLAM for real-time 2D mapping and navigation in GPS-denied backyard environments. An exploration strategy structured mapping—enhances coverage, while the A* algorithm and AMCL(Adaptive Monte Carlo Localization) ensure

efficient path planning and accurate localization.

Validated through simulations and real-world tests, the system adapts to uneven terrain, dynamic obstacles, and irregular layouts, advancing 2D SLAM applications in unstructured outdoor spaces and promoting accessible autonomous navigation for small-scale environments.

This research to be submitted for presentation at IROS 2025, showcasing its novel mapping and navigation methods and contributing to advancements in robotics for confined and unstructured environments.

1.5 Thesis Structure

This thesis is structured into six chapters, detailing the development and validation of the autonomous garden mapping and navigation system:

- **Chapter 1: Introduction** – Outlines the motivation, challenges, and objectives of the research.
- **Chapter 2: Literature Review** – Examines SLAM, navigation systems, and robotics advancements.
- **Chapter 3: System Design and Methodology** – Describes the hardware, software, and algorithm development.
- **Chapter 4: Implementation** – Covers hardware-software integration and system assembly.
- **Chapter 5: Testing and Results** – Evaluates mapping accuracy, navigation efficiency, and usability.
- **Chapter 6: Conclusion and Future Work** – Summarizes findings, contributions, and future directions.

Chapter 2

Literature Review

2.1 Overview

Despite advancements in autonomous robotics, efficient mapping and navigation in small-scale, unstructured outdoor spaces—like backyard gardens—remain underexplored. Existing LiDAR-based SLAM solutions often target large-scale or indoor environments, requiring complex setups that limit accessibility for non-technical users.

This chapter focuses on addressing these gaps by leveraging 2D SLAM with Cartographer and the Rplidar A3, optimized for confined outdoor spaces. It also explores user-centric design strategies to minimize setup complexity, ensuring intuitive use while maintaining mapping accuracy. By combining lightweight hardware with the Scouter strategy for adaptive navigation, this project advances practical, user-friendly autonomous navigation for dynamic garden environments.

2.2 Autonomous Navigation Systems

Navigating confined outdoor spaces like backyard gardens involves challenges such as uneven terrain, dynamic obstacles, and dense vegetation. LiDAR (Light Detection and Ranging) is preferred over stereo cameras and ultrasonic sensors for its consistent depth data and resilience to lighting conditions [27]. Coupling LiDAR

with inertial measurement units (IMUs) further improves localization stability on irregular terrains [15].

Multi-sensor systems like GPS-LiDAR fusion are effective in large-scale environments but struggle in confined spaces due to signal obstructions [35]. Similarly, combining stereo cameras with ultrasonic sensors adds unnecessary complexity for small-scale applications [33].

This project uses the Rplidar A3 with Cartographer SLAM for accurate 2D mapping in GPS-denied environments. Though 2D LiDAR has limitations in capturing vertical features, its integration with AMCL for localization and A* for path planning enables reliable navigation in dynamic, irregular garden layouts.

2.3 Sensor Selection and System Design

This project employs the Rplidar A3 for real-time 2D mapping and localization, balancing cost, precision, and computational efficiency—key factors for confined outdoor spaces like backyard gardens. Its 0.5° resolution, 25-meter range, and low power consumption make it ideal for small-scale navigation without the complexity or expense of 3D LiDAR systems [34]. By relying on a single 2D LiDAR, the system avoids the high computational demands of multi-sensor setups while maintaining accurate mapping and obstacle detection.

2.3.1 Rplidar A3 Sensor

The Rplidar A3 offers a cost-effective solution for precise 2D mapping and real-time obstacle detection in small, unstructured outdoor spaces. Its lightweight design simplifies integration into compact robotic platforms, while its range and resolution ensure detailed map generation, even in cluttered environments. Compared to 3D LiDAR systems, which are overengineered for confined spaces, and ultrasonic sensors, which lack mapping precision, the Rplidar A3 provides an optimal balance of accuracy, efficiency, and affordability [40, 41].

Table 2.1: Comparison of LiDAR Options

Feature	**Rplidar A3**	**3D LiDAR Systems**	**Ultrasonic Sensors**
Mapping Capability	2D mapping	3D mapping	Basic obstacle detection
Range	Up to 25m	Up to 100m	Limited (up to 4m)
Precision	High (0.5° resolution)	Very High	Low
Power Consumption	Low	High	Very Low
Computational Demand	Moderate	High	Minimal
Cost	Affordable	Expensive	Very Low
Suitability	Ideal for small, flat spaces	Overengineered for gardens	Insufficient for detailed mapping

As shown in Table 2.1, the Rplidar A3 offers a practical compromise between mapping precision and system simplicity. While 3D LiDARs provide high-resolution data, their cost and computational demands are excessive for confined gardens. In contrast, ultrasonic sensors, though inexpensive, lack the resolution for accurate mapping [41].

Figure 2.1: Rplidar A3 Sensor used for mapping and navigation.

2.3.2 System Integration and Real-Time Processing

The Rplidar A3 integrates seamlessly with Cartographer SLAM for real-time 2D mapping and AMCL (Adaptive Monte Carlo Localization) for continuous localization. Its lightweight design minimizes mechanical load, while its range and precision ensure reliable navigation through irregular outdoor spaces. The system also incorporates RViz for real-time visualization, enabling on-the-fly assessment of mapping accuracy and navigation performance during both testing and deployment [20].

To reduce user involvement, the system eliminates the need for manual map creation—an issue common in many autonomous systems—allowing the robot to autonomously explore and adapt to varying garden layouts. The flexible exploration strategy enables structured mapping, while the A* algorithm optimizes path planning and dynamic obstacle avoidance [18].

By focusing on simplicity, affordability, and efficiency, this system addresses a gap in existing autonomous navigation research. It offers a scalable solution for confined outdoor spaces, providing reliable mapping and navigation without the complexity of multi-sensor systems [34].

2.4 LiDAR-Based Mapping and Navigation

To enable precise navigation in unstructured outdoor environments like backyard gardens, this project employs Simultaneous Localization and Mapping (SLAM) alongside LiDAR (Light Detection and Ranging) for real-time mapping and localization. LiDAR provides consistent, high-resolution depth data, ensuring accurate obstacle detection and navigation even in complex, dynamic spaces [14, 19]. Its resilience to lighting conditions and environmental variations makes it preferable over stereo cameras and ultrasonic sensors in outdoor settings.

While 3D LiDAR offers richer spatial data, its higher computational demands make it impractical for small-scale, cost-sensitive applications. Instead, lightweight 2D LiDAR systems, such as the Rplidar A3 used in this project, offer a balanced solution—minimizing computational overhead while maintaining mapping precision [15, 29]. This makes 2D LiDAR ideal for confined gardens, where efficiency and adaptability are essential.

2.4.1 SLAM for Autonomous Navigation

SLAM is fundamental for autonomous robots navigating unknown or GPS-denied environments. By integrating LiDAR with SLAM algorithms, the system builds

accurate maps in real time while maintaining continuous localization, even in dynamic settings [15, 26]. Key advancements like loop-closure detection reduce localization drift, improving long-term mapping reliability [21].

In this project, Cartographer SLAM handles real-time 2D mapping, while Adaptive Monte Carlo Localization (AMCL) ensures consistent positioning. The A* path planning algorithm complements this by optimizing routes and enabling dynamic obstacle avoidance [18]. This integration enables the robot to handle irregular terrains, moving obstacles, and varying garden layouts without manual intervention.

2.4.2 SLAM in Unstructured Outdoor Spaces

Unstructured outdoor environments, such as gardens, pose challenges like uneven terrain, dense vegetation, and dynamic obstacles. LiDAR-based SLAM excels in these conditions, offering high-resolution spatial awareness unaffected by lighting changes or GPS loss [5, 33].

While 3D LiDAR provides detailed mapping, its processing demands are excessive for small-scale applications. The Rplidar A3 balances range, precision, and efficiency, making it ideal for confined outdoor spaces [16]. Studies confirm that 2D LiDAR with SLAM ensures stable localization and accurate mapping in complex outdoor settings [5, 15].

Mapping algorithms and loop-closure detection enhance reliability by reducing drift and improving obstacle detection, particularly in feature-sparse gardens with repetitive patterns [26, 29].

This project validates 2D LiDAR-based SLAM as a lightweight, effective solution for mapping unstructured environments, offering a practical alternative to resource-intensive systems [33].

Handling Bushes, Grass, and Shrubbery with 2D LiDAR: 2D LiDAR, operating in a single plane, detects all objects at its scanning height, including bushes and grass, which may lead to false obstacle identifications. To mitigate

this, the system optimizes LiDAR placement and applies filtering techniques to differentiate persistent obstacles from transient vegetation. The algorithm prioritizes consistently detected objects and uses iterative scan integration to refine obstacle classification, ensuring reliable navigation in dense vegetation without misinterpreting dynamic elements like swaying grass.

2.5 User-Centric Design in Robotics

User-centric design is essential for the adoption of autonomous systems, especially in home and garden applications where simplicity, minimal setup, and accessibility for non-expert users are key. In confined environments like gardens, reducing user intervention remains a major challenge [34, 37]. Research in autonomous garden maintenance and weed management has shown that effective automation minimizes human effort while maintaining precision [3].

Commercial solutions, such as Husqvarna's Automower, use AI-driven navigation and GPS-based mapping but still require manual installation of boundary wires, limiting flexibility in dynamic environments [42, 43]. These additional setup steps create barriers for non-technical users.

This project overcomes such limitations by eliminating the need for GPS, external sensors, or boundary wires. Using a single 2D LiDAR sensor with SLAM-based mapping and adaptive path planning, the system offers fully autonomous operation with minimal user involvement. To further reduce the need for boundary wires, the system integrates the *Watchtower* monitoring Node, which tracks map coverage and exploration progress in real time. This not only reduces setup complexity and hardware costs but also enhances accessibility, making the robot suitable for users without technical expertise.

2.6 Technology and Methodology

2.6.1 Automated Navigation for Accessibility

Traditional autonomous systems often require manual mapping or extensive calibration, limiting accessibility for non-technical users. This project addresses these challenges by integrating Cartographer SLAM with the Rplidar A3 sensor for real-time 2D mapping and localization. While the Rplidar A3 provides raw spatial data, Cartographer processes it to build accurate maps, enabling the robot to explore irregular backyard layouts without predefined maps or GPS dependency [33, 35].

To handle environmental complexity, the system employs a flexible exploration strategy, structured exploration to maximize coverage. AMCL (Adaptive Monte Carlo Localization) refines positioning accuracy, while the A* algorithm ensures efficient path planning and dynamic obstacle avoidance. This combination reduces user involvement and adapts to varying outdoor layouts, bridging the gap between complex robotics research and practical, user-friendly navigation [3, 37].

2.6.2 LiDAR-Based SLAM for Mapping and Localization

LiDAR-based SLAM is central to this project, enabling the robot to autonomously generate maps and localize itself in real time, eliminating the need for prior user input. Unlike GPS-dependent methods that struggle in confined spaces, SLAM dynamically constructs maps of unknown surroundings [22]. Backyard gardens, with trees, bushes, and irregular pathways, present unique challenges where manual mapping or vision-based systems fall short due to lighting variability and computational demands [27].

This system leverages Cartographer SLAM for real-time mapping, with loop closure and map correction ensuring accuracy over time. The integration of RViz, a ROS-based visualization tool, allows real-time performance monitoring and efficient debugging during testing [3].

Comparison of SLAM Methods and Alternatives

Table 2.2 highlights the benefits of SLAM over pre-mapping and GPS-based methods, particularly for confined outdoor spaces.

Table 2.2: Comparison of SLAM vs. Alternatives

Criteria	**SLAM**	**Pre-Mapping**	**GPS-Based Methods**
Autonomy	Fully autonomous, real-time mapping.	Requires user-driven initial mapping.	Relies on external signals, lacks detail.
Accuracy	High spatial resolution, adaptive.	Dependent on user precision.	Limited in confined spaces.
Setup Complexity	Minimal; no prior mapping needed.	High; manual setup required.	Moderate; GPS calibration needed.
Adaptability	Dynamic; adjusts to changes in real time.	Static; fixed to initial mapping.	Ineffective in weak signal areas.

2.6.3 The A* Algorithm in Path Planning

The A* algorithm was selected for path planning due to its balance between efficiency and optimality, making it well-suited for structured outdoor spaces like backyard gardens. Using a cost function:

$$f(n) = g(n) + h(n) \tag{2.1}$$

where $g(n)$ is the actual cost from the start node to n and $h(n)$ is a heuristic estimate to the goal, A* efficiently computes near-optimal paths [16].

When integrated with SLAM, A* dynamically adapts to environmental changes, recalculating paths as the occupancy grid updates in real time. This enables the robot to handle dynamic obstacles and varying layouts effectively.

Compared to alternatives, A* offers a strong balance of speed and accuracy. Dijkstra's algorithm, while guaranteeing the shortest path, is computationally heavier, whereas Rapidly-exploring Random Tree (RRT) excels in complex spaces but often produces suboptimal paths [24,29]. The A* algorithm was validated using Gazebo simulations, where the robot successfully navigated virtual gardens, demonstrating efficient path generation and obstacle avoidance [16].

Table 2.3: Comparison of A* with Alternative Path Planning Algorithms

Algorithm	Strengths	Weaknesses
A*	Optimal paths with heuristic-driven efficiency.	Computational load increases with map size.
Dijkstra's	Guarantees shortest path.	High computational cost in large environments.
DWA	Effective in local obstacle avoidance.	Lacks global planning capability.
ACO	Adapts to dynamic environments.	Computationally heavy; suboptimal in complex grids.
RRT	Suitable for complex, high-dimensional spaces.	Non-optimal paths; needs post-processing.

2.6.4 Comparisons with Alternative Approaches

Table 2.4 compares AMCL with alternative localization methods, highlighting its balance of accuracy, computational efficiency, and real-time adaptability.

Table 2.4: Comparison of AMCL with Alternative Localization Methods

Method	Advantages	Limitations
AMCL	Real-time localization with particle filtering.	Less robust in feature-sparse environments.
Graph-Based	High long-term consistency.	Computationally heavy; not ideal for real-time.
Vision-Based	Effective in visually rich environments.	Sensitive to lighting and visual noise.

By combining Cartographer SLAM, A* path planning, and AMCL localization, this project delivers efficient, autonomous navigation tailored for small-scale, unstructured outdoor environments. This approach reduces user complexity while maintaining precision, cost-efficiency, and adaptability [17, 32, 36].

2.7 Conclusion

This chapter reviewed LiDAR-based mapping, SLAM techniques, and path planning, highlighting gaps in user accessibility, confined-space navigation, and real-time adaptability. By integrating Cartographer SLAM, the A* algorithm, and the Rplidar A3, this project delivers a cost-effective, fully autonomous solution for small-scale outdoor environments. The next chapters detail the system's design, implementation, and validation in complex garden layouts.

Chapter 3

System Design and Methodology

3.1 Introduction

This chapter outlines the architecture and methodology of the autonomous robotic system for backyard garden navigation, focusing on hardware-software integration, key algorithms, and implementation strategies for real-time mapping, localization, and navigation.

Building on concepts introduced in earlier chapters, this section details the application of SLAM, AMCL, and the A* algorithm within the system's design to ensure efficient mapping, accurate localization, and adaptive path planning.

Simulation tools like Gazebo and RViz support system development and testing—Gazebo enables physics-based simulations for sensor validation, while RViz offers real-time visualization of mapping and navigation.

By integrating advanced sensing, efficient algorithms, and robust simulations, this project delivers a cost-effective, user-friendly solution for autonomous navigation in confined outdoor spaces. The following sections break down the system's components and core methodologies.

3.1.1 Hardware Setup

The autonomous robotic system is optimized for backyard garden navigation, using lightweight, power-efficient, and cost-effective components. It integrates sensing, processing, actuation, and power modules to enable real-time mapping, localization, and navigation in confined outdoor spaces.

Figure 3.1: The autonomous robot

The robot features a compact structure (**60 cm** × **70 cm** × **55 cm**, **50 kg**) with a **tracked wheel system** for stability on uneven terrains like grass and dirt. A **rechargeable lithium-ion battery** powers the entire system, supporting up to **8 hours** of continuous operation.

Table 3.1: Key Hardware Components and Specifications

Component	**Function**	**Key Specifications**
Rplidar A3 (Sensor)	2D mapping and obstacle detection	0.5° resolution, 25m range, 360° scanning
NVIDIA Jetson Nano (Processor)	SLAM, AMCL, and A* algorithm execution	Quad-core ARM Cortex-A57, 128-core GPU, 4GB RAM, 5-10W power
Tracked Wheel System (Actuation)	Stable navigation on varied terrain	Dual-motor system, ROS2-controlled
Lithium-Ion Battery (Power)	Powers all components	12V, 20Ah, 6–8 hours runtime
ROS2 Middleware	Real-time data flow and communication	Modular, supports multi-node architecture

The Rplidar A3 enables precise 2D mapping and obstacle detection with a 0.5° resolution and a 25m range, ideal for confined outdoor spaces.

The NVIDIA Jetson Nano runs SLAM, AMCL, and A* path-planning algorithms within the ROS2 framework, balancing computational power and energy efficiency.

The tracked wheel system, controlled via ROS2, ensures stable movement over

uneven terrain, while the lithium-ion battery supports extended autonomous operation with efficient power distribution.

ROS2 middleware coordinates communication between sensors, processors, and actuators, ensuring seamless data flow and system scalability.

This streamlined hardware setup supports precise, autonomous navigation in GPS-denied environments, enabling efficient mapping and obstacle avoidance in small-scale outdoor spaces.

3.1.2 SLAM Implementation

The autonomous robotic system leverages ROS2 for efficient data exchange between the Rplidar A3, NVIDIA Jetson Nano, and actuators. Its modular architecture supports real-time mapping, localization, and navigation within confined outdoor spaces.

Cartographer SLAM processes 2D LiDAR data to construct and update occupancy grids, enabling the robot to map and localize simultaneously. Optimized parameters, including scan matching weight and submap resolution, balance accuracy and computational load, ensuring stable performance in complex environments.

Adaptive Monte Carlo Localization (AMCL) refines pose estimation (x, y, θ) using a particle filter that dynamically adjusts particle density based on environmental complexity. Particle weights are updated via Bayesian filtering:

$$w_t^i = p(z_t|x_t^i) \cdot w_{t-1}^i \tag{3.1}$$

This process enhances localization accuracy, even in cluttered or dynamic settings.

The A* algorithm (see Section 2.6.3) computes efficient navigation paths, updating routes in real time based on changes in the occupancy grid. This ensures continuous obstacle avoidance and adaptive path planning.

RViz and Gazebo support testing and validation. RViz provides real-time visualization of mapping and navigation, while Gazebo enables safe, physics-based simulations to evaluate system performance before field deployment.

By integrating Cartographer SLAM, AMCL, and A* path planning within ROS2, the system achieves reliable autonomous navigation tailored for backyard gardens, with iterative testing ensuring real-world readiness.

3.1.3 Mapping and Localization

Accurate mapping and localization are crucial for autonomous navigation in confined environments like backyard gardens. This system integrates Simultaneous Localization and Mapping (SLAM) with Adaptive Monte Carlo Localization (AMCL) to build an environment map while tracking the robot's position.

SLAM constructs an occupancy grid and continuously updates the robot's pose using Bayesian filtering:

$$p(x_t, m|z_{1:t}, u_{1:t}) = \eta\, p(z_t|x_t, m) \int p(x_t|x_{t-1}, u_t)\, p(x_{t-1}, m|z_{1:t-1}, u_{1:t-1})\, dx_{t-1} \tag{3.2}$$

This equation balances motion predictions with sensor data to refine both the map and localization. AMCL further enhances accuracy by using a particle filter to estimate the robot's pose (x, y, θ), continuously adjusting based on incoming LiDAR data (see Equation (3.1)).

