GERMAN - MONGOLIAN INSTITUTE
FOR RESOURCES AND TECHNOLOGY

The present work was submitted to
the German-Mongolian Institute of Resources and Technology

DEVELOPMENT OF CONTROL SYSTEM OF AUTONOMOUS CLEANING ROBOT FOR LARGE AREA

Bachelor's Thesis

By

TELMEN Tamir

Study program: Mechatronic Engineering

Student ID: B2100517

1st Supervisor/Examiner: Ph. D. Odbileg .N

2nd Supervisor/Examiner: Ph. D. Youngsuk Kim

Ulaanbaatar/Nalaikh

2026

Statutory Declaration

Tamir, Telmen — B2100517

Last Name, First Name — Student ID Number

I hereby affirm in lieu of an oath that I provided the submitted bachelor thesis

DEVELOPMENT OF CONTROL SYSTEM OF AUTONOMOUS CLEANING ROBOT FOR LARGE INDOOR AREA

I did not use any sources other than those stated. In case that the work is additionally submitted on a data medium, I declare that the written and the electronic form are completely identical. The work was not submitted in the same or similar form to any examination authority.

Nalaikh, Ulaanbaatar,
5/28/2026

Place, Date

Signature

The present work was submitted to the

Faculty of Electrical and Mechanical Engineering

Acknowledgement

I would like to express my sincere gratitude to my supervisors, Ph.D. Odbileg N. and Ph.D. Youngsuk Kim, for their continuous guidance, technical advice, and constructive feedback throughout the development of this thesis. Their support and recommendations were highly valuable in shaping the direction of this work and improving the quality of the final result.

I would also like to extend my appreciation to my colleagues and fellow students for their encouragement, helpful discussions, and support during the development of this project.

Finally, I would like to thank my family for their patience, understanding, and continuous support throughout my academic studies. Their encouragement has been an important source of motivation during the completion of this thesis.

Abstract

This thesis presents the development and simulation-based validation of a control system for an autonomous cleaning robot designed for large indoor areas. The selected test environment is the main hall and corridor area of the GMIT first floor, which includes long corridors, curved boundaries, irregular wall geometry, and fixed internal obstacles representing a realistic and navigationally complex indoor space.

The system is structured as a two-stage simulation workflow implemented in CoppeliaSim EDU and MATLAB R2025b. In the first stage, a Pioneer P3DX differential-drive robot equipped with a 360-degree LiDAR sensor assembly performs boundary-following exploration, generating an occupancy grid map that is exported for external processing. In the second stage, MATLAB processes the raw map through noise removal, free-space extraction, obstacle inflation, and edge polyline extraction, then generates a complete cleaning mission combining boundary edge-cleaning paths, interior zig-zag coverage paths, safe transition routes, and a return-to-start path. The mission is executed in CoppeliaSim using a pure pursuit path tracking controller, with cleaning coverage tracked continuously throughout execution.

The system achieved a final cleaning coverage of 99.4 percent, exceeding the predefined success threshold of 95 percent. This result was obtained under ideal simulation conditions in which the map and execution environment are identical and localization error is absent. These conditions represent an upper bound on expected performance, and physical deployment would introduce additional factors including sensor noise, odometry drift, and environmental changes not captured in simulation. A proposed hardware deployment architecture for transferring the developed system to a physical Roborock-based platform is included as a foundation for future implementation and validation.

Keywords: autonomous cleaning robot, LiDAR, occupancy grid, coverage path planning, edge cleaning, zig-zag coverage, pure pursuit, CoppeliaSim, MATLAB

List of Figures

Figure 3.1 System Workflow Diagram

Figure 4.1 Printed Version of GMIT First Floor Plan

Figure 5.1 SketchUp 3D Reconstruction of the GMIT Main Hall and Corridors

Figure 5.2 GMIT Simulation Environment and Its Dimensions in CoppeliaSim

Figure 5.3 Pioneer P3DX Robot Model and Its Dimensions

Figure 5.4 LiDAR Sensor Assembly

Figure 6.1 Robot Exploration Track in CoppeliaSim — Boundary Following Result

Figure 6.2 Raw Occupancy Map Generated from Exploration Stage

Figure 6.3 Eroded Free Map After Noise Removal

Figure 6.4 Final Main Free Map After Region Extraction

Figure 6.5 Safe Navigation Map with Edge Polylines Overlay

Figure 6.6 Extracted Edge Polylines of GMIT Environment

Figure 6.7 Generated Edge-Cleaning Mission with Safe Transfer Paths

Figure 6.8 Generated Interior Zig-Zag Coverage Paths and Transfers

Figure 6.9 Final Combined Cleaning Mission Path

Figure 6.10 Robot Path Following Behavior Result

Figure 6.11 Edge-Cleaning Path Following Result

Figure 6.12 Interior Zig-Zag Path Following Behavior

Figure 6.13 Transition Movement Between Cleaning Path Sections

Figure 6.14 Return-to-Start Movement Result

Figure 6.15 Final Cleaning Coverage Visualization After Mission Completion

List of Tables

Table 5.1 Parameters Used for Noise Removal and Free Region Extraction

Table 5.2 Parameters Used for Safe Transition Map Generation

Table 5.3 Parameters Used for Edge-Cleaning Coverage Generation

Table 5.4 Parameters Used for Interior Zig-Zag Coverage Generation

Table 5.5 Parameters Used for Final Mission Assembly

List of Abbreviations

A* — A-Star Search Algorithm

AMR — Autonomous Mobile Robot

CPP — Coverage Path Planning

CSV — Comma-Separated Values

EDU — Educational (CoppeliaSim EDU Edition)

GMIT — German-Mongolian Institute for Resources and Technology

LiDAR — Light Detection and Ranging

MAT — MATLAB Binary Data File Format

MATLAB — Matrix Laboratory

OBJ — Wavefront Object File Format

P3DX — Pioneer 3-DX

PID — Proportional-Integral-Derivative

PSO — Particle Swarm Optimization

RDP — Ramer-Douglas-Peucker Algorithm

ROS — Robot Operating System

SLAM — Simultaneous Localization and Mapping

2D — Two-Dimensional

3D — Three-Dimensional

Table of Contents

Statutory Declaration **1**
Acknowledgement **2**
Abstract **3**
List of Figures **4**
List of Tables **5**
List of Abbreviations **6**
Table of Contents **7**
1. Introduction **10**
1.1 Background and Motivation 10
1.2 Problem Statement 10
1.3 Research Objectives 11
1.4 Scope and Limitations 11
2. Literature Review **12**
2.1 Autonomous Cleaning Robots and Indoor Navigation 12
2.2 LiDAR-Based Mapping 13
2.3 Occupancy Grid Representation 14
2.4 Coverage Path Planning 15
2.4.1 Boundary and Edge Cleaning 16
2.4.2 Boustrophedon Interior Coverage 17
2.4.3 Combined Coverage Strategy 17
2.5 Pure Pursuit Path Tracking 18
2.6 Differential-Drive Motion Control 19
2.7 ROS-Based Robot Control for Hardware Deployment 20
3. System Overview **21**
3.1 Overall System Architecture 21
3.2 System Workflow Description 21
3.3 Proposed Hardware Deployment Architecture 23
3.4 Software Tools and Environment 24
3.4.1 SketchUp 2017 24
3.4.2 CoppeliaSim EDU 24
3.4.3 MATLAB R2025b 25
4. Materials and Methods **25**
4.1 Indoor Environment Selection 25
4.1.1 Area Selection Criteria 25
4.1.2 Environment Reconstruction Approach 26
4.2 Robot and Sensor Configuration 26
4.2.1 Robot Model 26
4.2.2 LiDAR Sensor Configuration 27
4.3 Navigation Strategy 27