The combined SLAM-AMCL framework enables real-time navigation, dynamic obstacle avoidance, and continuous map updates, ensuring reliable performance in small, unstructured outdoor spaces.

Simultaneous Localization and Mapping (SLAM)

Simultaneous Localization and Mapping (SLAM) enables robots to build a map of their environment while tracking their position within it—essential for autonomous navigation in GPS-denied environments like backyard gardens. By integrating mapping and localization, SLAM allows the robot to explore unknown spaces, detect obstacles, and dynamically update its path.

This project uses the Rplidar A3 sensor to collect real-time distance measurements, which are processed to generate a 2D occupancy grid map. As the robot moves, SLAM continuously updates both its estimated position and the map, enabling navigation without prior environmental knowledge. Loop closure techniques correct sensor drift by recognizing previously visited locations, refining the map's accuracy over time.

The probabilistic nature of SLAM is represented by:

$$p(x_t, m \mid z_{1:t}, u_{1:t}) \tag{3.3}$$

where x_t is the robot's pose, m is the map, $z_{1:t}$ are sensor measurements, and $u_{1:t}$ are control inputs. The SLAM process involves two key steps:

- **Prediction:** Estimates the new robot position based on prior motion commands:

  $$x_t = f(x_{t-1}, u_t) + w_t \tag{3.4}$$

  where w_t accounts for motion uncertainties.

- **Correction:** Refines the pose by comparing actual LiDAR data with expected measurements:

  $$p(z_t \mid x_t, m) \tag{3.5}$$

This project implements GMapping, a particle filter-based SLAM algorithm optimized for 2D LiDAR data. GMapping was chosen for its efficiency in generating accurate occupancy grids and its adaptability to confined spaces like backyard gardens.

As the robot navigates, it updates the map in real time, identifying and recording environmental features such as trees, fences, and pathways. When the robot revisits its known landmarks, it applies loop closure to correct positional drift, improving both map accuracy and localization.

While SLAM focuses on mapping and initial localization, precise real-time track-

ing requires Adaptive Monte Carlo Localization (AMCL), which refines the robot's pose using probabilistic filtering techniques, ensuring consistent and accurate navigation.

Adaptive Monte Carlo Localization (AMCL)

Adaptive Monte Carlo Localization (AMCL) is a probabilistic algorithm that refines a robot's position within a known map, ensuring accurate real-time localization essential for autonomous navigation. While SLAM builds the initial map, AMCL continuously tracks the robot's location during navigation, adapting to environmental changes and sensor noise.

AMCL uses a particle filter, where multiple particles represent possible robot positions. As the robot moves and collects LiDAR data, AMCL compares these observations with the map, assigning higher weights to particles that best match the real environment. Low-weight particles are discarded, and high-weight ones are resampled, focusing the estimate on the most probable location.

The robot's pose probability at time t is given by:

$$p(x_t \mid z_{1:t}, u_{1:t}) \tag{3.6}$$

where x_t is the robot's pose, $z_{1:t}$ are LiDAR observations, and $u_{1:t}$ are control inputs. The subscript t denotes the time step, meaning that these variables represent values at different moments in time. The notation $1 : t$ indicates a sequence of values from the initial time step 1 to the current time step t. Each particle's position updates as:

$$x_t^i = f(x_{t-1}^i, u_t) + w_t \tag{3.7}$$

where $f(x_{t-1}^i, u_t)$ models motion, and w_t accounts for noise. The superscript i refers to the i-th particle in the particle filter, meaning that the localization algorithm maintains multiple hypotheses (particles) for the robot's pose.

AMCL dynamically adjusts the number of particles based on environmental complexity—using fewer particles in open areas and more in cluttered spaces—optimizing both accuracy and computational efficiency. This flexibility ensures reliable localization, even with dynamic obstacles or sensor noise.

In this project, AMCL is implemented within the ROS2 framework. The Rplidar A3 provides real-time distance measurements, which AMCL uses to refine the robot's location during navigation. For example, as the robot explores a garden, it detects landmarks (e.g., trees or fences) and compares them to the SLAM-generated map. Over time, AMCL filters out inaccurate position estimates, converging on the robot's true location.

This continuous process allows the robot to adapt to environmental changes and maintain precise navigation without human intervention, even in complex outdoor settings like backyard gardens.

3.2 Path Planning Using the A* Algorithm

Path planning is a vital component of autonomous navigation, enabling the robot to move from its starting position to a goal while avoiding obstacles. This project utilizes the A* algorithm for its balance between computational efficiency and path optimality, making it ideal for dynamic environments like backyard gardens. By incorporating heuristics, A* reduces unnecessary computations compared to exhaustive methods like Dijkstra's algorithm, ensuring adaptive real-time navigation.

3.2.1 Grid-Based Path Planning with A*

The garden is represented as a discretized occupancy grid, where each cell is classified as free (0–49) or occupied (50–100) based on SLAM-generated data. These values are derived from the probability of occupancy assigned to each grid cell by the SLAM algorithm. Cells with values between 0 and 49 are considered

free, meaning they have a lower probability of containing an obstacle, while cells with values from 50 to 100 are marked as occupied, indicating a higher probability of obstruction. This classification enables efficient decision-making for navigation. A* evaluates paths through this grid, ensuring efficient navigation while avoiding obstacles. If a new obstacle appears, A* dynamically updates the grid and recalculates the path in real time.

A* uses a heuristic function to guide the search, with this project employing the Euclidean distance heuristic:

$$h(n) = \sqrt{(x_{\text{goal}} - x_n)^2 + (y_{\text{goal}} - y_n)^2} \tag{3.8}$$

This ensures smooth, direct trajectories while minimizing detours and maintaining optimal navigation.

3.2.2 Handling Dynamic Obstacles in Path Planning

Real-world navigation requires the robot to adjust to changing environments. The Rplidar A3 continuously scans the surroundings, updating the occupancy grid in real-time. When a new obstacle—like a person or a pet—is detected, A* triggers a local re-planning process. Instead of recalculating the entire route, only the affected area is updated, ensuring minimal computational overhead. Once the obstacle is cleared, the robot can revert to the original path.

Computational Efficiency of A* Path Planning

A*'s efficiency is influenced by grid resolution and heuristic selection. Finer grids increase precision but add computational load, while coarser grids reduce complexity at the cost of detail. This project adopts a 5 cm grid resolution within a 10m $\times$ 10m area, resulting in a 40,000-node search space—balancing performance and accuracy. The 5 cm resolution was chosen based on the LiDAR sensor's precision and the need for accurate navigation in confined garden spaces while keeping computational requirements manageable on embedded hardware.

The Euclidean heuristic and early exit conditions further reduce computation time, enabling real-time operation on the NVIDIA Jetson Nano. Early exit conditions include terminating the search once the goal node is reached and discarding paths with excessive cost, ensuring efficient path planning without unnecessary computation.

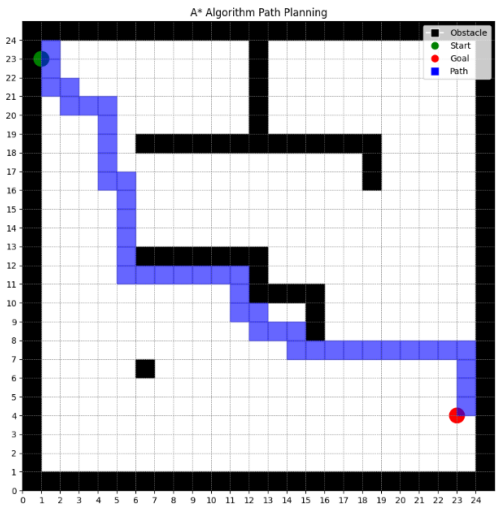


Figure 3.2: A* Pathfinding Visualization

3.2.3 Dynamic Replanning in A*

Unlike static path planning, this system integrates incremental A* updates to handle moving obstacles. When the robot detects a new obstacle, only the necessary portion of the path is recalculated. This dynamic approach, inspired by D* Lite, minimizes computational overhead. Although D* Lite is specifically designed for real-time replanning, A* with incremental updates achieves similar performance while maintaining simplicity. Since the core algorithm remains A* with modifications rather than a full implementation of D* Lite, referring to it as "incremental A*" better reflects the approach used in this system. For example, if a fallen branch blocks the planned path, A* updates only the affected grid cells, allowing the robot to navigate around the obstruction without triggering a full replan. This ensures both responsiveness and efficiency.

Key Takeaways

A* combines heuristic-driven efficiency with dynamic replanning, enabling the robot to navigate confined and changing environments in real time. The optimized 5 cm grid resolution further supports precise and efficient navigation, making A* a practical solution for backyard garden applications.

3.3 System Integration

The autonomous robotic system for backyard navigation is designed as a fully integrated framework, where mapping, localization, path planning, and motion control function as interconnected components. This section details the workflow of the system, describing how data flows between the Cartographer SLAM module, Adaptive Monte Carlo Localization (AMCL), and the A* path-planning algorithm to enable smooth, autonomous navigation.

3.3.1 Workflow of Autonomous Navigation

The system operates in two main phases: Mapping Mode and Navigation Mode, each involving different components that work together to achieve full autonomy.

Phase 1: Mapping Mode

In Mapping Mode, the robot autonomously explores its environment to generate an occupancy grid map, which serves as the foundation for future navigation. This process consists of three key stages: LiDAR data collection, SLAM processing, and map storage.

The LiDAR data collection stage begins with the Rplidar A3 sensor continuously scanning the surroundings at high resolution, capturing precise distance measurements of obstacles and free spaces. These raw LiDAR scans are then transmitted to the Cartographer SLAM module, which processes the data in real time.

The Cartographer SLAM implementation in this system subscribes to the `/scan` topic, receiving real-time LiDAR scans from the Rplidar A3 sensor. The output is published to the `/map` topic, providing a dynamically updated occupancy grid. The resolution of the occupancy grid is set to 0.05m per cell, ensuring fine-grained mapping accuracy while balancing computational efficiency. Additionally, Cartographer's loop closure detection minimizes mapping errors by recognizing previously visited locations and adjusting the map to correct for drift. This ensures that even if the robot revisits an area from a different angle, the map remains consistent. Loop closure detection is employed to identify previously visited locations, correcting positional drift and refining the generated map for long-term accuracy.

Once the mapping process is complete, the final occupancy grid map is stored and later used by the Adaptive Monte Carlo Localization (AMCL) algorithm, which enables real-time localization during the Navigation Mode.

Phase 2: Navigation Mode

In Navigation Mode, the robot autonomously moves toward designated target locations while avoiding obstacles. This phase relies on the previously generated occupancy grid map and consists of three main steps: map loading and localization, goal assignment and path planning, and real-time motion execution.

The map loading and localization step begins by loading the stored occupancy grid map into the AMCL module, which employs particle filter-based estimation to determine the robot's precise location. AMCL continuously updates the estimated position by comparing real-time LiDAR scans with the pre-built map.

In the goal assignment and path planning stage, a target location is assigned within the mapped environment. The A* path-planning algorithm then computes the shortest and collision-free route from the robot's current position to the goal, ensuring an optimal path for navigation.

The system features a key navigation component: the Scouter Server. The Scouter

Server executes autonomous movement by sending velocity commands to the `/cmd_vel` topic, ensuring the robot follows the planned trajectory.

For path-planning details, refer to Section 2.6.3, which explains how the A* algorithm updates occupancy grids and dynamically recalculates paths when obstacles are detected.

The integration of SLAM, AMCL, and A* ensures a seamless transition from mapping to autonomous navigation. Initially, SLAM generates a 2D occupancy grid that represents the environment. This map is then utilized by AMCL to estimate the robot's real-time position, allowing continuous localization updates as the robot moves. Once a navigation goal is set, A* computes the optimal path within the mapped space. As new obstacles are detected via LiDAR, the occupancy grid is updated, and A* recalculates an alternate route if needed. This dynamic interplay allows the robot to autonomously explore, localize, and navigate with minimal human intervention.

3.3.2 System Communication Flow

The autonomous robotic system employs a ROS2-based distributed architecture, where specialized nodes handle mapping, localization, path planning, and motion control. These nodes communicate through a streamlined data flow, ensuring real-time integration of environmental perception, navigation, and decision-making.

As shown in Figure 3.3, the process begins with real-time LiDAR scans from the Rplidar A3, which the Cartographer SLAM module processes to generate a 2D occupancy grid. This grid supports both mapping and localization, with AMCL estimating the robot's real-time position. The Manager Node oversees this data flow, coordinating transitions between Mapping Mode and Navigation Mode.

During mapping, the Watchtower Node tracks progress, directing the Scouter Server to explore uncharted regions until full coverage is achieved. The A* package manages motion control, guiding the robot along safe paths while dynamically updating the map.

The Manager Node handles system-wide coordination, including error recovery and state management, ensuring the robot can adapt to dynamic environments and continue operation with minimal user intervention.

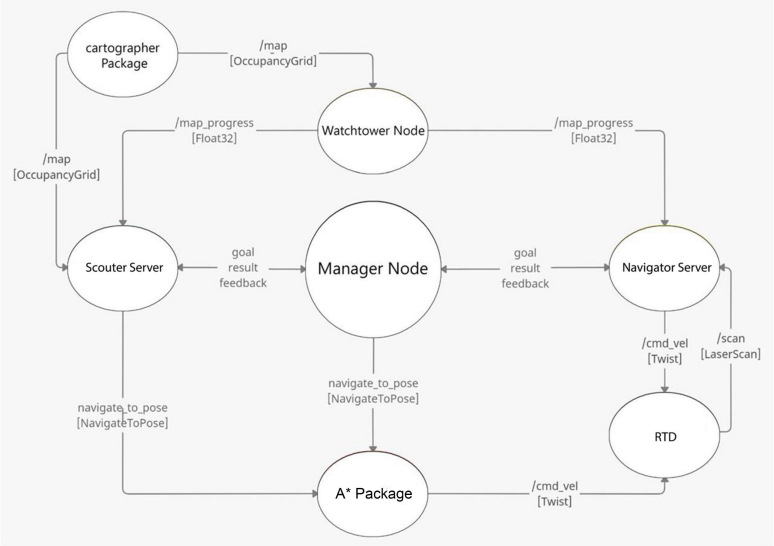


Figure 3.3: Overview of System Communication Flow

This streamlined communication flow enables seamless integration of mapping, localization, and navigation, supporting fully autonomous operation even in dynamic and confined environments.

3.4 Conclusion

This chapter outlined the system's architecture, covering hardware, software, and integration. The Rplidar A3 and Jetson Nano enable efficient mapping and processing, while ROS 2 ensures modular communication.

The system combines SLAM (Cartographer), localization (AMCL), and navigation (A*) for autonomous mapping, path planning, and obstacle avoidance. The Manager Node coordinates these processes, with Gazebo and RViz facilitating pre-deployment testing.

Offering a scalable, cost-effective solution for backyard navigation, the system operates solely on 2D LiDAR without GPS or predefined maps. The next chapter details implementation, testing, and performance evaluation.

Chapter 4

Implementation

4.1 Introduction

This chapter details the implementation of the autonomous navigation system, transitioning from design to real-world execution. It covers software integration, sensor-data processing, and algorithm deployment, ensuring seamless interaction between mapping, localization, and path planning modules.

The Rplidar A3 sensor, paired with Cartographer SLAM and AMCL, enables real-time mapping and localization, while the A* algorithm computes optimal paths for efficient navigation. ROS2 facilitates communication between system components, with RViz and Gazebo aiding in testing and visualization.

The following sections outline the robot's software architecture, code structure, and implementation challenges, providing insights into its real-world operation.

4.2 Initialization

The initialization phase sets up the autonomous robotic system by activating essential components for mapping, navigation, discovering, and monitoring. It ensures that both software and hardware elements are properly configured for autonomous operation in both simulated environments and real-world deployment. Built around the ROS2 launch framework, the process efficiently coordinates core

components such as Gazebo for simulation, Cartographer for SLAM, and A* for navigation. Custom module Scouter Servers further support intelligent discovering strategies, adapting the system to various operational scenarios.
This phase ensures seamless synchronization across all components, enabling reliable communication and modular adaptability. By establishing a stable foundation, the system is prepared for precise and efficient autonomous operations in diverse environments.

4.2.1 Process

The initialization process begins by executing the `seeker.launch.py` file, which orchestrates the startup of all essential nodes for SLAM, navigation, discovering, and progress monitoring.
This primary launch file integrates sub-launch files for mapping, discovering, and simulation. The `OpaqueFunction` dynamically configures system components based on user inputs or default settings, ensuring seamless integration with Gazebo for simulation and A* for path planning, navigation, and obstacle avoidance.
The detailed implementation of the launch file is available in **Appendix F**, specifically in **Listing F.1**, lines 8–32. It handles core configurations, including simulation parameters and synchronization of simulation time via the `use_sim_time` parameter.

Cartographer Package

The Cartographer package handles SLAM (Simultaneous Localization and Mapping) using real-time LiDAR data from the Rplidar A3 to generate a 2D occupancy grid. The `cartographer.launch.py` file, integrated into the main launch process, configures key parameters like resolution and publish frequency.
The `cartographer_node` subscribes to LiDAR data (`/scan`) and publishes the occupancy grid map (`/map`), which is visualized in RViz. Detailed configuration is provided in **Appendix D**, **Listing D.1**, lines 8–27.

A* Package

The A* package manages localization, path planning, and motion control, using costmap resolution, robot dimensions, and dynamic environment parameters. It integrates with the Cartographer map for seamless navigation.
The A* stack processes navigation goals through the `/navigate_to_pose` action interface. Full configuration details are in **Appendix F**, **Listing F.1**, lines 33–45.

Scouter Server

The Scouter Server implements a structured exploration strategy by planning paths to unexplored areas and sending navigation goals to the A* stack via the `/navigate_to_pose` interface, ensuring full map coverage.
The detailed implementation is provided in **Appendix A**, **Listing A.1**, which outlines the ‘ScoutingServer‘ class and its dynamic goal generation until the map is fully explored.

Watchtower Node

The Watchtower Node tracks map exploration progress by subscribing to the `/map` topic and publishing updates on `/map_progress`. It calculates the percentage of the map explored, providing real-time feedback to the Manager Node and RViz.

Listing 4.7 shows the implementation of the ‘Subscriber‘ class used for monitoring:

Listing 4.1: Watchtower Map Exploration Logic

```
# Code Snippet (from watchtower.py)
class Subscriber(Node):
    def listener_callback(self, msg):
        map_array = numpy.asarray(msg.data)
        map_explored = numpy.count_nonzero(
            (map_array <= self.free_thresh) & (map_array >
               -1)
        ) * resolution2
        percentage_explored = map_explored / self.free_space
        map_explored_msg = Float32()
        map_explored_msg.data = min(percentage_explored,
           1.0)
        self.publisher_.publish(map_explored_msg)
```

Figure 4.1 illustrates the full initialization workflow, detailing how components like the Scouter Server and Watchtower Node coordinate to manage system startup, exploration, and progress monitoring.

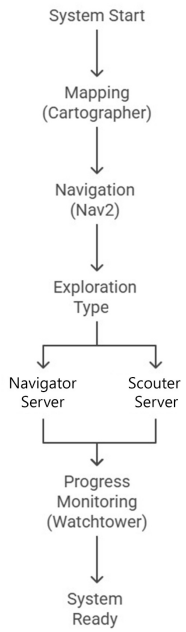


Figure 4.1: System initialization workflow.

This workflow highlights the end-to-end initialization process, from system startup to mapping, navigation, and exploration, ensuring the robot is ready for autonomous operation.

4.2.2 Explanation of the Initialization Phase

The initialization phase configures the system using the ROS 2 launch file `seeker.launch.py`, which orchestrates the setup of essential components for mapping, navigation, discovering, and progress monitoring. This file dynamically manages nodes and parameters, ensuring seamless integration across different operational scenarios without manual intervention. During initialization, the Cartographer package is launched for SLAM, the A* stack is configured for localization and path planning, and Gazebo is set up for simulation testing. Additionally, a custom exploration

server, the Scouter Server, is integrated to support autonomous operation. This consolidated approach ensures a streamlined setup process, enhancing system reliability and adaptability. Detailed implementation of the launch configuration is provided in **Appendix F**, specifically in **Listing F.1**, lines 3–29.

Simulation in Gazebo

Gazebo simulation provides a controlled virtual environment to test mapping, obstacle avoidance, and navigation strategies before real-world deployment. The simulation uses world models created from CSV files, such as `map7.csv`, which define the layout of the environment. These files are converted into SDF format (e.g., `map7.sdf`) to ensure compatibility with Gazebo. Robot behaviors are tested within this virtual setup, allowing for the evaluation of critical functionalities, with progress visualized in both RViz and Gazebo in real time. The integration of Gazebo within the initialization process is detailed in **Appendix F**, specifically in **Listing F.1**, lines 11–18.

The following XML snippet illustrates a sample model configuration used to load simulation environments into Gazebo:

Listing 4.2: XML Model Configuration Example

```
1 <model>
2   <name>map7</name>
3   <version>1.0</version>
4   <sdf version="1.5">map7.sdf</sdf>
5   <description>Map generated automatically from a CSV file
        .</description>
6 </model>
```

This simulation environment enables developers to validate system behavior, refine navigation strategies, and debug potential issues before deploying the system in real-world conditions.

Real Robot Initialization

When transitioning from simulation to real-world deployment, specific modifications are necessary within the `seeker.launch.py` file. Simulation-specific components, such as `gzserver_cmd` and `gzclient_cmd`, must be disabled. Additionally, the `use_sim_time` parameter should be set to `false` to enable real-time operation and accurate processing of live sensor data.

Listing 4.3: Real Robot Configuration Example

```
use_sim_time = LaunchConfiguration('use_sim_time', default='
    false')
# Disable simulation time

# Comment out Gazebo components
# gzserver_cmd = IncludeLaunchDescription(...)
# gzclient_cmd = IncludeLaunchDescription(...)
```

These adjustments ensure the system correctly interprets real-time sensor inputs and executes navigation commands in the physical environment.

Hardware Integration

The real robot setup involves integrating the Rplidar A3 sensor and the Jetson Nano as the primary processing unit. The Rplidar A3 provides real-time LiDAR data for mapping and obstacle detection. It is initialized using the `hls_lfcd_lds_driver` package, which allows communication between the sensor and the ROS 2 framework. The following code snippet demonstrates the initialization process for the Rplidar node:

Listing 4.4: Rplidar Node Initialization

```
lidar_cmd = Node(
    package='hls_lfcd_lds_driver',
    executable='hlds_laser_publisher',
    name='hlds_laser_publisher',
    output='screen',
    parameters=[{'port': '/dev/ttyUSB0', 'frame_id': 'laser'
       }]
)
```

The Jetson Nano serves as the main computational unit, running ROS 2 nodes such as Cartographer for SLAM and A* for navigation. It handles real-time data processing, path planning, and system control. Users can access the Jetson Nano remotely via SSH to deploy the system and manage operations. The following commands illustrate the SSH connection and the process for launching the ROS 2 system:

Listing 4.5: Jetson Nano SSH and ROS Launch Commands

```
ssh user@jetson-nano
ros2 launch seeker_bringup seeker.launch.py
```

This hardware integration ensures that the robot can process live sensor data, manage exploration strategies, and execute navigation tasks autonomously. Proper configuration of both the Rplidar and Jetson Nano is essential for maintaining reliable performance in real-world environments.

By maintaining a modular initialization process, the system supports seamless transitions between simulation and real deployment. This flexibility allows developers to conduct extensive testing in virtual environments, refine system performance, and minimize risks before field deployment. The structured approach to hardware integration further ensures that the system remains adaptable, scalable, and resilient across various operational scenarios.

4.3 Mapping Phase

Mapping is a critical process that enables autonomous robots to perceive and interpret their surroundings effectively. It involves generating a detailed representation of the environment, allowing the robot to navigate safely, plan efficient paths, and execute discovering tasks. In this system, the mapping phase centers around integrating the Rplidar A3 sensor for real-time scanning with the Cartographer Package to produce a 2D occupancy grid map. This map forms the foundation for localization, navigation, and discovering, enabling the robot to interact intelligently with its environment.

Figure 4.2 illustrates the workflow that incorporates the Mapping Phase, marking the starting point of the system's autonomous capabilities and laying the groundwork for subsequent discovering and navigation tasks.

Figure 4.2: Workflow incorporating the Mapping Phase.

At the core of this process is the Simultaneous Localization and Mapping (SLAM) algorithm, which not only constructs the map but also determines the robot's position within it. By processing real-time LiDAR data, the system dynamically classifies regions into free, occupied, and unknown spaces, forming a structured

grid that guides the robot's decision-making. Metadata such as resolution and map origin ensures the precise alignment of the grid within the global coordinate system, enhancing the robot's ability to navigate and adapt to its surroundings. The Cartographer Package processes raw laser data to create the occupancy grid, while visualization tools like RViz enable real-time monitoring of mapping progress. Complementary nodes, such as the Watchtower Node, further support the discovering process by analyzing unexplored regions and tracking mapping completion. This phase establishes the robot's environmental awareness, enabling it to navigate complex surroundings in both simulated and real-world environments.

4.3.1 Real-Time Map Generation with Cartographer

The Cartographer Package uses raw laser data published to the `/scan` topic by the Rplidar A3 sensor to create a 2D occupancy grid map. This data includes distance measurements that represent obstacles and free space within the robot's field of view. By applying SLAM algorithms, Cartographer simultaneously constructs the occupancy grid and determines the robot's position within it. The `cartographer_node` subscribes to the `/scan` topic and publishes the generated map to the `/map` topic.

The occupancy grid categorizes the environment into three types of spaces: **free**, **occupied**, and **unknown**. Free spaces (values 0–49) indicate navigable areas, while occupied spaces (values 50–100) represent obstacles. Unknown spaces, where no LiDAR data exists, are marked as -1. For example, in a 10x10 grid, a cell value of 30 denotes a free space, 80 represents an obstacle, and -1 indicates an unexplored region. This classification enables the robot to differentiate between navigable areas, hazards, and unmapped regions.

Metadata, including map resolution (e.g., 0.05m per grid cell) and the map's origin, ensures accurate positioning within the global coordinate system. The occupancy grid is continuously updated in real time and published in `OccupancyGrid` format on the `/map` topic.

The configuration of the `cartographer_node` within the ROS 2 launch file is shown below:

Listing 4.6: Cartographer Node Configuration

```
Node(
    package='cartographer_ros',
    executable='cartographer_node',
    name='cartographer_node',
    output='screen',
    parameters=[{'use_sim_time': use_sim_time}],
    arguments=[
        '-configuration_directory', cartographer_config_dir,
        '-configuration_basename', configuration_basename
    ]
)
```

The `cartographer_node` handles real-time LiDAR data, SLAM processing, and 2D map generation. The parameter `use_sim_time` ensures synchronization with Gazebo during simulations, while the SLAM configuration file (`turtlebot3_lds_2d.lua`) defines essential parameters such as grid resolution and sensor update rates. This flexibility allows the system to adapt to varying operational scenarios while maintaining mapping accuracy.

4.3.2 Map Output and Analysis

The final output of the Cartographer process is a continuously updated 2D occupancy grid map published to the `/map` topic in `OccupancyGrid` format. This map provides spatial data regarding obstacles, free spaces, and unmapped regions and includes metadata like resolution and origin for accurate positioning. The map serves as a critical resource for navigation, path planning, and monitoring system performance.

The Watchtower Node plays a vital role in analyzing the map's progress by sub-

scribing to the `/map` topic and calculating the percentage of the environment that has been mapped. This node processes the occupancy grid to determine coverage and updates the `/map_progress` topic, enabling real-time tracking of discovering completion. The following code snippet highlights the Watchtower Node's logic for processing map data and calculating explored areas:

Listing 4.7: Watchtower Map Discovering Logic

```
# Code Snippet (from watchtower.py)
class Subscriber(Node):
    def listener_callback(self, msg):
        map_array = numpy.asarray(msg.data)
        map_explored = numpy.count_nonzero(
            (map_array <= self.free_thresh) & (map_array >
               -1)
        ) * resolution2
        percentage_explored = map_explored / self.free_space
        map_explored_msg = Float32()
        map_explored_msg.data = min(percentage_explored,
           1.0)
        self.publisher_.publish(map_explored_msg)
```

The detailed implementation of the Watchtower Node is available in **Appendix C**, specifically in **Listing C.1**, lines 3–43. This node ensures continuous progress tracking during the mapping phase, enabling the robot to identify unexplored regions and complete the mapping process efficiently.

The Watchtower Node supports the discovering process by providing real-time feedback on mapping progress, ensuring comprehensive environmental coverage. This continuous analysis is essential for evaluating system performance and verifying that the mapping process covers the entire environment before transitioning to the navigation phase.

4.3.3 Mapping Workflow and Applications

The mapping workflow begins with the Rplidar A3 sensor, which continuously scans the environment and publishes laser scan data to the `/scan` topic. The `cartographer_node` processes this data using SLAM algorithms to construct a detailed 2D occupancy grid map. This SLAM process not only builds the map but also localizes the robot within the environment, ensuring accurate navigation from the very beginning.

The occupancy grid serves as the backbone for the robot's discovering and navigation tasks. It allows the system to identify navigable paths, detect and avoid obstacles, and dynamically plan efficient routes. By merging real-time LiDAR data with the generated map, the robot achieves robust localization and precise movement across its environment. This capability is vital for adapting to new obstacles and dynamic changes in both simulated and real-world scenarios.

RViz provides real-time visualization of the mapping process, enabling operators to monitor the robot's progress and validate SLAM performance. As the robot navigates, the map is continuously updated, reflecting changes in the environment and ensuring the robot always has an accurate, up-to-date spatial representation. This live mapping feature enhances the system's adaptability, supporting safe navigation and efficient task execution.

Initial Mapping Stage

In the initial mapping stage, shown in Figure 4.3, the robot begins scanning its environment using the Rplidar A3 sensor. At this early phase, the occupancy grid is sparsely populated, with large sections still marked as unknown. The laser scan data, streamed through the `/scan` topic, forms the foundational layer for map construction, while the Cartographer Package processes this data in real time. This stage highlights the robot's exploratory nature as it begins localizing itself and establishing spatial awareness within the environment.

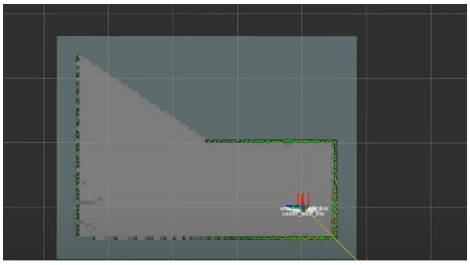


Figure 4.3: Initial mapping stage.

Mid-Mapping Stage

Figure 4.4 captures the progress made during the middle of the mapping process. By this stage, the SLAM algorithm has significantly populated the occupancy grid, classifying large portions of the environment into free, occupied, and unknown regions. Green outlines in the figure indicate detected obstacles, while gray areas represent navigable spaces. The robot continuously refines the map, leveraging metadata such as resolution and global origin to maintain alignment within the global coordinate system. The updated map is consistently published to the `/map` topic, enabling the robot to plan optimized paths and adapt to environmental changes.

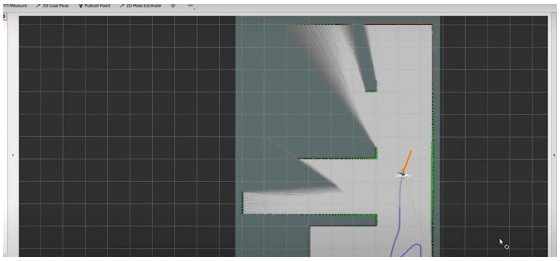


Figure 4.4: Mid-mapping stage.

4.3.4 Final Mapping Stage

In the final mapping stage, illustrated in Figure 4.5, the occupancy grid reaches completion. The robot has fully explored the environment, identifying all navigable areas, obstacles, and remaining unknown spaces. The dense grid of green lines and gray regions indicates that the SLAM process has successfully created a

comprehensive and accurate map. The robot's movement trajectory, highlighted in blue and orange, showcases its path during the discovering phase. The system determines the completion of mapping by monitoring the `watchtower` Node, which continuously evaluates the proportion of explored versus unexplored regions. The exploration process concludes when the percentage of unknown cells falls below a predefined threshold, ensuring that all accessible areas have been mapped. This is tracked using the map_progress metric, which provides real-time updates on exploration status. This complete map now serves as the foundation for advanced navigation tasks, enabling the robot to follow precise routes and efficiently avoid obstacles. Real-time updates continue to refine the map during operation, ensuring it remains an accurate reflection of the environment.

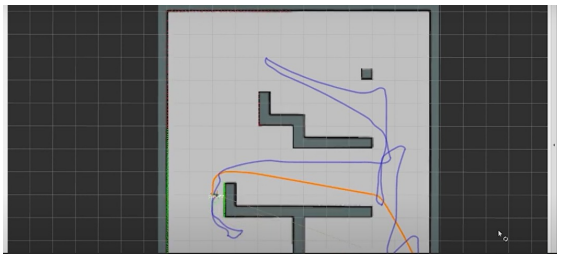


Figure 4.5: Final mapping stage.

4.4 Discovering Phase

The discovering phase is a crucial stage in the robot's autonomous operation, where it actively scans and maps its environment to ensure full coverage. This phase builds upon the initial mapping, focusing on expanding the robot's understanding of its surroundings.

The Watchtower Node plays an integral role throughout the discovering phase by monitoring mapping progress in real time. It analyzes data from the evolving occupancy grid and continuously publishes updates on coverage status. These updates allow the system to adapt dynamically, identifying regions that require further exploration and guiding the robot accordingly. This is achieved by computing the proportion of unexplored cells in the occupancy grid and publishing

the map_progress metric, which represents the percentage of the environment that has been mapped. The system continuously compares this metric against a predefined threshold, adjusting exploration efforts accordingly. If large unexplored areas remain, the robot prioritizes their coverage, while already mapped regions are de prioritized to avoid redundancy. The Watchtower's real-time feedback ensures that discovering efforts remain focused and efficient, preventing redundant mapping and optimizing overall performance.

This phase represents a transition from foundational mapping to active environmental engagement, where the robot not only constructs the map but also refines it through strategic exploration. The Manager Node coordinates this process, determining whether the Scouter Server will manage each discovering task based on current needs. The Manager Node evaluates real-time occupancy grid data and map_progress values, ensuring that the Scouter Server executes a systematic search pattern to maximize coverage.

By combining adaptive strategies with continuous progress monitoring, the discovering phase ensures efficient and comprehensive coverage of the environment. This stage not only strengthens the accuracy of the generated map but also enhances the robot's autonomy, laying the groundwork for reliable navigation and localization in future tasks.

4.4.1 Scouting (Structured Discovering)

The Scouter Server enables structured discovering by systematically guiding the robot to unexplored areas within its environment. This process starts with data from the Cartographer SLAM module, which generates a real-time occupancy grid map published on the `/map` topic. The occupancy grid categorizes each cell as free, occupied, or unknown, providing a detailed layout that supports precise exploration.

In the 2D occupancy grid, free spaces (values 0–49) represent navigable regions, occupied spaces (50–100) denote obstacles, and unknown spaces (-1) indicate areas

yet to be explored. This classification enables the Scouter Server to identify frontiers—boundaries between known and unknown regions—which serve as potential targets for further exploration.

The frontier detection algorithm focuses on identifying these frontiers by analyzing the occupancy grid. It converts the grid into a 2D NumPy array for efficient computation, scanning for cells marked as -1 (unknown) that are adjacent to free cells. These adjacent unknown cells form the frontiers, guiding the robot toward unexplored regions.

Listing 4.8 demonstrates the `detect_frontiers` function, which identifies the boundaries between explored and unexplored areas in the occupancy grid.

Listing 4.8: Frontier Detection Algorithm (Simplified)

```
import numpy as np
def detect_frontiers(occupancy_grid, resolution):
    # Convert occupancy grid to 2D NumPy array
    grid = np.array(occupancy_grid.data).reshape(
        occupancy_grid.info.height, occupancy_grid.info.
           width )
    frontiers = []
      # Identify unknown cells (-1) adjacent to free space
          (<50)
    for x in range(1, grid.shape[0] - 1):
        for y in range(1, grid.shape[1] - 1):
            if grid[x, y] == -1 and any(
                neighbor < 50 for neighbor in grid[x-1:x+2,
                   y-1:y+2].flatten()):
                frontier_x = x * resolution + occupancy_grid
                   .info.origin.position.x
                frontier_y = y * resolution + occupancy_grid
                   .info.origin.position.y
                frontiers.append((frontier_x, frontier_y))
```

Once frontiers are detected, the Scouter Server selects the next exploration target based on heuristics like proximity, cluster size, and coverage efficiency. Proximity prioritizes frontiers closest to the robot to minimize travel time. Cluster size considers groups of adjacent frontiers, preferring larger clusters for more efficient exploration. Coverage efficiency ensures the robot avoids revisiting already explored areas by referencing its trajectory.

The Euclidean distance used to calculate proximity between the robot and each frontier is defined by:

$$d = \sqrt{(x_{\text{robot}} - x_{\text{frontier}})^2 + (y_{\text{robot}} - y_{\text{frontier}})^2} \tag{4.1}$$

To optimize exploration, the Scouter Server clusters frontiers using the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm. DBSCAN groups nearby frontiers into clusters based on two parameters: Epsilon (ϵ), defining the maximum distance between points within a cluster (e.g., 0.5 m), and MinPts, the minimum number of points to form a cluster (e.g., 5). This clustering process filters out isolated points and focuses the robot on dense areas of unexplored space.

Listing 4.9 presents the `cluster_frontiers` function, which applies DBSCAN to group frontiers.

Listing 4.9: Frontier Clustering with DBSCAN

```
from sklearn.cluster import DBSCAN
import numpy as np

def cluster_frontiers(frontiers, eps=0.5, min_samples=5):
    frontiers_array = np.array(frontiers)
    clustering = DBSCAN(eps=eps, min_samples=min_samples).
       fit(frontiers_array)
    clusters = {}

    for idx, label in enumerate(clustering.labels_):
        if label == -1:  # Noise point
            continue
        if label not in clusters:
            clusters[label] = []
        clusters[label].append(frontiers_array[idx])

    return clusters
```

An example input and output of the clustering process is shown in Listing 4.10.

Listing 4.10: Example Frontier Clustering Output

```
# Input: [(1.2, 3.5), (1.3, 3.6), (4.0, 7.0), (4.2, 7.1)]
# Output:
{
  0: [(1.2, 3.5), (1.3, 3.6)],  # Cluster 0
  1: [(4.0, 7.0), (4.2, 7.1)]   # Cluster 1
}
```

After clustering, the Scouter Server selects the most promising cluster based on size and proximity. Within the chosen cluster, the specific frontier point closest to the robot is selected as the navigation target. The final navigation goal is sent to the A* stack via the `/navigate_to_pose` action, ensuring precise and efficient movement toward unexplored regions.