4.3.1 Exploration Strategy Selection..... 27
4.3.2 Exploration Completion Condition..... 28
4.4 Map Processing Strategy..... 29
4.4.1 Occupancy Grid..... 29
4.4.2 MATLAB-Based Mapping..... 29
4.5 Cleaning Method Selection..... 30
4.6 Robot Motion and Path Tracking..... 31
4.6.1 Tracking Method..... 31
4.6.2 Differential-Drive Motion Control..... 32
4.6.3 Speed Control..... 32
4.7 Evaluation Method..... 32
4.7.1 Evaluation Criteria..... 32
4.7.2 Success Metrics and Thresholds..... 33
4.7.3 Coverage Assessment Method..... 33
5. Implementation..... 34
5.1 Simulation Environment Reconstruction..... 34
5.1.1 SketchUp Layout Reconstruction Process..... 34
5.1.2 CoppeliaSim Environment Import and Scaling..... 35
5.2 Robot and Sensor Implementation..... 36
5.2.1 Pioneer P3DX Model..... 36
5.2.2 LiDAR Sensor Assembly..... 37
5.2.3 Sensing Script and LiDAR Data Pipeline..... 38
5.3 Stage 1: Exploration and Map Export..... 39
5.4 MATLAB Map Processing Pipeline..... 40
5.5 Cleaning Path Generation..... 43
5.6 Stage 2: Mission Execution..... 46
6. Results..... 49
6.1 Stage 1: Exploration Results..... 49
6.1.1 Boundary Following Behavior..... 49
6.1.2 Generated Occupancy Map Quality..... 50
6.2 MATLAB Map Processing Results..... 51
6.2.1 Noise Removal and Free Region Extraction Result..... 51
6.2.2 Safe Navigation Map Result..... 53
6.2.3 Edge Extraction Result..... 54
6.3 Cleaning Path Planning Results..... 55
6.3.1 Edge Cleaning Paths..... 55
6.3.2 Interior Zig-Zag Coverage Paths..... 56
6.3.3 Final Combined Cleaning Mission..... 57
6.4 Stage 2 Mission Execution Results..... 58
6.4.1 Path Following Performance..... 58
6.4.2 Transition Movement Results..... 60

6.4.3 Return to Start Result..... 60
6.4.4 Final Coverage Percentage..... 61
7. Discussion..... 62
7.1 Interpretation of Simulation Results..... 62
7.2 Navigation and Path Tracking Performance..... 62
7.3 Map Processing and Path Planning Analysis..... 63
7.4 Coverage Performance Analysis..... 64
7.5 Proposed Hardware Deployment Discussion..... 65
7.6 System Limitations..... 66
7.7 Comparison with Related Work..... 67
8. Conclusion..... 68
9. References..... 69

1. Introduction

1.1 Background and Motivation

Autonomous mobile robots have become increasingly important in a wide range of service and industrial applications. Among these, autonomous cleaning robots represent one of the most commercially relevant and practically demanded categories. The growing need to maintain cleanliness in large indoor environments such as airports, shopping centers, hospitals, and educational institutions has created significant interest in robotic systems that can perform cleaning tasks independently, reliably, and efficiently without continuous human supervision [1].

Commercially available robotic vacuum cleaners and floor cleaning robots, such as those produced by iRobot and Roborock, have demonstrated the practical feasibility of autonomous cleaning in residential and small office environments . However, these systems are primarily designed for small to medium-sized spaces with relatively simple layouts. When applied to large and irregularly shaped indoor environments, such systems often suffer from incomplete coverage, inefficient path planning, and an inability to navigate complex boundary structures that include curved walls and fixed internal obstacles [2].

Meeting these requirements demands a more structured approach to autonomous navigation, environment mapping, and cleaning path planning than what current low-cost commercial systems provide.

This thesis is motivated by the need for a control system that can address these challenges. By combining LiDAR-based exploration and occupancy mapping, MATLAB-based map processing and path planning, and simulation-validated path tracking, the developed system aims to demonstrate a practical and deployable approach to autonomous cleaning in large indoor environments.

1.2 Problem Statement

Existing autonomous cleaning robots designed for large indoor environments typically rely on complex simultaneous localization and mapping systems or pre-built environment maps, both of which require significant computational resources and specialized hardware. For low-cost or repurposed cleaning platforms, such approaches are often impractical. Furthermore, most coverage path planning methods assume simple rectangular or convex environments and do not adequately address spaces with curved boundaries, irregular shapes, and internal fixed obstacles.

The specific problem addressed in this thesis is the development of a control system for an autonomous cleaning robot that can explore a large and irregularly shaped indoor environment, generate a usable map from LiDAR sensor data using a boundary following strategy, plan a complete cleaning path that covers both boundary regions and interior floor areas, and execute the planned mission reliably using a path tracking controller. The selected target environment is the main hall and corridor area of the GMIT first floor, which represents a realistic example of a large indoor space with the navigational challenges described above.

The central research question is: Can a two-stage LiDAR-based control system achieve complete and reliable cleaning coverage in a large, irregularly shaped indoor environment without requiring a continuous SLAM system during execution?

1.3 Research Objectives

The main objectives of this thesis are as follows:

- To reconstruct the selected GMIT indoor environment in a simulation platform suitable for robot navigation and sensor testing.
- To develop an autonomous boundary exploration system that generates an occupancy grid map of the environment using LiDAR sensor data.
- To implement a MATLAB-based map processing pipeline that cleans the raw occupancy map, extracts edge polylines, and generates a safe navigation map.
- To design a combined cleaning path planning strategy that includes boundary edge-cleaning paths and interior zig-zag coverage paths connected by safe transition routes.
- To implement and evaluate a pure pursuit path tracking controller for mission execution, with cleaning coverage tracked as the primary performance metric.
- To propose a hardware deployment architecture describing how the developed control concept can be transferred to a physical cleaning robot platform.

1.4 Scope and Limitations

This thesis focuses on the simulation-based development and validation of the autonomous cleaning robot control system. The primary simulation environment is the main hall and corridor area of the GMIT first floor, which was selected for its navigational complexity and practical relevance. Classrooms, the canteen, and other areas containing many movable obstacles such as chairs and tables were excluded

from the simulation environment, as these areas are not reliably accessible using LiDAR-based navigation alone.

The entire system was developed and evaluated within the CoppeliaSim EDU simulation environment. All robot control logic, sensor processing, occupancy mapping, and coverage tracking were implemented as Lua scripts running inside the simulation. Map processing and path planning were performed externally using MATLAB R2025b scripts. The simulation uses the Pioneer P3DX robot model as the mobile platform and a 360-degree LiDAR sensor assembly for environmental perception.

Physical hardware integration and real-world validation are outside the main scope of this thesis. A proposed hardware deployment architecture is described in Section 3.3 to outline how the developed control concept could be transferred to a physical platform. Hardware assembly and validation are planned as future work and will be completed separately from the simulation-based results presented in this thesis.