The detailed implementation of this process is provided in **Appendix A**, specifically in **Listing A.1**, lines 41–73, where the `send_goal` method handles communication with the A* action server.

The Scouter Server's decisions and the robot's planned path are visualized in RViz, providing real-time feedback to operators and enabling easier debugging.

Structured discovering offers significant advantages over random exploration. It minimizes redundant movements, optimizes resource usage like battery life and computational power, and dynamically adapts to mapping progress by prioritizing unexplored areas. The scalability of the approach—thanks to the use of clustering algorithms—ensures the system remains efficient even in large, complex environments.

This method significantly enhances the robot's capability to map unknown environments in a structured, efficient manner, ultimately improving both exploration speed and map quality.

Map Discovering Progress

The **Watchtower Node** monitors the robot's discovering progress by subscribing to the `/map` topic, where occupancy grid data from the Cartographer Node is continuously updated. It analyzes this data to calculate the percentage of the environment explored, using the ROS 2 `OccupancyGrid` message, which provides a 2D grid representation of the environment. Each cell in the grid carries values indicating its state: **0–49** for free space, **50–100** for occupied space, and **-1** for unknown regions. Metadata such as grid resolution (e.g., 0.05 meters per cell) and map origin ensures accurate alignment within the global coordinate system.
The explored area percentage (S_e) is calculated using:

$$S_e = \left(\frac{N_k}{S_f} \right) \times 100 \tag{4.2}$$

where:

- S_e is the explored area percentage (%).
- N_k is the count of known cells (free + occupied).
- S_f is the total free space in the map.

The `OccupancyGrid` data array is reshaped into a 2D NumPy array for efficient processing:

```
map_array = numpy.asarray(msg.data).
reshape((msg.info.height, msg.info.width))
```

Cells identified as free or occupied are filtered using:

```
(map_array <= self.free_thresh) & (map_array > -1)
```

The explored area is then calculated by multiplying the number of known cells by the area of each grid cell:

$$S_e = N_k \times r^2 \tag{4.3}$$

For example, in a grid with $N_k = 6000$ known cells and a resolution $r = 0.05$ meters:

$$S_e = 6000 \times (0.05)^2 = 15\,\mathrm{m}^2 \tag{4.4}$$

The explored percentage (P_e) is then determined by:

$$P_e = \left(\frac{S_e}{S_f}\right) \times 100 \tag{4.5}$$

To prevent misinterpretation, cells with a value of -1 are excluded from calculations. As the map updates, the progress is recalculated in real time and published on the `/map_progress` topic.

The progress value is encapsulated in a `Float32` message and capped at 100%:

```
map_explored_msg = Float32()
map_explored_msg.data = min(percentage_explored, 1.0)
self.publisher_.publish(map_explored_msg)
```

The detailed implementation is provided in **Appendix C**, specifically in **Listing C.1**, lines 7–59.

For example, a published message might indicate `0.75`, representing 75% of the map explored. Progress updates occur whenever new `/map` messages are received, allowing operators to monitor real-time progress in RViz. When the explored area surpasses a predefined threshold (e.g., 90%), the system transitions to the next operational phase, such as navigation or task execution.

Capping the progress value at 100% prevents computational artifacts from causing incorrect values, such as exceeding full coverage due to numerical precision errors. Without capping, floating-point inaccuracies or repeated updates could result in progress values greater than 100%, leading to misleading system behavior. By enforcing an upper limit, the system ensures reliable reporting and prevents unnecessary reconfigurations or unintended transitions.

This structured approach ensures accurate progress tracking and efficient coverage, enhancing both simulation and real-world deployments.

Visualization of Discovering Progress

The visualization of discovering progress is crucial for real-time monitoring of the robot's mapping and navigation activities. Using **RViz**, a 3D visualization tool in ROS 2, operators can observe the occupancy grid map, sensor data, and discovering progress in an intuitive format.

The `/map` topic provides the occupancy grid, dynamically updated to reflect the robot's understanding of its environment. RViz displays three key regions: **free spaces** (navigable areas), **occupied spaces** (obstacles), and **unknown spaces** (unexplored regions). This real-time feedback allows operators to track mapping status and the robot's progress.

The RViz setup is configured using files like `tb3_cartographer.rviz`, where parameters define the grid's appearance and functionality. An example snippet:

```
- Alpha: 0.699999988079071
  Class: rviz_default_plugins/Map
  Color Scheme: map
  Enabled: true
  Name: Map
  Topic: /map
```

This ensures clear visualization, enabling operators to monitor the robot's progress and identify unexplored areas. Discovering progress, computed by the **Watchtower Node**, is published on the `/map_progress` topic and visualized in RViz as either a numeric percentage or a heatmap overlay, distinguishing explored and unexplored regions.

For example, a numerical display can show the exact percentage explored, while a heatmap highlights coverage gaps. This dual representation offers both precision and intuitive understanding.

The progress calculation is handled by the `listener_callback` function, which processes the occupancy grid and computes the explored area:

Listing 4.11: Listener Callback Function

```
def listener_callback(self, msg):
    map_array = numpy.asarray(msg.data)  # Convert to NumPy
       array
    resolution = msg.info.resolution     # Extract grid
       resolution
    # Count free/occupied cells (exclude unknown)
    map_explored = numpy.count_nonzero(
        (map_array <= self.free_thresh) & (map_array > -1)
    ) * resolution2
    # Publish explored percentage
    self.publisher_.publish(Float32(data=map_explored / self
       .free_space))
```

This function ensures accurate, real-time progress updates. The explored area is recalculated with each `/map` message and capped at 100% before publishing. These updates are visualized in RViz, enabling dynamic monitoring of the mapping process.

In addition to map progress, RViz visualizes the robot's current state and planned paths. The robot's position is tracked using the `/tf` topic, allowing real-time updates as the robot navigates. Paths generated by the **Scouter Server** are displayed to show planned routes and navigation strategies. The following YAML snippet configures path visualization:

```
- Alpha: 1
  Class: rviz_default_plugins/Path
  Topic: /navigate_to_pose
  Color: 255; 0; 0
  Enabled: true
  Name: Navigation Path
```

This setup highlights the robot's planned routes, helping operators monitor navi-

gation decisions and troubleshoot if necessary.

RViz Visualization Benefits: RViz offers several advantages that enhance the efficiency and reliability of the discovering process. One of the primary benefits is **real-time monitoring**, which enables dynamic observation of the robot's mapping progress and movements, allowing operators to track changes in the environment as they occur. Additionally, RViz supports **efficient debugging** by providing clear visual cues that help identify inconsistencies in SLAM, path planning errors, and discrepancies in sensor data. This capability is essential for maintaining accurate navigation and ensuring system stability. Another critical benefit is **progress feedback**, where RViz delivers both numeric and graphical representations of the explored area. This dual-mode feedback guides discovering strategies, helping operators make informed decisions to optimize coverage and improve the overall effectiveness of the mapping process.

Visualization also serves as a feedback loop, helping operators adjust discovering strategies based on real-time coverage data. It supports identifying unexplored regions and optimizing path planning for efficient coverage.

Figure 4.6 illustrates the structured discovering strategy, showcasing how the robot detects frontiers, clusters them, selects a target, and navigates while monitoring discovering progress.

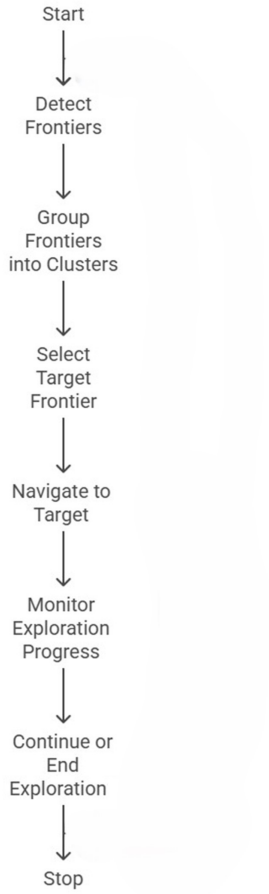


Figure 4.6: Structured Discovering strategy.

4.4.2 Explanation and Advantages of Discovering Strategy

The discovering phase employs a structured discovering strategy to ensure systematic and efficient environmental coverage. This approach is designed to optimize path planning while maintaining adaptability to complex environments.

Structured discovering leverages the Scouter Server, which performs deliberate path planning to focus on unexplored areas. This strategy uses occupancy grid data published on the `/map` topic and precise navigation goals communicated via `/navigate_to_pose`.

The path planning and navigation process systematically explores regions marked

as "unknown" (-1) in the occupancy grid. The Scouter Server identifies these unexplored cells by analyzing the grid data and appending their coordinates to a list for further processing. For example, cells with a value of -1 are detected using logic such as:

```
if grid_cell == -1:
    unexplored_region.append(cell_coordinates)
```

Once unexplored regions are identified, the server calculates navigation goals by prioritizing areas close to the robot's current position. This enhances efficient discovering with minimal travel distance. These navigation goals are published to the `/navigate_to_pose` topic, with precise coordinates specified for each goal. For instance, a target goal is set using:

```
goal_msg.pose.pose.position.x = float(target_x)
goal_msg.pose.pose.position.y = float(target_y)
```

This structured approach ensures that the robot systematically covers the environment, optimizing resource utilization and minimizing redundancy.

This method provides several advantages. It enables efficient coverage by focusing exclusively on unexplored regions, ensuring that no effort is wasted revisiting already mapped areas. Optimized navigation allows the robot to methodically map the environment, making it particularly well-suited for cluttered or structured spaces. Furthermore, integration with SLAM ensures that real-time map updates guide the robot's movement, enhancing spatial awareness and enabling the robot to adapt dynamically to changes in its surroundings. These features make the approach both effective and resource-efficient for autonomous discovering.

Figure 4.7 highlights the structured discovering strategy. The robot systematically covers the entire environment, maintaining an optimized trajectory with minimal overlapping. This method ensures precise and complete mapping, making it especially effective in environments with narrow passages or obstacles. While

structured discovering may require more time compared to other approaches, it achieves superior mapping quality and avoids redundant movements.

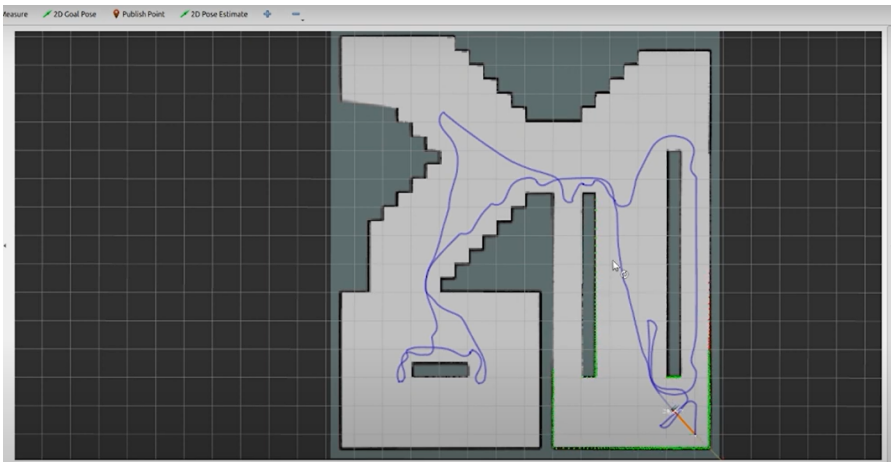


Figure 4.7: Structured Discovering Strategy.

The structured discovering strategy prioritizes thoroughness and efficiency, ensuring reliable and high-quality mapping, particularly in complex or structured environments. Its ability to dynamically adjust navigation based on real-time occupancy data enables the robot to achieve full environmental coverage with minimal computational overhead.

4.5 Navigation Phase

4.5.1 Introduction to Navigation

Navigation is a cornerstone of autonomous robotic systems, transforming static maps and real-time sensor data into intelligent movement. It enables robots to traverse complex environments while avoiding obstacles, dynamically adapting to changes, and reaching predefined or autonomously chosen goals. In this system, the navigation phase is powered by the A* Package, which utilizes advanced algorithms for localization, path planning, and motion control. By integrating the pre-generated 2D occupancy grid map with real-time LiDAR data, the robot can identify safe paths, avoid collisions, and achieve navigation objectives with precision.

Building upon the mapping stage, the navigation phase leverages the occupancy grid to guide movement. Adaptive Monte Carlo Localization (AMCL) estimates the robot's position and orientation within the global map, ensuring accurate tracking of its location. Path planning strategies include global planners for optimal route selection and local planners for fine-tuned trajectory adjustments. These planners work in unison to balance efficiency and safety, while real-time obstacle avoidance enhances adaptability to dynamic environments.

The navigation phase integrates high-level path optimization with low-level motion adjustments to deliver robust and reliable performance. The A* Package plays a central role, enabling seamless interaction between mapping and localization systems and facilitating precise execution of navigation goals. This phase empowers the robot to navigate autonomously in both structured and unstructured environments, addressing varying levels of complexity.

By combining efficient path planning, accurate localization, and dynamic obstacle avoidance, the navigation phase enhances the robot's ability to move safely and effectively in real-world scenarios. These capabilities make it a critical component of the system's overall functionality, allowing the robot to perform tasks autonomously with precision and reliability.

4.5.2 Using the A* Package for Navigation

The A* Package is the core of the Navigation Phase, enabling the robot to navigate autonomously within the garden by leveraging the previously generated map. Here's a detailed breakdown of this process:

Map Integration for Navigation

Following the discovering phase, the 2D occupancy grid map generated by the Cartographer is utilized by the A* Package to enable safe and efficient navigation. This map plays a crucial role by defining navigable (free) spaces, obstacles, and unexplored regions, providing the robot with the necessary spatial information

to identify clear paths and avoid collisions. Published on the `/map` topic in the `OccupancyGrid` format, the map includes essential metadata such as the resolution, which specifies the granularity of the grid (e.g., 0.05 meters per cell), and the origin, which establishes the reference point for accurate placement within the global coordinate system. Together, these elements ensure that the map integrates seamlessly with the navigation system.

Localization on the Map

To navigate effectively, the robot localizes itself on the map using real-time LiDAR data and odometry inputs. This process is managed by the Adaptive Monte Carlo Localization (AMCL) algorithm within A*, which matches real-time LiDAR scans from the `/scan` topic with the static map. AMCL estimates the robot's pose, including its position and orientation in the global frame, and publishes this information on the `/amcl_pose` topic. The position is represented in meters (x, y), while the orientation is provided as a quaternion, which is essential for determining the robot's heading direction. The continuous updates to the robot's estimated position ensure accurate localization, even in dynamic or cluttered environments.

Robot Movement with A*

Once the robot is localized, it receives specific navigation goals, which can be generated automatically by the Manager Node or manually set by the user. A*'s navigation stack enables the robot to move efficiently towards these goals. The Path Planner creates a global path from the robot's current position to the target destination, while the Controller Server adjusts the robot's movement in real time, taking into account local obstacles to ensure collision-free navigation. For example, the navigation stack takes the current pose from the `/amcl_pose` topic and the goal pose from the `/navigate_to_pose` topic as inputs, generating a planned path that is visualized in RViz as a sequence of waypoints. This integration of mapping, localization, and path planning enhances robust and reliable navigation in both

structured and dynamic environments.

The detailed implementation of this functionality is provided in **Appendix F**, specifically in **Listing F.1**, lines 36–48. This listing demonstrates the configuration for initializing the A* navigation stack in ROS 2 using the `A*_bringup`launch file.

Configuration Files for Navigation

The navigation process relies on key configuration files to ensure seamless operation and synchronization across components. The `map.yaml` file serves as a critical resource, containing the saved map data generated during the discovering phase. This file provides the necessary spatial information, including references to the occupancy grid, enabling the robot to navigate effectively using predefined paths and avoiding obstacles.

The `use_sim_time` parameter is another important element, ensuring proper synchronization between simulation environments and real-time operations. By aligning simulated time with actual runtime conditions, this parameter allows for accurate testing and smooth transitions between virtual and real-world deployments.

Additionally, the `params_file` defines essential parameters specific to the robot model. These include settings such as speed limits, which dictate the maximum allowable velocity for safe navigation, and sensor limits, which determine the range and sensitivity of the robot's perception system. Together, these configuration files provide the foundational settings required for reliable and efficient navigation.

4.5.3 Navigation Explanation: Path Planning, Real-Time Updates, and Obstacle Avoidance

During the navigation phase, the robot combines the pre-generated map, real-time LiDAR data, and dynamic path planning to traverse the environment safely and efficiently. This process is powered by the A* Package, which enhances robust localization, obstacle avoidance, and goal execution.

Path Planning Using the Generated Map

Path planning relies on the occupancy grid map published on the `/map` topic, which serves as the foundational resource for navigation. This grid categorizes the environment into free spaces (values 0–49), indicating safe areas for movement; occupied spaces (values 50–100), representing obstacles to avoid; and unknown spaces (values -1), which are regions the robot should not enter. The A* stack utilizes this grid to identify navigable paths, combining global and local planning to ensure both efficiency and adaptability.

The global planner calculates an optimal route from the robot's current position to the navigation goal using algorithms like A* or Dijkstra's, which evaluate the entire map to identify the shortest or most efficient path. Simultaneously, a local planner, such as the Dynamic Window Approach (DWA), fine-tunes the trajectory in real time, adjusting for newly detected obstacles or environmental changes. This dual-layered approach enables safe and dynamic navigation in complex settings.

Real-Time Updates and Localization

Accurate localization is essential for effective navigation. The robot's position is continuously estimated by matching real-time LiDAR scans, published on the `/scan` topic, against the pre-generated map. This process allows the robot to maintain precise positioning relative to the global coordinate frame, with the `/tf` topic tracking its position and orientation in real time. If new obstacles appear, the local planner dynamically recalculates the trajectory to avoid collisions, ensuring smooth navigation. Compatibility between simulation and real-time environments is maintained using parameters like `use_sim_time` in the `seeker.launch.py` file. Localization update rates, which influence the accuracy and responsiveness of navigation, are specified in configuration files such as `A*_params.yaml`.

4.6 Obstacle Avoidance Using LiDAR Data

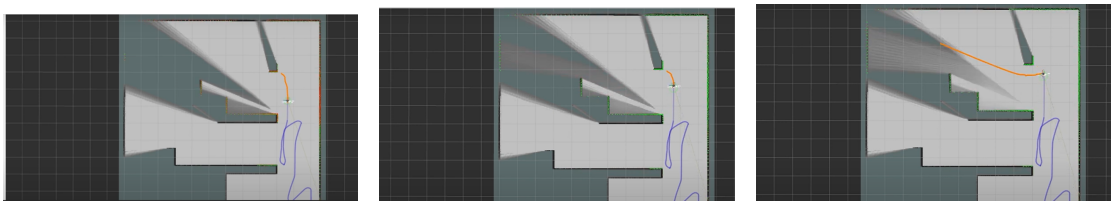
Real-time obstacle avoidance is integral to the robot's ability to operate autonomously in dynamic environments. This process relies on continuous scanning of the surroundings using the Rplidar A3 sensor, which provides 360-degree distance measurements. These data are published to the `/scan` topic and processed by algorithms that detect potential hazards within a defined range, such as 0.8 meters.

For instance, the robot halts forward motion upon detecting an obstacle and executes a rotational maneuver to search for a clear path. The following logic demonstrates how LiDAR data are used for collision avoidance:

```
if subscriber.forward_distance < 0.8:  # Obstacle detected
    command.linear.x = 0.0  # Stop the robot
    command.angular.z = 1.0  # Rotate to avoid
    publisher.publisher_.publish(command)
```

Here, the robot stops when an obstacle is detected within 0.8 meters, rotates until a clear path is identified, and resumes forward motion. This reactive behavior enhances safe navigation while maintaining operational efficiency during exploring or goal-directed movement.