Additional limitations of the current system include the assumption of a static environment with no dynamic obstacles, the absence of battery or power modeling, and the use of a rectangular robot body which has a higher probability of contact with walls during turning in narrow areas compared with round-shaped commercial cleaning robots.

2. Literature Review

2.1 Autonomous Cleaning Robots and Indoor Navigation

Autonomous mobile robots are robots that can move and perform tasks without direct human control. In cleaning applications, these robots must be able to sense the environment, avoid obstacles, plan movement, and cover the required floor area. Therefore, autonomous cleaning robots usually depend on several important technologies, including localization, mapping, coverage path planning, obstacle avoidance, and motion control [1].

Indoor cleaning is different from simple robot navigation because the robot does not only need to move from one point to another. It must clean as much of the reachable floor area as possible. This is especially important in large indoor environments such as hospitals, airports, shopping centers, schools, and office buildings. In these environments, random movement is not efficient because it can repeat some areas while leaving other areas uncleaned [1].

Earlier indoor mobile robot systems showed that simple sensor-based navigation can support autonomous movement. For example, Paneru et al. developed an indoor mobile robot using ultrasonic sensors and odometry for obstacle detection and navigation [2]. Their work shows that even simple sensor systems can be useful when they are combined with suitable control logic.

Modern commercial cleaning robots show that autonomous cleaning technology is already practical in real indoor environments. Commercial robots such as iRobot and Roborock models commonly use sensors such as LiDAR, cameras, ultrasonic sensors, and wheel encoders to detect walls, obstacles, and open floor areas. Many modern robots can create a two-dimensional floor map and clean the room using a systematic path instead of moving randomly.

For example, Roborock robots use LiDAR-based navigation to scan the environment and build a floor map. Some advanced models also use camera-based or AI-based obstacle detection to recognize small objects and avoid them during cleaning. These examples show that mapping, path planning, and obstacle avoidance are already important technologies in real consumer cleaning robots. However, most home cleaning robots are designed mainly for houses and small or medium indoor spaces. Large and irregular indoor areas still require more structured planning and control.

The system developed in this thesis follows the same general idea. A LiDAR sensor is used to collect distance information from the environment. This information is used to generate an occupancy grid map. Then, cleaning paths are planned from the map, and the robot follows the planned path using a motion controller. The following sections review the main technologies used in this work, including LiDAR-based mapping, occupancy grid representation, coverage path planning, pure pursuit tracking, differential-drive control, and ROS-based hardware deployment.

2.2 LiDAR-Based Mapping

LiDAR stands for Light Detection and Ranging. It is a sensing method that measures distance by sending laser signals and detecting the reflected signal from surrounding objects. In mobile robotics, LiDAR is widely used because it can measure the distance between the robot and nearby walls or obstacles with good accuracy.

A 2D LiDAR sensor scans the environment on a horizontal plane. As the robot moves, the LiDAR continuously detects walls, obstacles, and open spaces around the robot. These distance measurements can be used to build a map of the environment. For

indoor cleaning robots, this is useful because most important objects, such as walls, columns, doors, and fixed obstacles, can be represented in a two-dimensional map.

Several studies have shown that LiDAR sensors are suitable for indoor mapping. Brindza et al. developed and tested a LiDAR-based mobile mapping system for indoor environments [3]. Their results showed that 2D laser range data can be used to create accurate maps of rooms and corridors. This supports the use of LiDAR for autonomous cleaning robots because the robot needs a reliable map before it can plan a complete cleaning path.

One important challenge in mapping is that indoor environments can change over time. People, chairs, temporary objects, and other movable items can appear or disappear between cleaning sessions. Stefanini et al. studied a 2D LiDAR-based map updating method for long-term operation in dynamic environments [4]. Their work shows the importance of separating fixed structural objects, such as walls and doorframes, from temporary objects, such as people or movable furniture [4].

The most common method for LiDAR-based map construction is simultaneous localization and mapping, or SLAM. In SLAM, the robot estimates its own position while also building or updating the map of the environment. This is important because the robot needs to know both where it is and what the environment looks like. In this thesis, the mapping concept is based on 2D LiDAR data and occupancy grid representation. The resulting map is used later for map processing and cleaning path generation.

2.3 Occupancy Grid Representation

An occupancy grid is a common map format used in mobile robotics. In this method, the environment is divided into small square cells. Each cell represents a small part of the real environment. A cell can be marked as free, occupied, or unknown depending on the sensor data [5].

Elfes introduced the occupancy grid as a probabilistic map representation [5]. In this method, each grid cell stores the probability that the corresponding area is occupied by an obstacle. When the robot receives new sensor measurements, the values of the grid cells are updated. If a LiDAR ray passes through a cell, that cell is more likely to be free. If the ray ends at a cell, that cell is more likely to contain an obstacle [5].

Thrun, Burgard, and Fox explain occupancy grid mapping in the probabilistic robotics framework [6]. They describe how sensor measurements can be used to update grid

cells and how the map can be built step by step as the robot moves through the environment [6]. This approach is widely used because it is simple, effective, and suitable for indoor robot navigation.

Occupancy grids are especially useful for cleaning robots because they are easy to process for path planning. Free cells show where the robot can move, occupied cells show where walls or obstacles are located, and unknown cells show areas that have not been observed yet. After the map is created, it can be processed to remove noise, inflate obstacles, and generate safe navigation paths.

MATLAB also provides useful tools for working with occupancy maps. The Robotics System Toolbox includes map objects such as `occupancyMap` and `binaryOccupancyMap`, which support map storage, ray insertion, and obstacle inflation [7]. Obstacle inflation is important because the robot has a physical size. By expanding the obstacle area in the map, the planner can generate paths that keep the robot away from walls and obstacles [7].

In this thesis, the occupancy grid is used as the main map representation because it matches the 2D LiDAR data and is suitable for MATLAB-based processing. The map is first generated in CoppeliaSim from LiDAR measurements and then exported to MATLAB for cleaning, free-space extraction, edge extraction, and path planning.

2.4 Coverage Path Planning

Coverage path planning is the process of creating a path that allows a robot to pass over all reachable areas of a target region while avoiding obstacles [8]. This is very important for cleaning robots because the goal is not only to reach a destination, but also to clean the full floor area.

The main goals of coverage path planning are high coverage, low repeated movement, and safe navigation. A good cleaning path should cover most of the cleanable floor area. It should also reduce unnecessary overlap because repeated cleaning of the same area wastes time and energy [8].

Galceran and Carreras reviewed many coverage path planning methods used in robotics [8]. They grouped these methods into several types, including exact cellular decomposition, approximate cellular decomposition, and grid-based methods [8]. In

exact cellular decomposition, the free space is divided into smaller cells that can be covered systematically. In grid-based methods, the environment is divided into grid cells, and the robot plans coverage based on those cells.

A common coverage pattern for cleaning robots is the back-and-forth sweeping pattern. This is often called the boustrophedon pattern. It is useful because it allows the robot to clean large open areas in an organized way. The robot moves along one row, turns at the end, and then moves back along the next row in the opposite direction [8].

Jayalakshmi et al. also reviewed recent coverage path planning methods, including methods for dynamic environments [9]. Their review shows that practical cleaning robots often use a combination of boundary cleaning and systematic interior coverage [9]. This type of hybrid method is useful because wall areas and open floor areas require different cleaning behavior.