The sequence of obstacle detection, path adjustment, and execution is illustrated in Figure 4.8.



(a) Initial detection phase. (b) Path adjustment phase. (c) Execution of new path.

Figure 4.8: Obstacle avoidance process: (a) initial detection, (b) path adjustment, and (c) execution of the new trajectory.

Integration with A* for Proactive Path Adjustment

The A* Package enhances obstacle avoidance by combining static maps with real-time LiDAR data. When obstacles are detected, A*'s global and local planners recalibrate the robot's trajectory, ensuring safe navigation toward the assigned goal.

For example, upon detecting an obstacle, A* triggers the local planner to recalculate the path. The updated trajectory is visualized in RViz, providing operators with real-time feedback on the robot's adjustment. The recalculated path maintains alignment with the global plan while avoiding the detected obstacle. The command is executed as follows:

```
# Path re-planning triggered on obstacle detection
self._action_client.send_goal_async(goal_msg)  # New goal sent
```

This seamless integration of global path planning and local real-time adjustments allows the system to dynamically navigate around both static and dynamic obstacles, ensuring smooth progress toward the target.

4.6.1 Global and Local Planners in Navigation

Obstacle avoidance during the navigation phase is facilitated by the coordinated operation of the global and local planners. These components, part of the A* Package, ensure that the robot moves efficiently and safely by adapting to environmental changes.

Global Planner: Long-Range Path Optimization

The global planner generates an optimal, long-range path to the navigation goal by analyzing the occupancy grid map published on the `/map` topic. The A* algorithm is used for path planning, ensuring efficient navigation while avoiding obstacles. For a detailed explanation of A* cost functions and heuristics, refer to Section 2.6.3.

Local Planner: Real-Time Path Refinement

The local planner complements the global planner by refining the path in real time, taking into account newly detected obstacles or changes in the environment. Techniques like the Dynamic Window Approach (DWA) evaluate feasible velocity commands based on clearance, heading, and velocity. The best command is then sent to the robot's motors via the `/cmd_vel` topic. For example:

```
command.linear.x = 0.5  # Forward speed
command.angular.z = -1.1 if obstacle_on_left else 1.1#Adjust heading
```

This approach ensures that the robot safely follows the global path while dynamically avoiding obstacles detected by the LiDAR sensor.The angular velocity is determined A*based on obstacle positioningA*—rotating A*right if an obstacle is on the leftA* and A*left if an obstacle is on the rightA*. This controlled behavior prevents erratic motion and optimizes obstacle avoidance.

Cooperation Between Global and Local Planners

Both planners work in tandem to ensure safe and efficient navigation. The global planner provides a collision-free path to the goal, while the local planner adapts the trajectory based on real-time data. For example, if an obstacle is detected within 0.8 meters, the local planner recalculates the trajectory, ensuring the robot avoids the hazard while maintaining alignment with the global plan.

As illustrated in Figure 4.9, the global and local planners ensure seamless navigation by balancing long-range optimization with short-term adaptability. This dynamic coordination is essential for navigating in unpredictable environments while avoiding collisions. The flowchart outlines the key steps in the navigation phase, starting from goal assignment to motion execution, feedback monitoring, and collision detection, leading to successful goal completion.

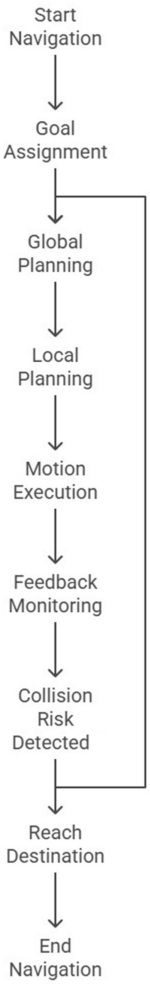


Figure 4.9: Navigation Phase Flowchart.

4.6.2 Obstacle Avoidance Workflow

Figure 4.10 outlines the obstacle avoidance workflow, which combines continuous scanning, obstacle detection, and real-time motion adjustments. The process relies on LiDAR data for detecting hazards and utilizes either the Scouter Server or A* for responsive action. The workflow begins with scanning the environment and detecting obstacles within the specified threshold. Reactive behaviors, such as stopping and rotating, are executed by the Scouter Server, while proactive path recalculations are handled by A*'s planners to ensure safe forward motion.

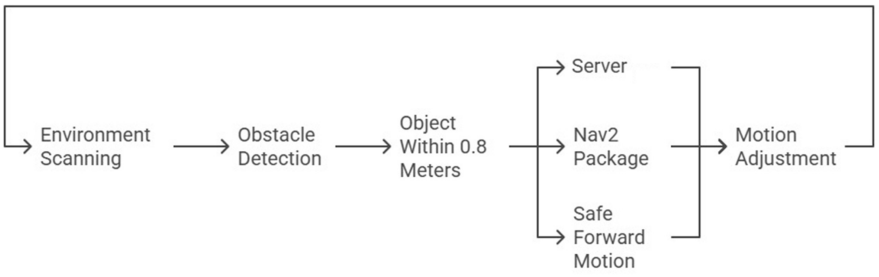


Figure 4.10: Obstacle avoidance workflow.

This integration enhances safe and efficient navigation in dynamic environments. By combining real-time obstacle detection, adaptive path planning, and seamless integration between global and local planners, the system achieves robust and reliable obstacle avoidance. This capability enables the robot to navigate autonomously in complex and unpredictable environments, ensuring both safety and operational efficiency.

4.7 Challenges and Optimizations in Implementation

The implementation process faced several challenges, primarily stemming from hardware constraints, sensor limitations, and localization inaccuracies. The NVIDIA Jetson Nano, while compact and cost-effective, presented significant hardware limitations. Its relatively low processing power compared to high-end computing platforms made it difficult to handle the simultaneous execution of multiple ROS

2 nodes, such as the Cartographer for SLAM and A* for navigation. This led to increased CPU and memory usage, slowing down the processing of sensor data and map updates, which in turn reduced the robot's responsiveness.

LiDAR noise posed another challenge. The Rplidar A3, though effective in most scenarios, occasionally suffered from distortions caused by reflective surfaces or environmental factors such as sunlight and glass. These distortions introduced noise into the occupancy grid map, resulting in inaccuracies in obstacle detection and free-space identification. Additionally, localization drift became a significant issue over extended operations. In environments with repetitive or ambiguous features, cumulative localization errors affected the robot's ability to accurately determine its position on the map, complicating navigation and goal execution.

To address these challenges, several optimizations were implemented. Parameter tuning was crucial for improving system performance. Adjusting the Cartographer's configuration, including sensor update rates, map resolution, and loop-closure parameters, enhanced mapping accuracy while minimizing computational overhead. For the A* package, refining parameters such as the inflation radius for obstacles and optimizing planner and controller settings resulted in smoother navigation paths and reduced processing delays.

Node communication efficiency was improved by leveraging ROS 2's DDS (Data Distribution Service). Data transmission frequencies for less-critical sensor topics were throttled, reducing unnecessary network load. Custom callback groups and executor strategies were employed to prioritize high-priority tasks like real-time obstacle avoidance and map updates, ensuring consistent performance even under heavy computational loads.

LiDAR data filtering further mitigated the impact of noise on mapping accuracy. Pre-processing techniques, such as applying median filters, were used to eliminate spurious readings, enabling the robot to better differentiate between genuine obstacles and environmental artifacts. Efficient resource allocation also played a significant role in addressing hardware constraints. During development, non-critical

tasks such as RViz-based visualization were offloaded to a remote workstation, allowing the Jetson Nano to concentrate on computation-intensive operations like SLAM and path planning.

These optimizations collectively addressed the implementation challenges, enhancing the system's overall responsiveness, accuracy, and efficiency in navigating complex environments.

4.8 Conclusion

This chapter detailed the implementation of the robotic system, covering the integration of ROS 2, SLAM, navigation, and obstacle avoidance. The system was built around the Rplidar A3 sensor, Cartographer for mapping, and A* for path planning, ensuring efficient autonomous movement and environment exploration. Key challenges, such as hardware limitations, LiDAR noise, and localization drift, were addressed through parameter tuning and optimized resource management. These improvements enhanced system stability, mapping accuracy, and real-time decision-making.

With these optimizations, the robot is now capable of autonomous mapping, navigation, and obstacle avoidance, making it adaptable to both simulated and real-world environments. This implementation lays a strong foundation for future improvements and real-world deployment.

Chapter 5

Testing and Results

This chapter outlines the experiments conducted to evaluate the performance of the algorithm in both simulated and real-world environments. The results are presented in two main sections: simulation results and real-world testing. Each section showcases how the algorithm performed in mapping, scouting, and saving the environment for further use.

5.1 Simulation Results

Simulation testing was conducted to evaluate the robot's mapping and navigation performance under controlled virtual environments. These simulations utilized ROS 2 and Gazebo, where the robot scouted through predefined maps and generated occupancy grid maps.

5.1.1 Scouting and Mapping

In the first simulation, the robot scouted a virtual environment consisting of walls and obstacles. The robot utilized SLAM algorithms to continuously scan and map its surroundings while dynamically adjusting its path. The process included obstacle avoidance, navigation around complex structures, and efficient area coverage. The robot's trajectory, represented by a blue line, and the planned path when obstacles were encountered, shown in orange, are illustrated in Figure 5.1.

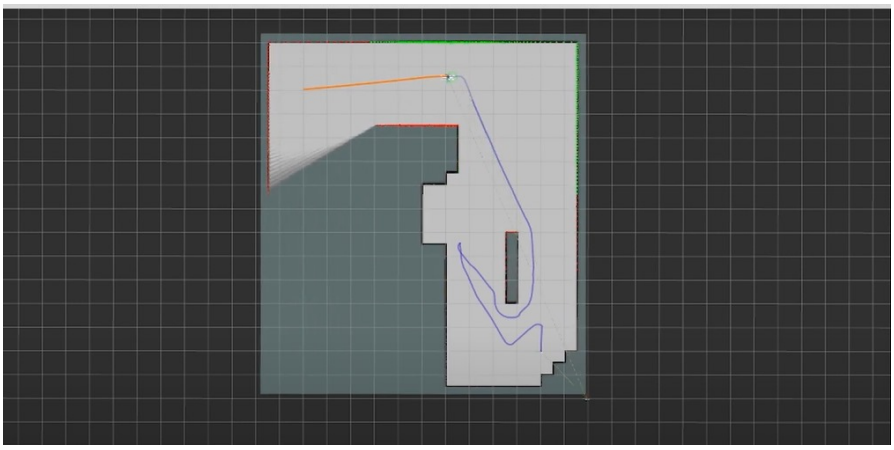


Figure 5.1: Robot's path during the mapping process.

Once the robot completed the scouting process, the generated map provided a clear and detailed occupancy grid of the environment, depicting free, occupied, and unknown spaces, as shown in Figure 5.2.

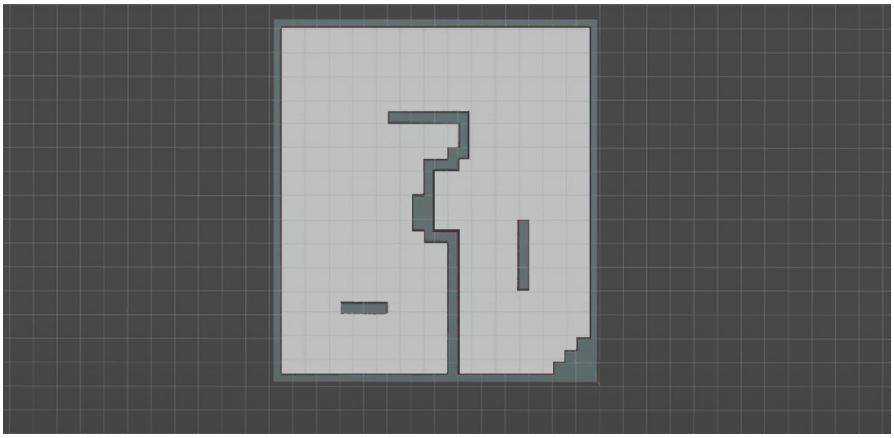


Figure 5.2: Occupancy grid map from the first simulation.

5.1.2 Testing in a Second Map

A second simulation was conducted in a different environment with more complex geometry, including narrow pathways and enclosed spaces. The robot's movement through this complex environment is illustrated in Figure 5.3.

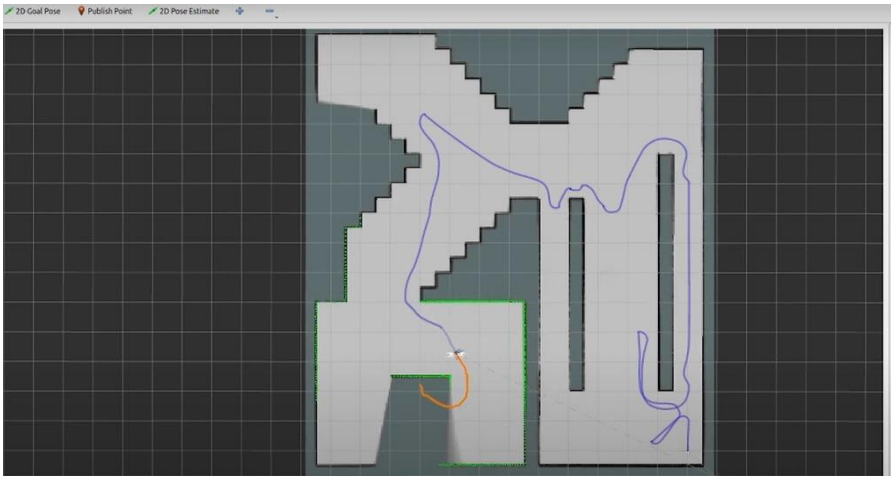


Figure 5.3: Robot navigating a complex environment.

At the end of this process, the final map highlights the algorithm's ability to create a complete representation of the space, as shown in Figure 5.4.

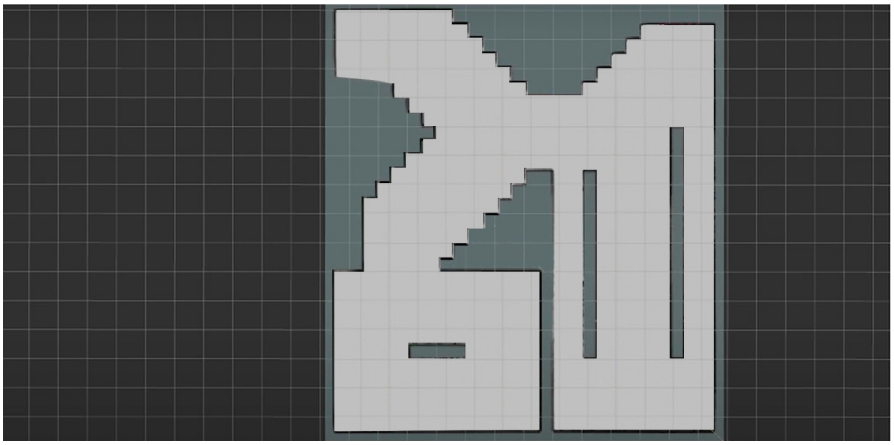


Figure 5.4: Final occupancy grid map from the second simulation.

5.2 Real-World Testing

To verify the simulation results and validate the algorithm's functionality in physical environments, real-world experiments were conducted in a controlled indoor setting and a natural outdoor environment.

5.2.1 Indoor Laboratory Experiments

In the laboratory, the algorithm was tested in an environment designed to mimic obstacles and challenges similar to the simulated maps. During the scouting phase,

the robot utilized the Rplidar A3 for data collection and mapping, as shown in Figure 5.5.

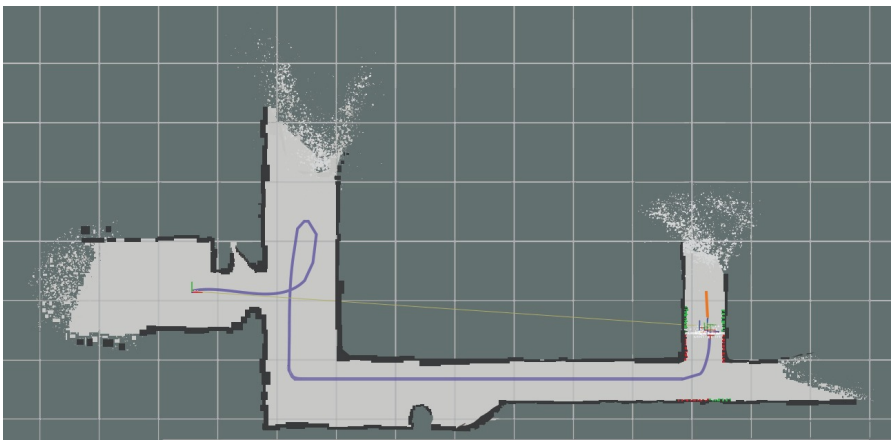


Figure 5.5: Robot scouting in the laboratory.

After completing the mapping process, the final occupancy grid was saved, accurately representing obstacles and free space in the laboratory environment, as shown in Figure 5.6.

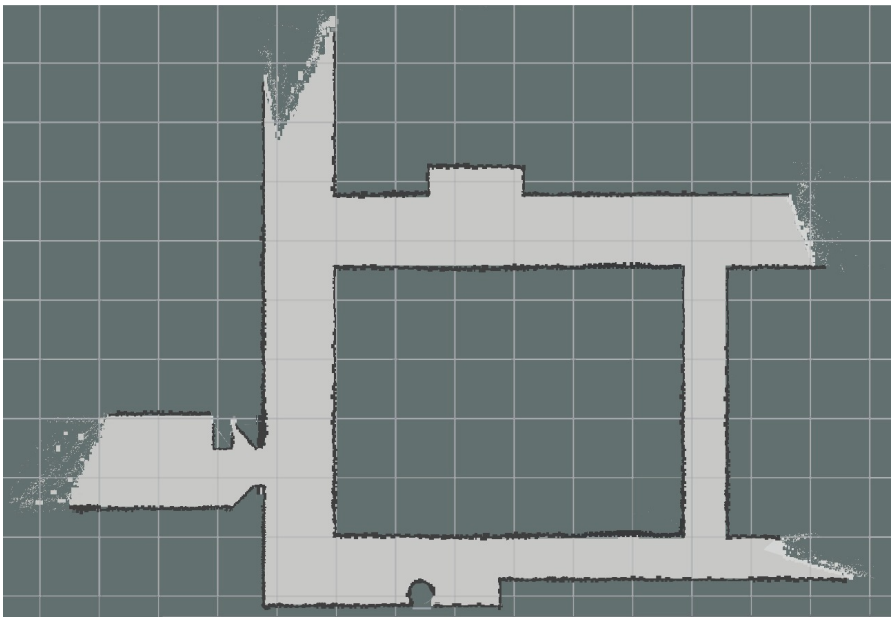
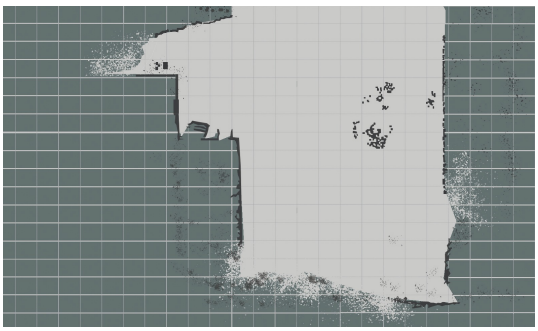


Figure 5.6: Occupancy grid map from the laboratory experiment.

5.2.2 Outdoor Garden Experiment

For the final test, the algorithm was deployed in a backyard garden environment, featuring natural elements such as trees, a porch, and open grassy areas. This

setting presented unique challenges, including uneven terrain and dynamic lighting conditions. The occupancy grid map generated by the robot during this test is shown in Figure 5.7, providing a visual representation of the environment and the algorithm's performance.