In this thesis, a combined coverage path planning strategy is used. First, edge-cleaning paths are generated along the walls and fixed obstacles. Then, zig-zag coverage paths are generated for the interior free space. This combination is suitable for the selected GMIT environment because the area contains both irregular boundaries and large open floor regions.

2.4.1 Boundary and Edge Cleaning

Boundary cleaning is the process of cleaning near walls, corners, and fixed obstacle edges. This part is important because normal interior zig-zag paths may not clean close enough to walls or irregular boundaries. If the robot only follows a zig-zag pattern, some areas near the edges may remain uncleaned.

Wall following is one method that can be used for boundary cleaning. In wall following, the robot uses distance sensors to keep a desired distance from the wall. The robot continuously adjusts its movement based on the measured distance. If the robot is too close to the wall, it turns away. If it is too far from the wall, it turns closer.

Adriansyah et al. studied wall-following control using a PID controller and particle swarm optimization for parameter tuning [10]. Their work showed that closed-loop control can improve wall-following performance, especially near corners and irregular wall surfaces [10]. This is useful for cleaning robots because they need to keep a stable distance from walls during edge cleaning.

Boundary cleaning is also important in practical cleaning systems because dirt often collects near walls and corners. Galceran and Carreras explain that boundary coverage is commonly used together with interior coverage in cleaning robot applications [8]. Cleaning the boundary first can help define the cleaning region and improve the overall cleaning result.

2.4.2 Boustrophedon Interior Coverage

The boustrophedon pattern is a back-and-forth coverage method. The word refers to a movement pattern similar to how an ox moves while plowing a field. In robotics, this means the robot moves along parallel lanes. After reaching the end of one lane, it turns and follows the next lane in the opposite direction.

This method is useful for cleaning large open floor areas because it is simple, systematic, and easy to generate from a map. The distance between two lanes can be selected based on the robot's cleaning width. If the lane spacing is chosen correctly, the robot can cover the floor area without leaving large gaps.

Galceran and Carreras describe boustrophedon decomposition as one of the common methods for complete coverage path planning [8]. The method divides the free space into regions that can be covered using back-and-forth motion. It is especially useful for environments where systematic and complete floor coverage is required [8].

For indoor cleaning applications, the boustrophedon pattern is practical because many indoor spaces contain long corridors, halls, or rectangular floor areas. It also reduces random movement and makes the robot path easier to analyze. Compared with random cleaning, zig-zag coverage is more predictable and can achieve better coverage efficiency.

2.4.3 Combined Coverage Strategy

A combined coverage strategy uses more than one cleaning pattern to cover different parts of the environment. This is useful because one method is not always suitable for all areas. For example, edge cleaning is better for walls and obstacle boundaries, while zig-zag coverage is better for wide open areas.

Galceran and Carreras explain that many practical cleaning systems use boundary coverage together with interior coverage [8]. In this method, the robot first cleans near the boundary and then covers the remaining interior space using a systematic pattern. This improves both completeness and efficiency [8].

Jayalakshmi et al. also state that hybrid coverage strategies are useful for real cleaning environments [9]. These methods can combine boundary following, grid-based planning, and systematic coverage to handle different types of floor layouts [9]. This is important for environments that include irregular walls, curved boundaries, columns, and internal obstacles.

The GMIT first-floor main hall and corridor area has both boundary regions and open interior floor regions. Therefore, a combined cleaning strategy is more suitable than using only one cleaning method. Edge cleaning is used for walls and fixed obstacle boundaries, while zig-zag coverage is used for the main reachable floor area.

In this thesis, the final cleaning mission is created by combining edge-cleaning paths, zig-zag coverage paths, safe transition paths, and a return-to-start path. This creates one complete waypoint-based mission that can be imported into CoppeliaSim and followed by the robot.

2.5 Pure Pursuit Path Tracking

After the cleaning path is planned, the robot needs a tracking controller to follow the path. A path is usually represented as a set of waypoints. The controller calculates the robot motion commands needed to move from one waypoint to the next while staying close to the planned path.

Pure pursuit is a common path tracking method for mobile robots. It works by selecting a target point on the path in front of the robot. This target point is called the lookahead point. The robot then calculates the turning motion needed to move toward this point. As the robot moves, the lookahead point also moves forward along the path.

The main advantage of pure pursuit is that it is simple and stable for many mobile robot applications. It does not require a very complex model of the robot. It can also follow both straight and curved paths. This makes it suitable for cleaning missions that include straight zig-zag lanes, curved boundary paths, and transition movements.

Morales et al. studied the pure pursuit method for path tracking of nonholonomic mobile robots using 2D laser scanner information [11]. Their work showed that pure pursuit can be used for reactive path tracking in indoor environments [11]. This supports its use in mobile robot navigation and path following tasks.

The lookahead distance is an important parameter in pure pursuit control. A short lookahead distance can make the robot follow the path closely, but it may also cause unstable or oscillating motion. A longer lookahead distance can make the movement smoother, but the robot may cut corners or move away from the desired path. Therefore, the lookahead distance must be selected carefully.

2.6 Differential-Drive Motion Control

Differential-drive robots are widely used in indoor mobile robot applications. A differential-drive robot usually has two main driving wheels placed on the left and right sides of the robot. Each wheel can be controlled independently. The robot may also have one or more passive caster wheels for balance.

The movement of a differential-drive robot depends on the speeds of the left and right wheels. If both wheels move at the same speed, the robot moves forward. If one wheel moves faster than the other, the robot turns. If the wheels move in opposite directions, the robot can rotate in place. This makes differential drive suitable for indoor environments where the robot needs to turn in narrow spaces.

Thrun, Burgard, and Fox describe the motion model of differential-drive robots in the probabilistic robotics framework [6]. Their work explains how wheel velocities can be used to estimate the robot's movement and position over time [6]. However, odometry can include errors because of wheel slip, uneven floor surfaces, and sensor noise. Therefore, sensor-based correction is often needed in real robot systems [6].

The differential-drive structure is useful for cleaning robots because it is mechanically simple and easy to control. It can also perform sharp turns, which are needed at the ends of zig-zag cleaning lanes. This is especially useful for boustrophedon coverage, where the robot must turn around and continue along the next lane.

In this thesis, the Pioneer P3DX robot model is used as the simulation platform. This robot has a differential-drive structure. The pure pursuit controller provides desired linear and angular motion, and this motion is converted into left and right wheel commands. This allows the robot to follow the planned cleaning paths in the simulation environment.

2.7 ROS-Based Robot Control for Hardware Deployment

The Robot Operating System, or ROS, is a widely used software framework for robotics. It provides tools for communication between robot software modules, sensor data handling, mapping, navigation, and motion control. ROS is commonly used in research and development because it allows different robot components to work together in a modular way.

ROS is useful for mobile robots because it supports many common robotics functions. These include LiDAR data processing, SLAM, occupancy grid mapping, path planning, and differential-drive control. Many of these functions are available as existing ROS packages, which can reduce the amount of software that must be developed from the beginning.

Roy et al. developed a ROS-SLAM-based autonomous vacuum cleaner system [12]. Their system used a 2D LiDAR sensor, SLAM-based mapping, path planning, and differential-drive control [12]. This is similar to the general structure used in this thesis, where the system is divided into sensing, mapping, planning, and control stages.