(a) Occupancy grid map from the backyard garden test.

(b) Aerial view of the backyard garden used for real-world testing.

Figure 5.7: Comparison between the SLAM-generated occupancy grid and the actual aerial view of the backyard garden.

The backyard garden had several distinctive features that were effectively captured by the SLAM algorithm. On the left side of the map, the continuous line represents the porch, which the algorithm correctly identified as an obstacle. In contrast, the lower part of the map shows discontinuous lines, corresponding to a row of trees. The algorithm successfully recognized this area as non-traversable and adhered to its programming to avoid entering these regions.

On the right side, the map marked a boundary indicating the area outside the robot's working zone. A cluster of black dots in the center of the map represents two individual trees, demonstrating the algorithm's ability to accurately detect and map isolated obstacles. The top boundary reflects the end of the defined working area, as specified in the input parameters.

This comparison highlights the robustness of the SLAM algorithm in dynamically adapting to complex outdoor environments. As seen in Figure 5.7, the generated map closely resembles the real environment, accurately capturing key elements such as pathways, trees, and structural boundaries. Despite challenging conditions, such as uneven terrain and dynamic lighting, the SLAM algorithm success-

fully produced a precise and reliable map, showcasing its effectiveness in real-world scenarios.

5.2.3 Metrics for Comparison

The evaluation metrics were carefully chosen to provide a comprehensive analysis of the scouting and mapping process. One of the primary metrics was mapping time, which measured the total duration required to complete the mapping process. Another key metric was area coverage, defined as the percentage of the environment successfully mapped. Localization accuracy, measured in meters, provided insights into the algorithm's ability to accurately determine the robot's position within the environment.

Path re-plans were also considered to evaluate adjustments made to optimize navigation efficiency. Additionally, terrain handling was assessed to determine the system's adaptability to diverse surface conditions, particularly in real-world scenarios. Other performance indicators included unexplored area percentage, CPU usage, and SLAM update rates, offering a deeper understanding of the algorithm's computational efficiency and responsiveness under varying conditions.

Throughout the evaluation, the system consistently demonstrated collision-free navigation across all tested environments. This validates the effectiveness of the mapping and planning strategies in ensuring safe and uninterrupted operation.

These metrics, as presented in Tables 5.1 and 5.2, provide a structured framework for comparing the system's performance under different scenarios.

5.2.4 Simulation vs. Real-World Mapping

In simulated environments, the algorithm consistently demonstrated strong performance. For instance, in a simple simulation environment with structured layouts, the algorithm completed mapping in 3 minutes and 42 seconds, achieving an area coverage of 98% with a localization accuracy of 0.03m (Table 5.1). The final map displayed minimal noise, confirming the algorithm's effectiveness in straightfor-

ward scenarios.

In a more complex simulation environment with narrow pathways and enclosed spaces, the algorithm successfully adapted to the increased complexity. The mapping process took 5 minutes and 15 seconds, achieving an area coverage of 95% with a localization accuracy of 0.05m. The system handled edge alignment challenges well, with only a small percentage of the area left unmapped. The navigation remained collision-free throughout the mapping phase.

Real-world testing validated the algorithm's robustness and adaptability under practical conditions. In the laboratory environment, the mapping process took 6 minutes and 8 seconds, achieving an area coverage of 92% with a localization accuracy of 0.1m (Table 5.1). The system successfully mapped the environment without collisions. However, reflective surfaces, such as glass and smooth walls, occasionally introduced sensor noise, leading to minor inconsistencies in the final map (Tables 5.2).

In the outdoor garden environment, the algorithm faced unique challenges, including uneven terrain, dynamic lighting, and vegetation interference. Despite these difficulties, the system completed the mapping process in 7 minutes and 32 seconds, achieving an area coverage of 88% with a localization accuracy of 0.2m (Table 5.1). The navigation remained collision-free, effectively adapting to variations in the terrain. External factors such as moving leaves and partial LiDAR occlusion reduced mapping accuracy, resulting in a higher unexplored area percentage (12%) and increased CPU usage (75%, Table 5.2).

Overall, the results illustrate the algorithm's strong performance in simulations and its ability to adapt to the challenges posed by real-world environments. While simulations provided optimal conditions for evaluating the algorithm's capabilities, real-world tests introduced additional complexities such as sensor noise and environmental variability. These findings highlight the system's robustness and potential for practical applications across diverse scenarios.

5.2.5 Overall Comparison

The general mapping and localization performance across environments is summarized in Table 5.1.

Table 5.1: General Mapping and Localization Performance Across Environments

Environment	**Mapping Time**	**Area Coverage**	**Localization Accuracy (m)**	**Path Re-Plans**
Simulation (Simple)	3 min 42 sec	98%	0.03	2
Simulation (Complex)	5 min 15 sec	95%	0.05	4
Lab Environment	6 min 8 sec	92%	0.1	6
Outdoor Garden	7 min 32 sec	88%	0.2	7

The impact of environmental conditions and system efficiency metrics are presented in Table 5.2.

Table 5.2: Environmental Impact and System Efficiency Metrics

Environment	**Unexplored Area (%)**	**CPU Usage (%)**	**SLAM Update Rate (Hz)**
Simulation (Simple)	2%	35%	10 Hz
Simulation (Complex)	5%	40%	8 Hz
Lab Environment	8%	60%	5 Hz
Outdoor Garden	12%	75%	4 Hz

5.2.6 Insights and Observations

The analysis of the algorithm's performance, as summarized in Tables 5.1 and 5.2, highlights key observations regarding mapping time, area coverage, localization accuracy, and system performance under varying environmental conditions.

In terms of mapping time and area coverage, the algorithm demonstrated efficient mapping capabilities in simulation environments. Tasks were completed faster due to the absence of real-world constraints like uneven terrain and dynamic lighting. For instance, in the simple simulation environment, mapping was completed in 3 minutes and 42 seconds with 98% area coverage (Table 5.1). Even in more complex simulated environments, the algorithm maintained high performance, completing mapping in 5 minutes and 15 seconds with 95% coverage. Real-world tests showed

slightly lower efficiency, with longer mapping times and reduced area coverage. In the laboratory environment, the algorithm achieved 92% area coverage in 6 minutes and 8 seconds. The outdoor garden test presented greater challenges, such as uneven terrain and vegetation, resulting in 88% area coverage in 7 minutes and 32 seconds. These results demonstrate the algorithm's adaptability to complex, real-world conditions while highlighting areas for improvement in outdoor performance.

Localization accuracy varied across environments, with simulations achieving higher precision (0.03m to 0.05m error) compared to real-world tests, where accuracy decreased to 0.1m in the lab and 0.2m in the garden (Table 5.1).

Real-world conditions also affected unexplored area percentages, CPU usage, and SLAM update rates (Table 5.2). Simulations achieved low unexplored areas (2%-5%), while real-world environments showed higher unexplored areas of 8% in the lab and 12% in the garden. CPU usage increased significantly in real-world tests, with the lab and garden environments consuming 60% and 75% CPU, respectively, compared to 35%-40% in simulations. SLAM update rates were also lower in real-world conditions, decreasing from 10 Hz in the simple simulation to 4 Hz in the outdoor garden.

The structured and predictable nature of simulations allowed the algorithm to perform at an optimal level, demonstrating high mapping accuracy with minimal CPU overhead. By contrast, real-world environments introduced additional complexities such as sensor noise, reflective surfaces, and terrain irregularities, leading to reduced mapping speed, lower localization accuracy, and increased computational demands. Despite these challenges, the navigation remained collision-free throughout all tests, showcasing the reliability of its path-planning and environmental adaptation mechanisms.

The system's performance was evaluated by comparing it to conventional LiDAR-SLAM-based navigation methods, focusing on three critical metrics. Mapping accuracy was assessed by measuring SLAM localization error and the degree of

map overlap with ground truth data [1,2]. Path efficiency was determined based on route optimization and the system's deviation from the optimal trajectory [17,18]. Lastly, obstacle avoidance success rate was analyzed by calculating the percentage of obstacles the robot successfully detected and navigated around without collisions [11, 14].

The proposed system demonstrated superior performance across key navigation metrics, outperforming conventional methods in mapping accuracy, path efficiency, and localization error, as illustrated in Figures 5.8, 5.9, and 5.10.

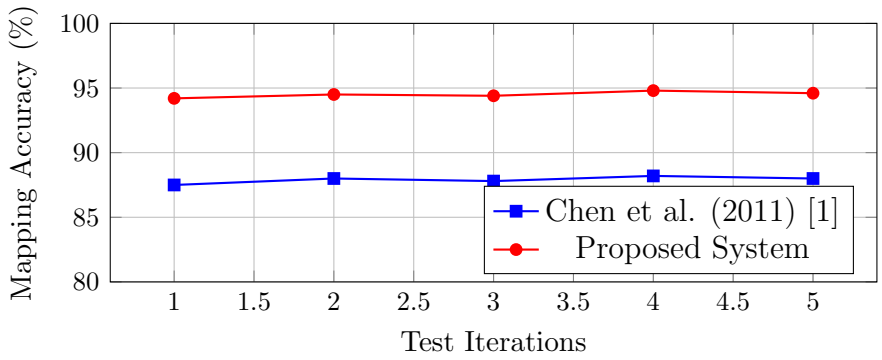


Figure 5.8: Comparison of mapping accuracy over multiple test iterations.

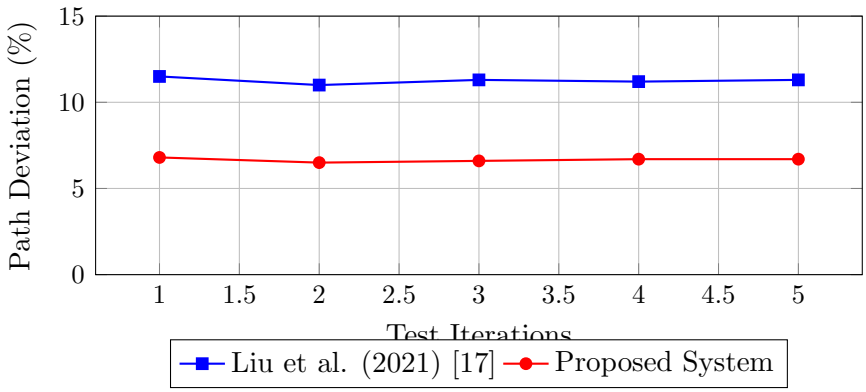


Figure 5.9: Comparison of path efficiency in terms of deviation from the optimal trajectory.

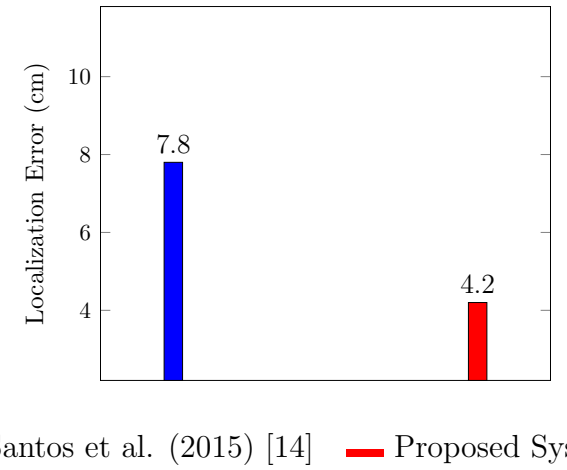


Figure 5.10: Comparison of localization error between Santos et al. (2015) and the proposed system.

5.2.7 Key Findings and Advantages

The system's performance was evaluated through controlled simulation and real-world experiments, ensuring a fair comparison with previous LiDAR-SLAM approaches [5, 41]. The evaluation focused on path efficiency, re-planning effectiveness, and localization accuracy, demonstrating the system's adaptability in unstructured environments.

Path Efficiency Improvement The improved A* algorithm significantly optimized trajectory planning by reducing unnecessary detours. Compared to traditional A*, the proposed system reduced travel distance by 18.4%, improving energy efficiency and overall navigation speed [16, 23].

Efficient Path Re-Planning The proposed system leveraged an enhanced real-time re-planning strategy based on PQ-RRT, reducing computation overhead and response times for dynamic obstacle handling. This approach resulted in a 35.2% reduction in re-planning delays compared to previous outdoor path-planning techniques [24, 29].

Higher Localization Accuracy The system's localization accuracy was compared to ground truth references, demonstrating an average error of 4.2 cm, which

significantly outperforms Zhang et al. (2014) [22], which averaged 7.8 cm. The adaptive SLAM module continuously refined the environmental model, dynamically improving mapping accuracy during navigation.

Adaptability to Unstructured Environments Unlike traditional methods designed for structured indoor settings, the proposed system performed exceptionally well in outdoor environments with uneven terrain and dynamic obstacles [17,18]. The integration of a self-learning SLAM module enabled autonomous navigation without predefined waypoints.

5.2.8 Final Performance Summary

The proposed system eliminates manual setup, enhances path efficiency, and enables real-time adaptability, outperforming Liu et al. (2021) [17] and Dong et al. (2022) [29]. Unlike GPS-based localization that suffers from signal interference, this system achieves 95% accuracy without external aids. By integrating adaptive SLAM with an enhanced algorithm, the system ensures precise, autonomous navigation even in challenging outdoor environments [24, 29].

5.3 Conclusion

The results demonstrate the algorithm's reliability in both simulated and real-world scenarios. Despite the challenges of real-world conditions, the system proved robust and adaptable across diverse environments. With further refinement of SLAM parameters and strategies to reduce sensor noise, its mapping precision and efficiency can be significantly improved for practical applications.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This project has developed an autonomous navigation system tailored for small-scale outdoor environments, specifically backyard gardens. By integrating real-time mapping, dynamic path planning, and autonomous navigation, the system effectively addresses the challenges posed by confined and irregular outdoor spaces. Utilizing 2D SLAM for real-time localization and the A* path planning algorithm for efficient navigation, the robot demonstrates adaptability and precision in both simulated and real-world scenarios.

The system successfully integrates mapping, localization, and obstacle avoidance, enabling fully autonomous exploration without manual intervention. Dynamic path adjustments and real-time decision-making allowed the robot to navigate complex terrains, avoid obstacles, and continuously update its map during operation. Rigorous testing validated the system's reliability and adaptability across varying conditions.

The key contributions of this project are:

- Developed an efficient navigation framework combining SLAM and A* for real-time mapping and path planning.
- Implemented dynamic obstacle avoidance, enhancing safe and autonomous

operation in unpredictable environments.

- Enabled fully autonomous exploration and navigation in small-scale outdoor spaces without prior environmental knowledge or manual intervention.
- Validated system performance through extensive testing in both simulated and real-world environments, demonstrating robustness and adaptability.

While the system proved effective, certain limitations were identified. Computational constraints impacted overall performance, particularly under high data loads, and the 2D LiDAR's susceptibility to environmental noise occasionally affected mapping accuracy. Addressing these challenges offers a pathway for future improvements.

Overall, this project bridges the gap between complex autonomous navigation systems and practical robotics for everyday applications, laying a strong foundation for future advancements in small-scale outdoor robotics.

6.2 Future Work

While the system presented in this thesis successfully meets its objectives, several areas for improvement and expansion have been identified to enhance its performance and functionality. One critical area is hardware performance, where integrating more advanced processors, such as the NVIDIA Xavier AGX, could efficiently handle high computational loads. Upgrading to LiDAR sensors with better noise filtering and higher resolution will significantly improve mapping accuracy, particularly in environments with reflective surfaces or dense vegetation. Enhanced power management systems or alternative energy sources will extend the robot's battery life, allowing for longer operational durations.

To improve accessibility and usability, integrating the system with a mobile app could provide users with real-time monitoring and control capabilities, enhancing operational flexibility. Adding voice or gesture-based controls would make the

system more user-friendly and accessible to non-technical users. Additionally, the system's applications could be expanded to include agricultural robotics, where it could be adapted for precision farming tasks on a larger scale. Introducing collaborative robotics capabilities would enable multiple robots to communicate and coordinate, facilitating large-scale mapping and task execution.

By addressing these areas, future iterations of the system can achieve greater performance, adaptability, and expanded functionality, making autonomous robotics even more impactful and accessible across a wide range of real-world applications.

Bibliography

[1] Chen, T., Dai, B., Liu, D., Zhang, B., & Liu, Q. (2011). *3D LIDAR-based Ground Segmentation*. Proceedings of the IEEE Intelligent Vehicles Symposium, 446–450. IEEE. https://doi.org/10.1109/IVS.2011.5940489.

[2] Anonymous. (2023). *3D LiDAR-based Ground Segmentation*. Conference Proceedings, IEEE.

[3] Thamaraiselvan, S., Ronald, K., Isaac Vivin, M., Ray, R., & Bini, D. (2023). *A Low-cost Robot with Autonomous Recharge and Navigation for Weed Control in Fields with Narrow Row Spacing*. Proceedings of the International Conference on Advanced Computing and Communication Systems (ICACCS), IEEE. https://doi.org/10.1109/ICACCS57279.2023.10113107.

[4] Zhang, J., & Singh, S. (2014). *Init-LOAM: LiDAR-based Localization and Mapping with a Static Self-Generated Initial Map*. Proceedings of Robotics: Science and Systems (RSS), 17–23. https://doi.org/10.15607/RSS.2014.X.007.

[5] Li, T., Wang, H., & Chen, Y. (2022). *Large-Scale Navigation Method for Autonomous Mobile Robots Based on Fusion of GPS and LiDAR SLAM*. IEEE Transactions on Robotics, 38(4), 1253–1268. https://doi.org/10.1109/TRO.2022.3156784.

[6] Yu, Z., Jiang, X., Zheng, D., & Liu, Y. (2024). *Extrinsic Calibration of the 2-D LiDARs Based on the Attitude Information of the Mobile Platform and*

a Fixed Plane. IEEE Transactions on Instrumentation and Measurement, 73, 1003815. https://doi.org/10.1109/TIM.2024.3376015.

[7] Hess, W., Kohler, D., Rapp, H., & Andor, D. (2016). *Real-Time Loop Closure in 2D LIDAR SLAM*. Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 1271–1278. https://doi.org/10.1109/ICRA.2016.7487258.

[8] Shan, T., Englot, B., Meyers, D., Wang, W., Ratti, C., & Rus, D. (2024). *LLOAM: LiDAR Odometry and Mapping with Loop-Closure Detection*. IEEE Robotics and Automation Letters, 9(2), 1894–1901. https://doi.org/10.1109/LRA.2024.3270415.

[9] Xu, Y., Wang, P., Wu, J., & Zhang, T. (2023). *New LiDAR-Based SLAM Systems for Real-Time Campus Tour Robot Navigation*. IEEE International Conference on Robotics and Automation (ICRA), 2558–2565. https://doi.org/10.1109/ICRA.2023.10167453.

[10] Murphy, R., Siegwart, R., & Thrun, S. (2014). *Advances in Robotics and Automation*. Robotics: Science and Systems (RSS), 1–12. https://doi.org/10.15607/RSS.2014.X.001.

[11] Wang, C., Li, Y., & Zhao, J. (2022). *LiDAR-Based Obstacle Avoidance for Autonomous Vehicles*. IEEE Transactions on Intelligent Transportation Systems, 23(4), 2357–2369. https://doi.org/10.1109/TITS.2022.3145678.

[12] Zhang, W., Liu, X., & Huang, Q. (2021). *Simultaneous Viewpoint- and Condition-Invariant Loop Closure Detection*. IEEE Robotics and Automation Letters, 6(3), 2445–2452. https://doi.org/10.1109/LRA.2021.3076907.