Although the main implementation of this thesis is simulation-based, the system is designed with future hardware deployment in mind. The proposed hardware direction includes using a repurposed cleaning robot platform with a Raspberry Pi controller and a 360-degree LiDAR sensor. In a future physical implementation, ROS could be used to connect the LiDAR, motor driver, mapping module, path planner, and motion controller.

CoppeliaSim also supports robot simulation and can be connected with ROS through its ROS interface [13]. This makes it useful for testing robot behavior before moving to physical hardware. MATLAB also provides useful tools for map processing, image processing, path planning, and robot control development [14], [15]. Therefore, the simulation workflow used in this thesis can be considered a foundation for future ROS-based physical implementation.

Most related implementations either rely on a continuously running SLAM system during mission execution, assume a pre-built map, or evaluate coverage only in simple rectangular environments. This study addresses the gap by combining offline LiDAR-based mapping, external MATLAB path planning, and simulation-validated execution in a geometrically complex real building layout without requiring SLAM during cleaning.

3. System Overview

3.1 Overall System Architecture

The system developed in this study consists of three main parts: a 3D environment model reconstructed using SketchUp, map processing and path planning executed in MATLAB, and a simulation environment built in CoppeliaSim. The overall architecture follows a sense-plan-act structure, in which the robot first explores and maps the environment and cleaning paths are planned externally, and finally the robot executes the planned mission.

The system was designed around a two-stage simulation framework. In the first stage, the robot autonomously explores the indoor environment and generates an occupancy grid map using LiDAR sensor data. This map is then exported from CoppeliaSim as a CSV file and passed to MATLAB for further processing. In the second stage, the processed map and planned cleaning paths are imported back into CoppeliaSim, where the robot executes the full cleaning mission by following the pre-planned waypoints. This separation between exploration, planning, and execution allowed each component to be developed and tested independently. This separation also prevented overloading that would otherwise occur if exploration, planning, and execution were performed within a single continuous simulation run. A proposed physical hardware architecture is also introduced separately in Section 3.3 to describe the possible transfer of the simulation workflow to a repurposed Roborock platform.

3.2 System Workflow Description

The complete system workflow consists of six stages as described below:

Stage 1 Environment Reconstruction: The first-floor main hall and corridor area of the GMIT building was selected as the target environment. A printed copy of the building floor plan was used as a reference to reconstruct the layout in SketchUp. The reconstructed 3D model was then exported as an OBJ file and imported into CoppeliaSim to form the simulation environment.

Stage 2 Autonomous Exploration and Mapping: In the first CoppeliaSim simulation, the Pioneer P3DX robot model equipped with a 360-degree LiDAR sensor assembly performed autonomous boundary exploration of the environment. Using a right-wall following navigation strategy, the robot explored through the whole map more safely

and simultaneously generated an occupancy grid map. Upon completing the exploration, the occupancy map and its metadata were exported as CSV files for external processing

Stage 3 MATLAB Map Processing: The exported occupancy map was imported into MATLAB R2025b. A series of processing steps were applied to clean the raw map, extract the main reachable free region, generate a safe navigation map, and extract exact edge polylines from the environment boundaries. The processed map outputs were saved as MAT and CSV files for use in the path planning stage.

Stage 4 Cleaning Path Planning: Using the processed map data, MATLAB generated edge cleaning paths along the outer boundary and internal obstacle boundaries, zig-zag coverage paths for the interior floor area. Both completed missions were exported as a MAT file containing ordered waypoint coordinates and segment classification information.

Stage 5 Final Mission Planning: Both MAT files are imported into the final mission MATLAB generator, which combines the two paths into a complete mission, including transition paths and a return-to-start path.

Stage 6 Mission Execution: The planned mission CSV file was imported into CoppeliaSim for the second simulation stage. The robot executed the full cleaning mission by following the pre-planned waypoints using a pure pursuit-based path tracking controller. Cleaning coverage was tracked continuously throughout the execution, and the final coverage percentage was recorded as the main performance metric.

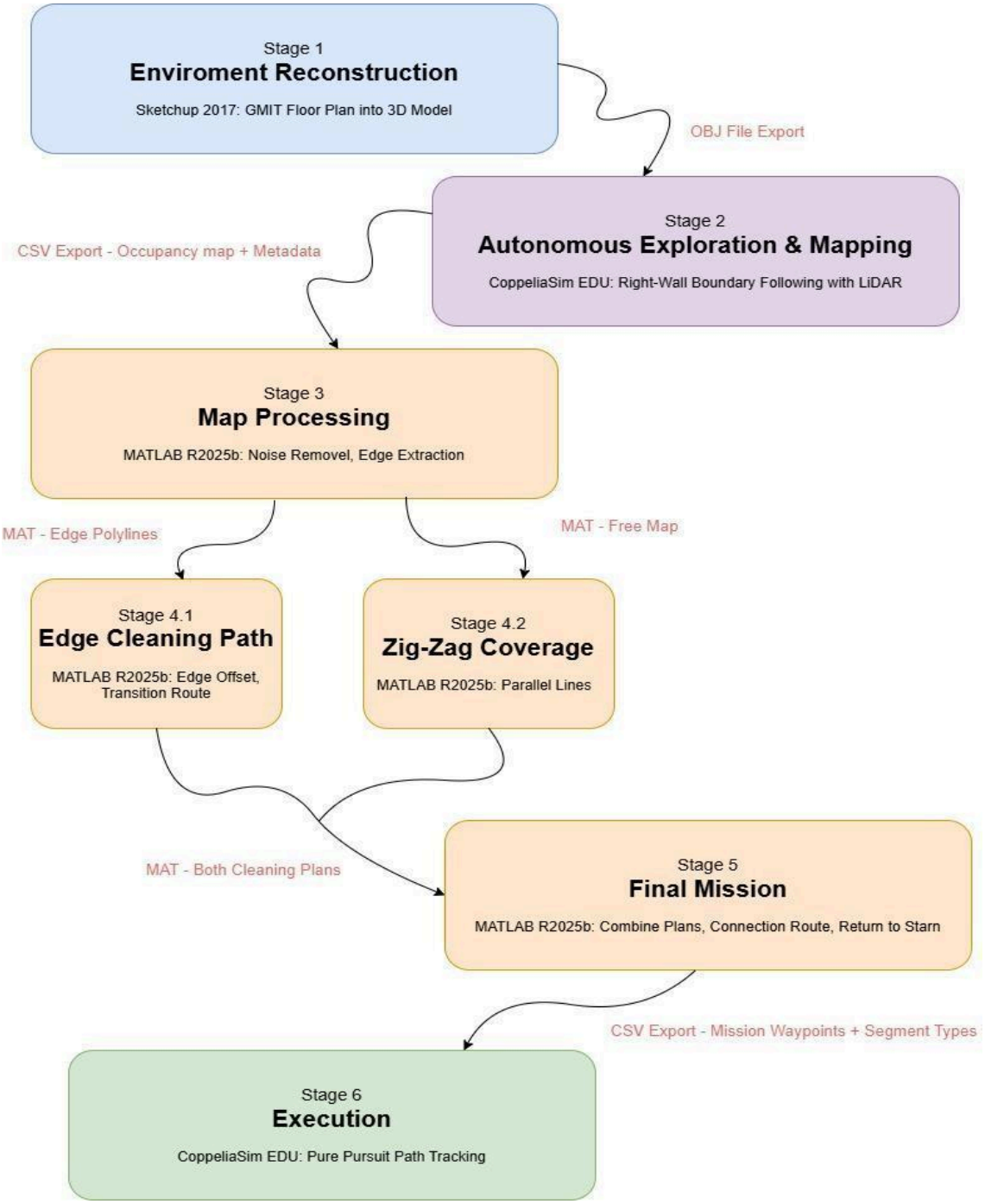


Figure 3.1 System Workflow Diagram

3.3 Proposed Hardware Deployment Architecture

The proposed physical platform is a repurposed Roborock S-series cleaning robot. This platform was selected because it includes a factory-installed rotating 360-degree LiDAR assembly, a differential-drive motor system with integrated encoders, a rechargeable battery, and a compact chassis suitable for indoor corridor navigation. The existing LiDAR, drive motors, and battery are retained from the original platform.