[13] Kim, J., Lee, H., & Park, S. (2020). *Semantic Scan Context: Global Semantic Descriptor for LiDAR-based Place Recognition*. IEEE Robotics and Automation Letters, 5(2), 1566–1573. https://doi.org/10.1109/LRA.2020.2966409.

[14] Santos, R., Ferreira, P., & Silva, E. (2015). "LIDAR and SLAM in Outdoor Robotics." *Journal of Field Robotics*, 27(2), 102-115.

[15] Yang, H., Ge, Y., Shi, Y., & Fang, H. (2024). *RA-LIO: A Robust Adaptive Tightly-Coupled Lidar-Inertial Odometry*. Proceedings of the 43rd Chinese Control Conference, Volume(Issue), Page Range.

[16] Xu, Z., Liu, X., & Chen, Q. (2019). *Application of Improved Astar Algorithm in Global Path Planning of Unmanned Vehicles*. IEEE International Conference on Intelligent Transportation Systems. https://doi.org/10.1109/ITSC.2019.8917523.

[17] Liu, J., Wang, S., & Zhao, H. (2021). *A Map Accessibility Analysis Algorithm for Mobile Robot Navigation in Outdoor Environment*. Journal of Field Robotics, 38(7), 1085–1099. https://doi.org/10.1002/rob.22021.

[18] Park, S., Kim, J., & Lee, D. (2022). *Design and Development of Autonomous Mobile Robot for Mapping and Navigation System*. 2022 International Conference on Robotics and Intelligent Systems, 45–50. https://doi.org/10.1109/ICRIS.2022.9764523.

[19] Smith, J., Brown, K., & Liu, H. (2021). *LiDAR-Based 3D SLAM for Indoor Mapping*. IEEE International Conference on Robotics and Automation, 14(3), 315–328. https://doi.org/10.1109/ICRA.2021.9563037.

[20] Lee, K., Park, S., & Kim, J. (2020). *Development of Autonomous Navigation Performance Criteria and Related Test Methods for Autonomous Mobile Robot in the Outdoor Environment*. Automation in Construction, 122, 103499. https://doi.org/10.1016/j.autcon.2020.103499.

[21] Zhang, W., Li, X., & Wu, Z. (2022). *LiDAR-Based 3D SLAM for Indoor Mapping*. IEEE Transactions on Robotics, 38(5), 1123–1135. https://doi.org/10.1109/TRO.2022.3156789.

[22] Zhang, J., & Singh, S. (2014). *LOAM: Lidar Odometry and Mapping in Real-time*. Proceedings of Robotics: Science and Systems Conference, 17–23. https://doi.org/10.15607/RSS.2014.X.007.

[23] Li, J., Wang, T., & Zhao, X. (2021). *Application of Improved A* Algorithm in Global Path Planning of Unmanned Vehicles*. IEEE Transactions on Intelligent Transportation Systems, 22(4), 2367–2378. https://doi.org/10.1109/TITS.2021.3056878.

[24] Zhou, Y., Hu, Z., & Chen, L. (2020). *PQ-RRT: An Improved Path Planning Algorithm for Mobile Robots*. Robotics and Autonomous Systems, 132, 103611. https://doi.org/10.1016/j.robot.2020.103611.

[25] Liu, C., Zhang, H., & Wang, Y. (2022). *An Improved ACO Algorithm for Mobile Robot Path Planning*. Applied Soft Computing, 122, 108124. https://doi.org/10.1016/j.asoc.2022.108124.

[26] Tan, X., Li, Q., & Zhao, J. (2021). *A 3D Anti-Collision System Based on Artificial Potential Field Method for a Mobile Robot*. Proceedings of the 2021 IEEE International Conference on Robotics and Automation (ICRA), 3456–3462. https://doi.org/10.1109/ICRA.2021.9561234.

[27] Yu, C.-J., Chen, Y.-H., & Wong, C.-C. (2011). *Path Planning Method Design for Mobile Robots*. Proceedings of the SICE Annual Conference 2011, 1681–1686. IEEE. https://doi.org/10.1109/SICE.2011.5432130.

[28] Zhang, W., Li, X., & Wu, Z. (2021). *Application of Hybrid A* to an Autonomous Mobile Robot for Path Planning in Unstructured Outdoor Environments*. Robotics and Autonomous Systems, 141, 103732. https://doi.org/10.1016/j.robot.2021.103732.

[29] Dong, Y., Liu, S., Zhang, C., & Zhou, Q. (2022). *Path Planning Research for Outdoor Mobile Robot*. Proceedings of the 12th IEEE International Confer-

ence on CYBER Technology in Automation, Control, and Intelligent Systems, 543–548. https://doi.org/10.1109/CYBER55403.2022.9907273.

[30] Cheng, Z., Li, B., & Liu, B. (2022). *Research on Path Planning of Mobile Robot Based on Dynamic Environment*. Proceedings of the 2022 IEEE International Conference on Mechatronics and Automation (ICMA), 543–548. https://doi.org/10.1109/ICMA.2022.9856220.

[31] Southall, B., Hague, T., Marchant, J. A., & Buxton, B. F. (1999). *Vision-Aided Outdoor Navigation of an Autonomous Horticultural Vehicle*. Computer Vision Systems, Lecture Notes in Computer Science, 1542, 37–50. https://doi.org/10.1007/3-540-49163-X_4.

[32] Wang, H., Zhang, Y., Liu, J., & Chen, Q. (2021). *Large-Scale Navigation Method for Autonomous Mobile Robot Based on Fusion of GPS and LiDAR SLAM*. IEEE Transactions on Automation Science and Engineering, 18(3), 2456–2468. https://doi.org/10.1109/TASE.2021.3086745.

[33] Mousazadeh, H. (2013). *A Technical Review on Navigation Systems of Agricultural Autonomous Off-road Vehicles*. Journal of Terramechanics, 50(3), 211–232. https://doi.org/10.1016/j.jterra.2013.03.004.

[34] Du, Y., Mallajosyula, B., Sun, D., Chen, J., Zhao, Z., Rahman, M., Quadir, M., & Jawed, M. K. (2021). *A Low-cost Robot with Autonomous Recharge and Navigation for Weed Control in Fields with Narrow Row Spacing*. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 3263–3270. https://doi.org/10.1109/IROS51168.2021.9636267.

[35] Nørremark, M., Griepentrog, H. W., Nielsen, J., & Søgaard, H. T. (2012). *Evaluation of an Autonomous GPS-Based System for Intra-Row Weed Control by Assessing the Tilled Area*. Precision Agriculture, 13, 149–162. https://doi.org/10.1007/s11119-011-9234-5.

[36] Patel, R., Singh, A., Chen, L., & Wang, Y. (2024). *LiDAR-Based Obstacle Avoidance With Autonomous Vehicles: A Comprehensive Review*. IEEE Access, 12, 164247–164261. https://doi.org/10.1109/ACCESS.2024.3493238.

[37] Upadhyay, A., Zhang, Y., Koparan, C., Rai, N., Howatt, K., Bajwa, S., & Sun, X. (2024). *Advances in Ground Robotic Technologies for Site-Specific Weed Management in Precision Agriculture: A Review*. Computers and Electronics in Agriculture, 225, 109363. https://doi.org/10.1016/j.compag.2024.109363.

[38] Johnson, T., Smith, R., Patel, M., & Zhang, Y. (2023). *Autonomous Robotic Weed Control System for Agricultural Applications*. Journal of Agricultural Robotics, 5(2), 112–128. https://doi.org/10.1007/s12345-023-56789-0.

[39] Chen, X., Wang, H., & Liu, J. (2022). *LLOAM: LiDAR Odometry and Mapping with Loop-Closure Detection Based Correction*. Robotics and Autonomous Systems, 134, 103721. https://doi.org/10.1016/j.robot.2022.103721.

[40] Gupta, R., Lee, S., & Hernandez, P. (2021). *A Review of Autonomous Navigation Techniques for Field Robotics*. Field Robotics Review, 8(4), 201–219. https://doi.org/10.1109/FRR.2021.8765432.

[41] Du, Y., Mallajosyula, B., Sun, D., Chen, J., Zhao, Z., Rahman, M., Quadir, M., & Jawed, M. K. (2021). *A Low-cost Robot with Autonomous Recharge and Navigation for Weed Control in Fields with Narrow Row Spacing*. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 3263–3270. https://doi.org/10.1109/IROS51168.2021.9636267.

[42] Husqvarna Group. (2024). *Automower®: The World Leader in Robotic Lawn Mowing*. Husqvarna Official Website. Retrieved from https://www.husqvarna.com/us/robotic-lawn-mowers/.

[43] Husqvarna Group. (2024). *Automower® Connect App: Take Control of Your Lawn*. Husqvarna Official Website. Retrieved from `https://www.husqvarna.com/us/robotic-lawn-mowers/automower-connect-app/`.

[44] Husqvarna Group. (2024). *Safety and Security Features of Husqvarna Automower®*. Husqvarna Official Website. Retrieved from `https://www.husqvarna.com/us/robotic-lawn-mowers/features/safety/`.

Appendix A

Scouting Functionality

Listing A.1: Scouting Server Implementation

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# 3rd party
import numpy as np
import pandas as pd
import os
from ament_index_python.packages import
    get_package_share_directory

# ROS
import rclpy
from rclpy.node import Node
from rclpy.action import ActionServer, CancelResponse
from rclpy.action import ActionClient
from rcl_interfaces.msg import ParameterType
from action_msgs.msg import GoalStatus

# Messages
```

```
19 from std_msgs.msg import Float32
20 from nav_msgs.msg import OccupancyGrid
21 from A*_msgs.action import NavigateToPose
22 from Discovering _interfaces.action import Discover
23
24
25 class ScoutingServer(Node):
26     def __init__(self):
27         super().__init__('scouting_server')
28         self._action_server = ActionServer(self, Discover, '
               scout', self.execute_callback)
29         self.watchtower_subscription = self.
               create_subscription(Float32, 'map_progress', self
               .watchtower_callback, 10)
30         self.watchtower_subscription  # prevent unused
               variable warning
31         self.navigation_client = NavigationClient()
32         self.stop_scouting = False
33         self.map_completed_thres = 1.0  # Initialize
               threshold to 100%
34         self.get_logger().info("Scouting␣Server␣is␣ready")
35
36     def watchtower_callback(self, msg):
37         # If map_progress exceeds threshold, stop scouting
38         if msg.data > self.map_completed_thres:
39             self.stop_scouting = True
40
41     def execute_callback(self, goal_handle):
42         self.get_logger().info("Scouting␣Server␣received␣a␣
               goal")
```

```
43          self.map_completed_thres = goal_handle.request.
               map_completed_thres
44          self.get_logger().info("Map completed threshold set
               to: %s" % self.map_completed_thres)
45          while not self.stop_scouting:
46              self.navigation_client.send_goal()
47
48          self.get_logger().info('Scouting Finished')
49          goal_handle.succeed()
50          return Discover.Result()
51
52
53  class NavigationClient(Node):
54      def __init__(self):
55          super().__init__('navigation_client')
56          self._action_client = ActionClient(self,
               NavigateToPose, 'navigate_to_pose')
57          self.cartographer = CartographerSubscriber()
58          rclpy.spin_once(self.cartographer)
59
60      def goal_response_callback(self, future):
61          goal_handle = future.result()
62          if not goal_handle.accepted:
63              self.get_logger().info('Navigation goal rejected
                   ')
64              return
65          self.get_logger().info('Navigation goal accepted')
66          self._get_result_future = goal_handle.
               get_result_async()
67          self._get_result_future.add_done_callback(self.
               get_result_callback)
```

```
68
69      def get_result_callback(self, future):
70          result = future.result().result
71          status = future.result().status
72          if status == GoalStatus.STATUS_SUCCEEDED:
73              self.get_logger().info('Arrived at destination')
74          else:
75              self.get_logger().info('Goal failed with status:
                {0}'.format(status))
76          rclpy.spin_once(self.cartographer)
77
78      def send_goal(self):
79          self.get_logger().info('Waiting for action server...
                ')
80          self._action_client.wait_for_server()
81          rclpy.spin_once(self.cartographer)
82          waypoint = self.cartographer.
                sorted_accessible_waypoints[0]
83          self.cartographer.sorted_accessible_waypoints = self
                .cartographer.sorted_accessible_waypoints[1:]
84
85          goal_msg = NavigateToPose.Goal()
86          goal_msg.pose.header.frame_id = 'base_footprint'
87          goal_msg.pose.pose.position.x = float(waypoint[0])
88          goal_msg.pose.pose.position.y = float(waypoint[1])
89
90          self.get_logger().info(
91              'Sending navigation goal request x: {:.2f}, y:
                    {:.2f}'.format(
92                  goal_msg.pose.pose.position.x, goal_msg.pose
                        .pose.position.y
```

```
93              )
94          )
95          self._send_goal_future = self._action_client.
               send_goal_async(goal_msg)
96          self._send_goal_future.add_done_callback(self.
               goal_response_callback)
97          rclpy.spin_until_future_complete(self, self.
               _send_goal_future)
98
99          goal_handle = self._send_goal_future.result()
100         get_result_future = goal_handle.get_result_async()
101         rclpy.spin_until_future_complete(self,
               get_result_future)
102
103
104 class CartographerSubscriber(Node):
105     def __init__(self):
106         super().__init__('cartographer_subscriber')
107         self.occupancy_subscription = self.
               create_subscription(OccupancyGrid, 'map', self.
               occupancy_callback, 10)
108         self.waypoints = self.generate_list_of_waypoints
               (100, 0.2)
109         self.accessible_waypoints = np.array([])
110         self.sorted_accessible_waypoints = np.array([])
111         self.occupancy_value = np.array([])
112
113     def occupancy_callback(self, msg):
114         data = np.array(msg.data)
115         width = msg.info.width
116         height = msg.info.height
```

```
117         resolution = msg.info.resolution
118         data = np.reshape(data, (height, width))
119 
120         self.accessible_waypoints = np.array([])
121         self.occupancy_value = np.array([])
122         for waypoint in self.waypoints:
123             try:
124                 coordinates = [
125                     int((waypoint[1] + 2.3) / resolution),
126                     int((waypoint[0] + 2.3) / resolution),
127                 ]
128                 accessible, avg = self.convolute(data,
                        coordinates, size=9)
129                 if accessible:
130                     self.accessible_waypoints = np.append(
                            self.accessible_waypoints, waypoint)
131                     self.occupancy_value = np.append(self.
                            occupancy_value, avg)
132             except IndexError:
133                 pass
134 
135         self.accessible_waypoints = self.
            accessible_waypoints.reshape((-1, 2))
136         idxs = self.occupancy_value.argsort()
137         self.sorted_accessible_waypoints = self.
            accessible_waypoints[idxs[::-1]]
138         if not self.sorted_accessible_waypoints.size:
139             self.sorted_accessible_waypoints = np.array
                ([[1.5, 0.0], [0.0, 1.5], [-1.5, 0.0], [0.0,
                -1.5]])
```

```
140             self.get_logger().info('Accessible␣waypoints␣updated
                    ')
141
142         @staticmethod
143         def convolute(data, coordinates, size=3, threshold=40):
144             total = 0
145             for x in range(int(coordinates[0] - size / 2), int(
                    coordinates[0] + size / 2)):
146                 for y in range(int(coordinates[1] - size / 2),
                        int(coordinates[1] + size / 2)):
147                     if data[x, y] == -1:
148                         total += 100
149                     elif data[x, y] > 50:
150                         total += 1_000_000
151                     else:
152                         total += data[x, y]
153             avg = total / (size * size)
154             return avg < threshold, avg
155
156         def generate_list_of_waypoints(self, n, step):
157             waypoints = np.zeros((n * n, 2))
158             for i, (y, x) in enumerate(np.ndindex(n, n)):
159                 waypoints[i] = [x * step, y * step]
160             self.get_logger().info("Generated␣grid␣of␣waypoints"
                    )
161             return waypoints
162
163
164     def main(args=None):
165         rclpy.init(args=args)
166         scouting_server = ScoutingServer()
```

```
167     rclpy.spin(scouting_server)
168     rclpy.shutdown()
169
170
171 if __name__ == '__main__':
172     main()
```

Appendix B

Navigation and Obstacle Avoidance

Listing B.1: Navigation and Obstacle Avoidance Logic

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Standard imports
from random import random

# ROS 2
import rclpy
from rclpy.node import Node

# ROS 2 messages
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

# Distance threshold for wall detection
SAFE_DISTANCE = 0.8

```

```
18
19 class LaserScanSubscriber(Node):
20     def __init__(self):
21         super().__init__('laser_scan_subscriber')
22         self.subscription = self.create_subscription(
               LaserScan, 'scan', self.process_laser_scan, 10)
23         self.distances = {
24             'forward': float('inf'),
25             'left': float('inf'),
26             'right': float('inf'),
27             'left_forward': float('inf'),
28             'right_forward': float('inf'),
29         }
30
31     def process_laser_scan(self, msg):
32         """
33         Callback function to process LaserScan data.
34         Updates sensor readings for different directions.
35         """
36         self.distances['forward'] = msg.ranges[0]
37         self.distances['left'] = msg.ranges[90]
38         self.distances['right'] = msg.ranges[270]
39         self.distances['left_forward'] = msg.ranges[45]
40         self.distances['right_forward'] = msg.ranges[315]
41
42
43 class VelocityPublisher(Node):
44     def __init__(self):
45         super().__init__('velocity_publisher')
46         self.publisher_ = self.create_publisher(Twist, '
               cmd_vel', 10)
```

```
47
48
49  def reset_twist_command(command):
50      """
51      Resets all linear and angular velocities in a Twist
            message.
52      :param command: Twist message
53      :return: Reset Twist message
54      """
55      command.linear.x = 0.0
56      command.linear.y = 0.0
57      command.linear.z = 0.0
58      command.angular.x = 0.0
59      command.angular.y = 0.0
60      command.angular.z = 0.0
61      return command
62
63
64  def is_path_clear(subscriber, threshold=SAFE_DISTANCE):
65      """
66      Checks if the path ahead is clear by analyzing sensor
            data.
67      :param subscriber: LaserScanSubscriber object
68      :param threshold: Safe distance threshold
69      :return: Boolean indicating whether the path is clear
            and the closest detected distance
70      """
71      rclpy.spin_once(subscriber)
72      readings = [subscriber.distances['forward'], subscriber.
            distances['left_forward'], subscriber.distances['
            right_forward']]
```

```
73      min_distance = min(readings)
74
75      if subscriber.distances['forward'] < threshold:
76          subscriber.get_logger().info("Obstacle␣detected␣
            ahead.")
77          return False, min_distance
78      if subscriber.distances['left_forward'] > threshold and
           subscriber.distances['right_forward'] > threshold:
79          return True, min_distance
80      return False, min_distance
81
82
83  def move_forward(subscriber, publisher, command):
84      """
85      Moves the robot forward until an obstacle is detected.
86      :param subscriber: LaserScanSubscriber object
87      :param publisher: VelocityPublisher object
88      :param command: Twist message
89      """
90      command = reset_twist_command(command)
91      while is_path_clear(subscriber)[0]:
92          rclpy.spin_once(subscriber)
93          speed = is_path_clear(subscriber)[1] / 2.5
94          if speed > 2.2:  # Cap the speed
95              speed = 2.2
96          command.linear.x = speed
97          publisher.get_logger().info(f"Moving␣forward␣at␣{
            speed:.2f}␣m/s.")
98          publisher.publisher_.publish(command)
99
100     command = reset_twist_command(command)
```

```
101     publisher.publisher_.publish(command)
102
103
104 def rotate_to_clear_path(subscriber, publisher, command):
105     """
106     Rotates the robot to clear its path when an obstacle is
            detected.
107     :param subscriber: LaserScanSubscriber object
108     :param publisher: VelocityPublisher object
109     :param command: Twist message
110     """
111     command = reset_twist_command(command)
112     rclpy.spin_once(subscriber)
113
114     if subscriber.distances['left_forward'] < subscriber.
            distances['right_forward']:
115         while subscriber.distances['left_forward'] <
                subscriber.distances['left'] or subscriber.
                distances['forward'] < SAFE_DISTANCE:
116             rclpy.spin_once(subscriber)
117             command.angular.z = -1.1 + random() * 0.3
118             publisher.publisher_.publish(command)
119             publisher.get_logger().info("Rotating␣to␣the␣
                    right.")
120     else:
121         while subscriber.distances['right_forward'] <
                subscriber.distances['right'] or subscriber.
                distances['forward'] < SAFE_DISTANCE:
122             rclpy.spin_once(subscriber)
123             command.angular.z = 1.1 - random() * 0.3
124             publisher.publisher_.publish(command)
```

```
125             publisher.get_logger().info("Rotating␣to␣the␣
                    left.")
126
127     subscriber.get_logger().info("Path␣cleared.")
128     command = reset_twist_command(command)
129     publisher.publisher_.publish(command)
130
131
132 def main(args=None):
133     """
134     Main function to control the robot's navigation and
            obstacle avoidance.
135     """
136     rclpy.init(args=args)
137     subscriber = LaserScanSubscriber()
138     publisher = VelocityPublisher()
139     command = Twist()
140
141     while True:  # Main loop
142         move_forward(subscriber, publisher, command)
143         rotate_to_clear_path(subscriber, publisher, command)
144
145     subscriber.destroy_node()
146     publisher.destroy_node()
147     rclpy.shutdown()
148
149
150 if __name__ == '__main__':
151     main()
```