The original control board is replaced by a Raspberry Pi 3B single-board computer, which serves as the main controller for the autonomous navigation system."

"The Raspberry Pi 3B receives LiDAR point data through a serial or USB interface and publishes motor velocity commands to the motor driver board. In the proposed architecture, the boundary exploration and occupancy mapping software runs on the Raspberry Pi 3B during the initial environment scan. The occupancy map is then transferred to an external computer where the MATLAB map processing and path planning pipeline is executed offline. The resulting mission CSV file is transferred back to the Raspberry Pi 3B for mission execution. The pure pursuit controller and coverage tracking functions, currently implemented as Lua scripts in CoppeliaSim, would be re-implemented as lightweight Python or C++ programs running on the Raspberry Pi 3B."

"The computational requirements of the execution stage are modest. The pure pursuit controller performs vector arithmetic and array lookups at each time step, and the coverage tracking function updates a binary grid at each step. Both operations are well within the processing capability of the Raspberry Pi 3B at the required update frequency of approximately 10 Hz. The exploration stage is more demanding due to continuous occupancy grid updates, and the map update interval may need to be increased from the current 0.25 seconds to 0.5 seconds to maintain stable operation on the embedded hardware. Full hardware integration and performance validation are planned as the next phase of this work.

3.4 Software Tools and Environment

3.4.1 SketchUp 2017

SketchUp 2017 was used to reconstruct the GMIT indoor environment from the printed building floor plan. The software was selected because it provided sufficient 2D drawing and 3D modeling tools for the purposes of this study, including line drawing, edge offsetting, scaling using the Measure Tape function, and 3D extrusion. The reconstructed model was exported in OBJ format for import into CoppeliaSim.

3.4.2 CoppeliaSim EDU

CoppeliaSim EDU is a robot simulation platform developed by Coppelia Robotics [13]. It provides a physics-based simulation environment that supports various robot models, sensors, and scripting capabilities. In this study, CoppeliaSim EDU was used for both

simulation stages. All robot control logic, LiDAR sensing, occupancy grid generation, mission execution, and coverage tracking were implemented using Lua scripts running inside the CoppeliaSim environment. The Pioneer P3DX robot model and the fastHokuyo LiDAR model were both sourced from the CoppeliaSim built-in model library.

3.4.3 MATLAB R2025b

MATLAB R2025b was used for all map processing and path planning operations that were performed outside of the simulation environment [14], [15]. MATLAB was selected for this stage because it provides strong built-in support for matrix operations, image processing, and geometric computation, which were all required for occupancy map processing, free region extraction, edge polyline generation, and cleaning path planning. All processing and planning scripts were implemented as MATLAB script files and executed sequentially to produce the final mission output.

4. Materials and Methods

4.1 Indoor Environment Selection

4.1.1 Area Selection Criteria

For the simulation environment, only the main hall and corridor areas of GMIT first floor were selected for this study. Other areas, such as classrooms and the canteen, were excluded because they contain many obstacles, including chairs and tables. These environments are difficult to clean using only LiDAR-based navigation, since the robot cannot reliably access narrow spaces between furniture legs. Even with additional sensors or a more advanced system design, cleaning such areas would still be limited by the physical dimensions of the robot. Therefore, the main hall and corridor layout was considered a more suitable and realistic environment for evaluating the proposed cleaning robot system. Despite this focused scope, the selected area still presented sufficient navigational complexity, including curved boundaries, irregular shapes, and internal fixed obstacles, to properly evaluate the proposed navigation and cleaning methods.

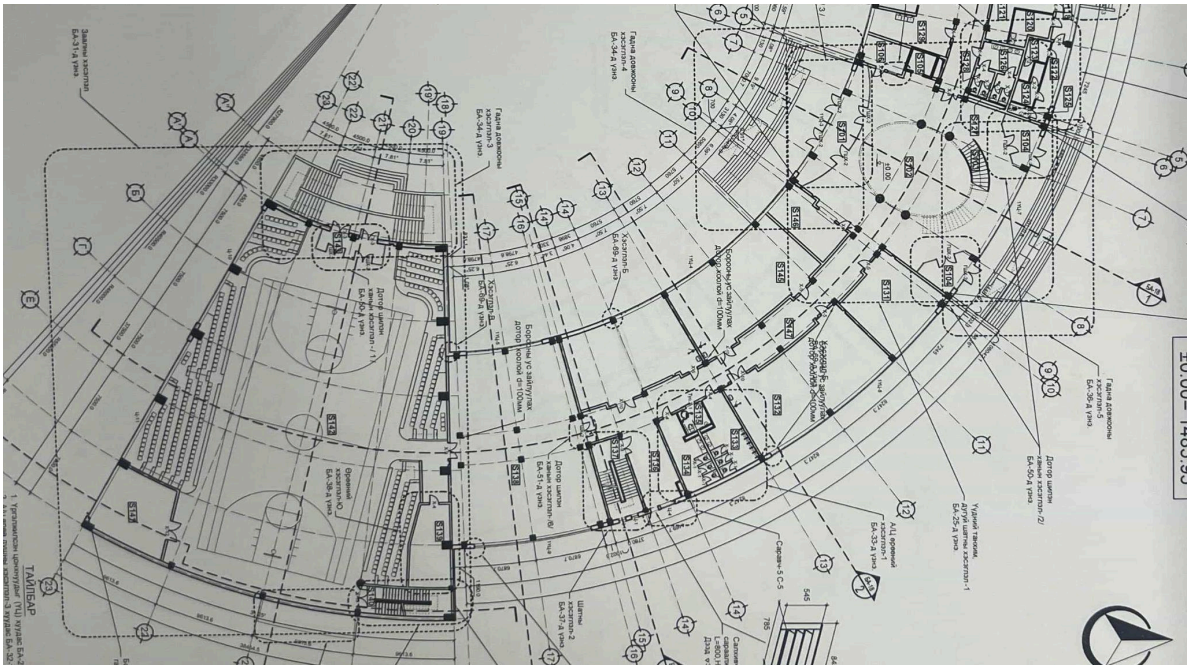


Figure 4.1 Printed Version of GMIT First Floor Plan

4.1.2 Environment Reconstruction Approach

The selected GMIT floor plan area was reconstructed in SketchUp, since this software was more suitable for drawing and modeling the environment than creating it directly in CoppeliaSim from a printed image alone. Complex and irregularly shaped environment reconstruction would have been more difficult and time consuming. SketchUp provided more suitable 2D drawing and 3D modeling tools that allowed the floor plan geometry to be traced and scaled more accurately and efficiently.

For a more stable and representative simulation environment, only obstacles that are permanently fixed in place and cannot be moved during normal daily use of the building were included. Temporary or frequently repositioned objects were excluded because including them would not accurately represent the actual cleaning environment, since their positions would differ between cleaning sessions. The reconstructed model was then imported into CoppeliaSim to form the final simulation environment used in this study.

4.2 Robot and Sensor Configuration

4.2.1 Robot Model

In this study, the Pioneer P3DX model available in the CoppeliaSim library was used as the mobile robot platform. It uses a differential-drive structure with two main driving

wheels and an additional free caster wheel for balance, which is consistent with the drive system commonly used in mobile robot navigation studies. The model was also directly available in the CoppeliaSim built-in library, which made it a practical choice given that the primary focus of this study was the development of the control system rather than the physical robot design.

One limitation of this choice was that the rectangular body shape of the robot has a slightly higher chance of contacting walls during turning maneuvers, especially in narrow areas of the environment than commercial round shaped cleaning robots. Despite this limitation, the Pioneer P3DX model was considered sufficiently suitable as a simulation platform for developing and evaluating the proposed control system.

4.2.2 LiDAR Sensor Configuration

A 360-degree scanning configuration was selected because it provides environmental perception around the full body of the robot. This allows the robot to detect walls, obstacles, and open regions from multiple directions during navigation. Such sensing is important for boundary following, obstacle avoidance, and occupancy map generation. The configuration is also consistent with the intended hardware direction of this thesis, where the existing Roborock platform includes a rotating 360-degree LiDAR assembly. Therefore, the simulation sensor configuration was designed to approximate the full-surrounding perception required for later physical deployment.

4.3 Navigation Strategy

4.3.1 Exploration Strategy Selection

Several exploration strategies were considered for the first simulation stage, including random walk and frontier-based exploration. Random walk was excluded because it does not guarantee systematic coverage of the environment and can result in large unexplored areas, particularly in large and irregularly shaped indoor spaces such as the GMIT main hall and corridor layout. Frontier-based exploration was also excluded because it requires a fully integrated SLAM system and significant additional computational complexity, which was beyond the scope of this study.

Boundary following was therefore selected as the exploration strategy. This method is well suited for large indoor environments when combined with a 360-degree LiDAR sensor with sufficient sensing range, as it allows the robot to systematically traverse the environment boundaries and generate a complete occupancy map without requiring

complex localization or planning algorithms. The 360-degree sensor configuration and 15 meter scan range used in this study provided sufficient environmental perception to support reliable boundary following navigation throughout the entire GMIT simulation environment [1], [2].

A known characteristic of boundary following exploration is that the robot traces the environment perimeter without physically entering the interior floor area. The interior regions of the map are therefore populated entirely from LiDAR rays cast inward from boundary positions. For the selected GMIT environment, interior areas located far from the boundary walls rely on rays approaching the 15 meter maximum scan range. At this range, discretization into the 0.05 meter grid resolution introduces angular quantization error that increases with distance. The practical effect is that interior free-space cells in large open areas may contain a higher density of noise and irregular occupied cells compared with cells near the boundary. The MATLAB noise removal and morphological processing pipeline described in Section 4.4.2 is intended to mitigate this effect, and its output is evaluated qualitatively in Section 6.2.

4.3.2 Exploration Completion Condition

In the boundary exploration stage, the robot needed a reliable condition to determine when the exploration of the environment was sufficiently complete and the occupancy map was ready for export. However defining a reliable completion condition required careful consideration. If the completion condition was defined only based on the robot returning to its starting position, the exploration would stop immediately at the beginning of the simulation, since the robot is already located at the starting position when the exploration begins.

To prevent this false completion, two additional conditions were defined:

- A minimum simulation time had elapsed
- A minimum travel distance had been covered

Only after satisfying all three conditions was the exploration stage considered complete and the occupancy map exported for the MATLAB processing stage.

4.4 Map Processing Strategy

4.4.1 Occupancy Grid

The LiDAR sensor used in this study operates in two dimensions. Therefore, occupancy grid mapping is directly compatible with the 2D LiDAR data, because it represents the environment as a grid of cells where each cell is classified as free, occupied, or unknown based on the sensor measurements. This makes it a natural and appropriate map format for the selected sensing configuration and for transferring map data between CoppeliaSim and MATLAB.

Occupancy grid mapping is also one of the most widely used mapping methods in mobile robot navigation, especially for indoor environments [5], [6]. It provides a simple and reliable representation of the environment that is suitable for later path planning operations. Since the cleaning robot only moves on the floor plane, the three-dimensional simulation environment does not need to be used directly for path planning. Instead, the environment can be represented as a two-dimensional occupancy map.

As the robot moves through the selected GMIT first-floor area, the LiDAR measurements update the grid map by identifying free, occupied, and unexplored regions [3]. After the exploration completion condition is satisfied, the generated occupancy map is exported as a CSV file for further processing in MATLAB.

4.4.2 MATLAB-Based Mapping

The occupancy grid map is stored as a two-dimensional grid, so it can be processed efficiently using MATLAB's matrix and geometric operations. This makes MATLAB suitable for filtering, modifying, and analyzing the map data. Since the raw occupancy map is generated from LiDAR measurements during robot motion, it may include small mapping errors such as noisy occupied cells, disconnected free-space regions, or incomplete boundary sections. These issues can reduce the reliability of the map representation [7] .

Therefore, MATLAB-based mapping is used as an intermediate stage between the raw occupancy map and the later system stages. The purpose of this stage is to convert the raw map into a cleaner and more structured form. Instead of relying directly on the raw map, MATLAB processing is expected to prepare map outputs that are easier to use and more consistent.

In this study, the MATLAB mapping stage is expected to produce three main outputs. The first output is a cleaned main free-space region, which represents the reachable floor area of the selected environment. The second output is a safer navigation map, where the usable free space is adjusted according to the robot size and safety margin. The third output is an extracted edge representation, which describes the boundary between free and occupied regions.

4.5 Cleaning Method Selection

In this study, a combined cleaning method is selected for the proposed autonomous cleaning robot system. The cleaning approach is based on two main path planning strategies: edge cleaning and zig-zag coverage. This approach is selected because the target environment includes both boundary areas and wide open floor areas, and a single cleaning pattern is not suitable for all parts of the environment [8], [9].

Edge cleaning:

Edge cleaning is selected for the areas near walls and fixed obstacles. These areas require a different cleaning behavior from the interior floor area because boundary regions can include curved walls, irregular edges, and internal obstacles. If only a zig-zag pattern is used, the robot may not follow these boundaries closely enough. Therefore, edge cleaning is used to guide the robot along the boundary regions of the environment.

Zig-zag coverage:

Zig-zag coverage is selected for the main interior floor area. This pattern is suitable for large indoor spaces because it is simple, systematic, and easy to generate from a two-dimensional map [8]. Compared with random cleaning movement, zig-zag coverage provides a more organized path and reduces unnecessary repeated movement. It also matches the main hall and corridor structure of the selected GMIT first-floor area, where large reachable floor regions need to be covered efficiently.