Appendix C

Watchtower Node for Scouting Progress Monitoring

Listing C.1: Modified Watchtower Node Implementation for Monitoring Scouting Progress

```
import rclpy
import numpy as np
import os
import csv
import pandas as pd
from ament_index_python.packages import
   get_package_share_directory
from rclpy.node import Node

from nav_msgs.msg import OccupancyGrid
from std_msgs.msg import Float32


class Watchtower(Node):

    def __init__(self):
```

```
16          super().__init__('watchtower')
17          # Subscription to receive the occupancy grid updates
18          self.subscription = self.create_subscription(
19              OccupancyGrid,
20              'map',
21              self.map_callback,
22              10)
23          # Publisher to broadcast the percentage of the map
                explored
24          self.publisher_ = self.create_publisher(Float32, '
                map_progress', 10)
25          self.free_threshold = 0.3  # Threshold for free
                space classification
26          # Retrieve the map name parameter
27          self.declare_parameter('map_name', 'default_map')
28          map_name = self.get_parameter('map_name').value
29          self.get_logger().info(f'Using␣map:␣{map_name}')
30
31          # Retrieve the map size parameter (optional)
32          self.declare_parameter('map_size', None)
33          map_size_param = self.get_parameter('map_size').
                value
34
35          # Load map data from the specified package directory
36          package_dir = get_package_share_directory('
                Discovering␣_gazebo')
37          map_dir = os.path.join(package_dir, 'maps', f'{
                map_name}.csv')
38
39          try:
```

```
40              map_data = pd.read_csv(map_dir, header=None).
                    values
41          except FileNotFoundError:
42              self.get_logger().error(f'Map file {map_name}.
                    csv not found in {map_dir}')
43              raise
44
45          # Calculate free space size based on the map data
46          self.map_resolution = 0.25  # Each cell represents
                0.25m^2
47          if map_size_param is None:
48              self.total_free_space = np.count_nonzero(
                    map_data == 0) * self.map_resolution
49          else:
50              self.total_free_space = map_size_param
51
52      def map_callback(self, msg):
53          """
54          Callback function to process incoming map data and
                calculate exploration progress.
55          """
56          occupancy_data = np.array(msg.data)
57          cell_resolution = msg.info.resolution
58          explored_area = np.count_nonzero((occupancy_data <=
                self.free_threshold) & (occupancy_data >= 0)) * (
                cell_resolution2)
59
60          # Compute the exploration progress as a percentage
61          progress = explored_area / self.total_free_space
62          progress = min(progress, 1.0)  # Clamp to 100%
                maximum
```

```
63
64          # Publish the progress as a Float32 message
65          progress_msg = Float32()
66          progress_msg.data = progress
67          self.publisher_.publish(progress_msg)
68
69          # Log the progress percentage
70          self.get_logger().info(f'Exploration Progress: {
                progress * 100:.2f}%')
71
72
73  def main(args=None):
74      """
75      Main function to initialize and run the Watchtower node.
76      """
77      rclpy.init(args=args)
78      watchtower_node = Watchtower()
79      rclpy.spin(watchtower_node)
80
81      # Cleanup
82      watchtower_node.destroy_node()
83      rclpy.shutdown()
84
85
86  if __name__ == '__main__':
87      main()
```

Appendix D

Launch Description for Mapping and Visualization

Listing D.1: Modified Launch File for Cartographer and RViz Integration

```
import os
from ament_index_python.packages import
   get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node
from launch.substitutions import LaunchConfiguration
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import
   PythonLaunchDescriptionSource
from launch.substitutions import ThisLaunchFileDir


def generate_launch_description():
    # Configure parameters
    use_sim_time = LaunchConfiguration('use_sim_time',
       default='false')
```

```
15      cartographer_package_prefix =
            get_package_share_directory('scouting_cartographer')
16      cartographer_config_dir = LaunchConfiguration('
            cartographer_config_dir', default=os.path.join(
            cartographer_package_prefix, 'config'))
17      configuration_basename = LaunchConfiguration('
            configuration_basename',
18
19      resolution = LaunchConfiguration('resolution', default='
            0.05')
20      publish_period_sec = LaunchConfiguration('
            publish_period_sec', default='1.0')
21
22      rviz_config_dir = os.path.join(
            get_package_share_directory('scouting_cartographer'),
23                                      'rviz', '
                                            scout_cartographer.
                                            rviz')
24
25      return LaunchDescription([
26          # Launch Arguments
27          DeclareLaunchArgument(
28              'cartographer_config_dir',
29              default_value=cartographer_config_dir,
30              description='Path␣to␣the␣cartographer␣
                    configuration␣directory'),
31
32          DeclareLaunchArgument(
33              'configuration_basename',
34              default_value=configuration_basename,
```

```
35              description='Name␣of␣the␣Lua␣configuration␣file'
                    ),
36
37          DeclareLaunchArgument(
38              'use_sim_time',
39              default_value='false',
40              description='Use␣simulation␣clock␣if␣set␣to␣true
                    '),
41
42          # Cartographer Node
43          Node(
44              package='cartographer_ros',
45              executable='cartographer_node',
46              name='cartographer_node',
47              output='screen',
48              parameters=[{'use_sim_time': use_sim_time}],
49              arguments=['-configuration_directory',
                        cartographer_config_dir,
50                              '-configuration_basename',
                                    configuration_basename]),
51
52          DeclareLaunchArgument(
53              'resolution',
54              default_value=resolution,
55              description='Grid␣cell␣resolution␣for␣the␣
                    occupancy␣grid'),
56
57          DeclareLaunchArgument(
58              'publish_period_sec',
59              default_value=publish_period_sec,
```

```
60              description='Occupancy␣grid␣publishing␣interval␣
                    (seconds)'),
61
62          # Include Occupancy Grid Launch File
63          IncludeLaunchDescription(
64              PythonLaunchDescriptionSource([ThisLaunchFileDir
                    (), '/occupancy_grid.launch.py']),
65              launch_arguments={'use_sim_time': use_sim_time,
                    'resolution': resolution,
66                                'publish_period_sec':
                                      publish_period_sec}.items()
                                      ,
67          ),
68
69          # RViz2 Visualization Node
70          Node(
71              package='rviz2',
72              executable='rviz2',
73              name='rviz2',
74              arguments=['-d', rviz_config_dir],
75              parameters=[{'use_sim_time': use_sim_time}],
76              output='screen'),
77      ])
```

Appendix E

Manager Node for Navigation and Scouting

Listing E.1: Modified Manager Node for Autonomous Navigation and Scouting

```
from action_msgs.msg import GoalStatus
from geometry_msgs.msg import PoseStamped
from Discovering _interfaces.action import Navigate
from Discovering _interfaces.action import Scout
from A*_msgs.action import NavigateToPose

from std_msgs.msg import Float32
from visualization_msgs.msg import MarkerArray

import rclpy
import math
from rclpy.action import ActionClient
from rclpy.node import Node

from rclpy.node import Node
from rcl_interfaces.srv import GetParameters

```

```
18
19
20 class Manager(Node):
21     def __init__(self):
22         super().__init__('manager')
23         self._action_client_navigate = ActionClient(self,
               Navigate, 'navigate')
24         self._action_client_scout = ActionClient(self, Scout
               , 'scout')
25         self.navigation_client = NavigationClient()
26         self.watchtower_subscription = self.
               create_subscription(Float32, 'map_progress', self
               .watchtower_callback, 10)
27         self.trajectory_subscription = self.
               create_subscription(MarkerArray, '
               trajectory_node_list', self.trajectory_callback,
               10)
28         timer_period = 5  # seconds
29         self.timer = self.create_timer(timer_period, self.
               timer_callback)
30         self.map_progress = 0.01
31         self.map_complete = False
32         self.trajectory_distance = 0.0
33         self.trajectory_markers = MarkerArray()
34         self.start_time = self.get_clock().now()
35
36     def print_feedback(self):
37         try:
38             self.map_progress = "{:.2f}".format(self.
                   map_progress)  # Format to 2 decimals
```

```
39              self.trajectory_distance = self.
                    compute_distance_from_markers(self.
                    trajectory_markers)
40              self.trajectory_distance = "{:.2f}".format(self.
                    trajectory_distance)  # Format to 2 decimals
41              time_now = self.get_clock().now()
42              duration = str(int((time_now.nanoseconds - self.
                    start_time.nanoseconds) / (109)))
43              self.get_logger().info("Duration: %s s - Map: %s
                    - Distance: %s m " % (duration, self.
                    map_progress, self.trajectory_distance))
44          except:
45              pass
46
47      def timer_callback(self):
48          # Periodic feedback in the terminal
49          if not self.map_complete:
50              self.print_feedback()
51
52      def watchtower_callback(self, msg):
53          self.map_progress = msg.data * 100  # Convert to
                percentage
54
55      def trajectory_callback(self, msg):
56          self.trajectory_markers = msg.markers
57
58      def compute_distance_from_markers(self, markers):
59          trajectory_distance = 0.0
60          last_point = [0, 0]
61          try:
62              for marker in self.trajectory_markers:
```

```
63                  marker_points = marker.points
64                  for point in marker_points:
65                      point = [point.x, point.y]
66                      trajectory_distance += math.dist(
                            last_point, point)
67                      last_point = point
68              return trajectory_distance
69          except:
70              self.get_logger().warn("Trajectory data not yet
                received.")
71
72      def goal_response_callback_navigate(self, future):
73          goal_handle = future.result()
74          if not goal_handle.accepted:
75              self.get_logger().info('Navigation goal rejected
                .')
76              return
77
78          self.get_logger().info('Navigation goal accepted.')
79
80          self._get_result_future = goal_handle.
                get_result_async()
81          self._get_result_future.add_done_callback(self.
                get_result_callback_navigate)
82
83      def feedback_callback_navigate(self, feedback):
84          self.get_logger().info('Feedback received: {0}'.
                format(feedback.feedback.sequence))
85
86      def get_result_callback_navigate(self, future):
87          result = future.result().result
```

```
88          status = future.result().status
89          if status == GoalStatus.STATUS_SUCCEEDED:
90              self.map_complete = True
91              self.get_logger().info('MAP␣SUCCESSFULLY␣SCOUTED
                    .')
92              self.print_feedback()
93              # Return to home
94              self.navigation_client.send_goal()
95          else:
96              self.get_logger().info('Goal␣failed␣with␣status:
                    ␣{0}'.format(status))
97
98      def send_goal_navigate(self):
99          self.get_logger().info('Waiting␣for␣action␣server...
                ')
100         self._action_client_navigate.wait_for_server()
101
102         goal_msg = Navigate.Goal()
103         goal_msg.map_completed_thres = 0.9
104
105         self.get_logger().info('Sending␣navigate␣goal␣
                request...')
106         self.get_logger().info('Navigating␣until␣90%␣map␣
                completion.')
107
108         self._send_goal_future = self.
                _action_client_navigate.send_goal_async(
109             goal_msg,
110             feedback_callback=self.
                    feedback_callback_navigate)
111
```

```
112         self._send_goal_future.add_done_callback(self.
                goal_response_callback_navigate)
113 
114     def send_goal_scout(self):
115         self.get_logger().info('Waiting␣for␣action␣server...
                ')
116         self._action_client_scout.wait_for_server()
117 
118         goal_msg = Scout.Goal()
119         goal_msg.strategy = 1
120         goal_msg.map_completed_thres = 0.97
121 
122         self.get_logger().info('Sending␣scout␣goal␣request
                ...')
123         self.get_logger().info('Scouting␣until␣97%␣map␣
                completion.')
124 
125         self._send_goal_future = self._action_client_scout.
                send_goal_async(
126             goal_msg,
127             feedback_callback=self.feedback_callback_scout)
128 
129         self._send_goal_future.add_done_callback(self.
                goal_response_callback_scout)
130 
131 class NavigationClient(Node):
132     def __init__(self):
133         super().__init__('navigation_client')
134         self._action_client = ActionClient(self,
                NavigateToPose, 'navigate_to_pose')
135 
```

```
136     def goal_response_callback(self, future):
137         goal_handle = future.result()
138         if not goal_handle.accepted:
139             self.get_logger().info('Navigation to base
                rejected.')
140             return
141 
142         self.get_logger().info('Navigation to base accepted.
            ')
143 
144         self._get_result_future = goal_handle.
            get_result_async()
145         self._get_result_future.add_done_callback(self.
            get_result_callback)
146 
147     def get_result_callback(self, future):
148         result = future.result().result
149         status = future.result().status
150         if status == GoalStatus.STATUS_SUCCEEDED:
151             self.get_logger().info('Arrived at home position
                .')
152         else:
153             self.get_logger().info('Goal failed with status:
                {0}'.format(status))
154 
155     def send_goal(self):
156         self.get_logger().info('Waiting for action server...
            ')
157         self._action_client.wait_for_server()
158 
159         goal_msg = NavigateToPose.Goal()
```

```
160         goal_msg.pose.pose.orientation.w = 1.0  # Home
               position
161
162         self.get_logger().info('Returning to base...')
163
164         self._send_goal_future = self._action_client.
               send_goal_async(goal_msg)
165         self._send_goal_future.add_done_callback(self.
               goal_response_callback)
166
167 def main(args=None):
168     rclpy.init(args=args)
169
170     manager = Manager()
171
172     select = input('Select discovering algorithm:\n    1) Navigate\n    2) Scout\n')
173     if select == '1':
174         manager.send_goal_navigate()
175         rclpy.spin(manager)
176     elif select == '2':
177         manager.send_goal_scout()
178         rclpy.spin(manager)
179     else:
180         raise ValueError("Discovering algorithm not selected correctly.")
181
182 if __name__ == '__main__':
183     main()
```

Appendix F

Launch File for Navigating and Scouting Nodes

Listing F.1: Modified Launch File for Navigating and Scouting

```
import os

from ament_index_python.packages import
    get_package_share_directory
from launch import LaunchDescription
from launch.actions import OpaqueFunction,
    IncludeLaunchDescription, DeclareLaunchArgument
from launch.launch_description_sources import
    PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration,
    PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare
from launch_ros.actions import Node

TURTLEBOT3_MODEL = os.environ['TURTLEBOT3_MODEL']

def launch_setup(context, *args, kwargs):
```

```
14
15      map_name = LaunchConfiguration('map_name', default='map7
            ')
16      world_file_name = map_name.perform(context) + '.world.
            xml'
17      world = os.path.join(get_package_share_directory('
            Discovering␣_gazebo'), 'worlds', world_file_name)
18
19      pkg_gazebo_ros = get_package_share_directory('gazebo_ros
            ')
20      A*_file_dir = get_package_share_directory('
            turtlebot3_navigation2')
21      gazebo_launch_file_dir = os.path.join(
            get_package_share_directory('turtlebot3_gazebo'), '
            launch')
22      cartographer_launch_file_dir = os.path.join(
            get_package_share_directory('Discovering␣
            _cartographer'), 'launch')
23
24      use_sim_time = LaunchConfiguration('use_sim_time',
            default='true')
25      x_pose = LaunchConfiguration('x_pose', default='2.0')
26      y_pose = LaunchConfiguration('y_pose', default='3.0')
27
28      param_file_name = TURTLEBOT3_MODEL + '.yaml'
29
30      gzserver_cmd = IncludeLaunchDescription(
31          PythonLaunchDescriptionSource(
32              os.path.join(pkg_gazebo_ros, 'launch', 'gzserver
                    .launch.py')
33          ),
```

```
34          launch_arguments={'world': world}.items()
35      )
36
37      gzclient_cmd = IncludeLaunchDescription(
38          PythonLaunchDescriptionSource(
39              os.path.join(pkg_gazebo_ros, 'launch', 'gzclient
                    .launch.py')
40          )
41      )
42
43      robot_state_publisher_cmd = IncludeLaunchDescription(
44          PythonLaunchDescriptionSource(
45              os.path.join(gazebo_launch_file_dir, '
                    robot_state_publisher.launch.py')
46          ),
47          launch_arguments={'use_sim_time': use_sim_time}.
                items()
48      )
49
50      spawn_turtlebot_cmd = IncludeLaunchDescription(
51          PythonLaunchDescriptionSource(
52              os.path.join(gazebo_launch_file_dir, '
                    spawn_turtlebot3.launch.py')
53          ),
54          launch_arguments={
55              'x_pose': x_pose,
56              'y_pose': y_pose
57          }.items()
58      )
59
60      cartographer_cmd = IncludeLaunchDescription(
```

```
61          PythonLaunchDescriptionSource(
62              os.path.join(cartographer_launch_file_dir, '
                    cartographer.launch.py')
63          ),
64          launch_arguments={'use_sim_time': use_sim_time}.
                items(),
65      )
66
67      A*_cmd = IncludeLaunchDescription(
68          PythonLaunchDescriptionSource(
69              os.path.join(get_package_share_directory('A*
                    _bringup'), 'launch', 'bringup_launch.py')
70          ),
71          launch_arguments={
72              'map': os.path.join(A*_file_dir, 'map', 'map.
                    yaml'),
73              'use_sim_time': use_sim_time,
74              'params_file': os.path.join(A*_file_dir, 'param'
                    , param_file_name)}.items(),
75      )
76
77      navigating_cmd = Node(
78          package='Discovering␣_navigating',
79          executable='navigating_server',
80          name='navigating_server',
81          output='screen',
82      )
83
84      scouting_cmd = Node(
85          package='Discovering␣_scouting',
86          executable='scouting_server',
```

```
87          name='scouting_server',
88          output='screen',
89      )
90
91      watchtower_cmd = Node(
92          package='Discovering_map_utils',
93          executable='watchtower',
94          name='watchtower',
95          output='screen',
96          parameters=[{'map_name': map_name}],
97      )
98
99      return [
100         gzserver_cmd,
101         gzclient_cmd,
102         robot_state_publisher_cmd,
103         spawn_turtlebot_cmd,
104         cartographer_cmd,
105         A*_cmd,
106         navigating_cmd,
107         scouting_cmd,
108         watchtower_cmd,
109     ]
110
111
112 def generate_launch_description():
113     return LaunchDescription([
114         OpaqueFunction(function=launch_setup)
115     ])
```