The lane spacing for the interior zig-zag coverage is set to 0.35 meters. This value is selected to be smaller than the 0.5 meter coverage tracking width, producing an overlap of 0.15 meters between adjacent lanes. This overlap ensures that the transition between lanes does not leave an uncleaned strip even when small path tracking deviations occur during lane direction changes. The relationship between lane spacing L, coverage width W, and overlap O is given by O = W minus L. With W equal to 0.5 meters and L equal to 0.35 meters, the overlap is 0.15 meters, representing 30 percent

of the coverage width. This overlap margin is consistent with practical cleaning robot configurations where partial lane overlap compensates for path following imprecision.

By combining edge cleaning and zig-zag coverage, the selected method provides a more suitable cleaning strategy for the proposed simulation system. Edge cleaning focuses on boundary regions, while zig-zag coverage focuses on the main reachable floor area. This separation allows each part of the environment to be cleaned using a path pattern that matches its shape and cleaning requirement.

4.6 Robot Motion and Path Tracking

4.6.1 Tracking Method

In this study, pure pursuit is selected as the tracking method for executing the planned cleaning paths. The purpose of the tracking method is to guide the robot along the desired cleaning path while maintaining stable and continuous motion. This is important because the planned path includes straight sections, curved boundary sections, and direction changes near walls and obstacle boundaries.

Pure pursuit is selected because it provides a simple and practical tracking approach for this type of simulation system. The method uses a point ahead of the robot on the desired path as the tracking target, allowing the robot to follow the overall path shape instead of reacting only to its current position. This helps the robot move more smoothly along both boundary-following and interior coverage paths, especially in sharp corners [11].

Alternative methods such as the Stanley controller and PID heading control were considered. The Stanley controller requires accurate velocity estimation, which is less reliable at the low speeds typical of cleaning missions. PID heading control is simple but does not naturally handle curved paths or smooth waypoint transitions. Pure pursuit was selected for its simplicity, stable behavior across both straight and curved path segments, and validated use in indoor navigation contexts [11].

During mission execution, the robot position is estimated using the ground-truth pose provided by CoppeliaSim. This is equivalent to assuming perfect localization throughout the mission. In a physical deployment, localization would need to be maintained using a combination of wheel odometry and LiDAR-based position correction. Odometry-only navigation over the full mission distance would accumulate positional drift that could cause the robot to deviate from planned paths, particularly in the long corridor sections

of the selected environment. Addressing localization in physical deployment is identified as a necessary extension of this work and is discussed further in Section 7.6.

4.6.2 Differential-Drive Motion Control

The Pioneer P3DX robot uses a differential-drive structure. Therefore, the robot movement is controlled by the left and right wheel velocities. When both wheels move at the same speed, the robot moves forward. When the wheel speeds are different, the robot turns [6].

This drive method is suitable for the proposed system because it is simple and matches the selected robot model. The tracking method provides the required forward and turning motion, and this motion is converted into left and right wheel velocities for the robot. This allows the robot to follow the planned cleaning path using its two driving wheels.

4.6.3 Speed Control

Speed control is included to keep the robot movement stable during path tracking. The robot should move fast enough to complete the cleaning mission efficiently, but not so fast that it becomes unstable near corners, curved boundaries, or direction changes.

In this study, the robot speed is limited by minimum and maximum speed values. The maximum speed prevents the robot from moving too aggressively, while the minimum speed prevents it from stopping unnecessarily during normal tracking. The turning speed is also limited to reduce sharp rotation and improve stability.

A slower motion is also used when the robot approaches the end of a path section. This helps the robot reduce overshooting and makes the movement smoother before changing to the next part of the planned path.

4.7 Evaluation Method

4.7.1 Evaluation Criteria

The developed system is evaluated by observing whether the robot successfully completes the full cleaning mission in the CoppeliaSim simulation environment. A successful mission requires the robot to complete the following sequential tasks: follow the planned edge cleaning paths along the environment boundaries and internal obstacle edges, complete the interior zig-zag coverage paths, move through transition

paths between cleaning sections, and return to the starting position after the cleaning mission is finished. It should be noted that the coverage evaluation is performed within the same simulation environment used for map generation and path planning. Because the map is derived from the same simulated space in which execution takes place, the coverage result reflects performance under ideal conditions where the map accurately represents the true environment geometry. This is an inherent characteristic of simulation-based validation and means the reported coverage percentage should be interpreted as an upper bound on expected real-world performance rather than a direct prediction of physical deployment results.

4.7.2 Success Metrics and Thresholds

The primary performance metric used in this study is the cleaning coverage percentage, which is calculated as the ratio of covered cleanable cells to the total number of cleanable cells in the environment. A coverage result of 95 percent or above is defined as a successful outcome for this study. This threshold was selected because minor uncovered areas near sharp corners or narrow boundary sections are acceptable given the physical limitations of the robot size and the complexity of the environment geometry.

4.7.3 Coverage Assessment Method

The cleaning coverage percentage is calculated and displayed continuously during the simulation, and the final value recorded at mission completion is used as the main evaluation result. For coverage tracking purposes, a cleaning width of 0.5 meters is applied around the robot center position at each simulation step. This value is selected to represent the effective cleaning swath of the intended physical platform rather than the Pioneer P3DX simulation model, which has no cleaning mechanism. The target Roborock platform uses side brushes that extend beyond the robot body width, providing an effective cleaning swath wider than the 0.41 meter robot body. The 0.5 meter tracking width represents a conservative estimate of this extended swath, adding approximately 0.045 meters of effective brush reach on each side of the robot body. This assumption should be validated against the specific brush geometry of the target hardware platform during physical deployment.

5. Implementation

This chapter describes the implementation of each system component following the design decisions established in Chapter 4. Readers are referred to Chapter 4 for the rationale behind each design choice, this chapter focuses on the specific configuration and construction details. Also the actual construction and configuration of the autonomous cleaning robot system developed in this study. Following the design decisions established in the Methods chapter, each component of the system was built and configured using CoppeliaSim EDU, SketchUp 2017, and MATLAB R2025b. The implementation is presented in the same sequential order as the system workflow, beginning with the simulation environment construction and ending with the mission execution stage.

5.1 Simulation Environment Reconstruction

5.1.1 SketchUp Layout Reconstruction Process

The GMIT first-floor main hall and corridor area was reconstructed in SketchUp 2017 using a printed copy of the architectural floor plan as a visual reference. Since the selected area could not be separated from the rest of the floor plan image, the boundary lines of the main hall and corridor were manually traced by drawing lines directly over the inserted floor plan photo. The wall boundaries were drawn by following the visible structure lines of the selected area, and the edges were then offset inward to create wall thickness throughout the layout.

Fixed obstacles present in the selected area were also modeled in the reconstruction. Circular column obstacles were represented using circle shapes, while rectangular obstacles such as heating units and fixed furniture were represented using rectangle shapes. Only permanently fixed obstacles were included, as established in the methodology.

After the 2D layout was completed, the **Measure Tape** function in SketchUp was used to scale the model to approximate real dimensions. By assigning a known real dimension to one specific wall segment of the building, the entire drawing was scaled proportionally to match the actual environment size. Finally, the **Extrude** function was used to convert the 2D floor plan drawing into a 3D model by giving height to all wall and obstacle shapes. The resulting 3D model represents the main hall and corridor layout of the GMIT first floor and is shown in Figure 5.1.

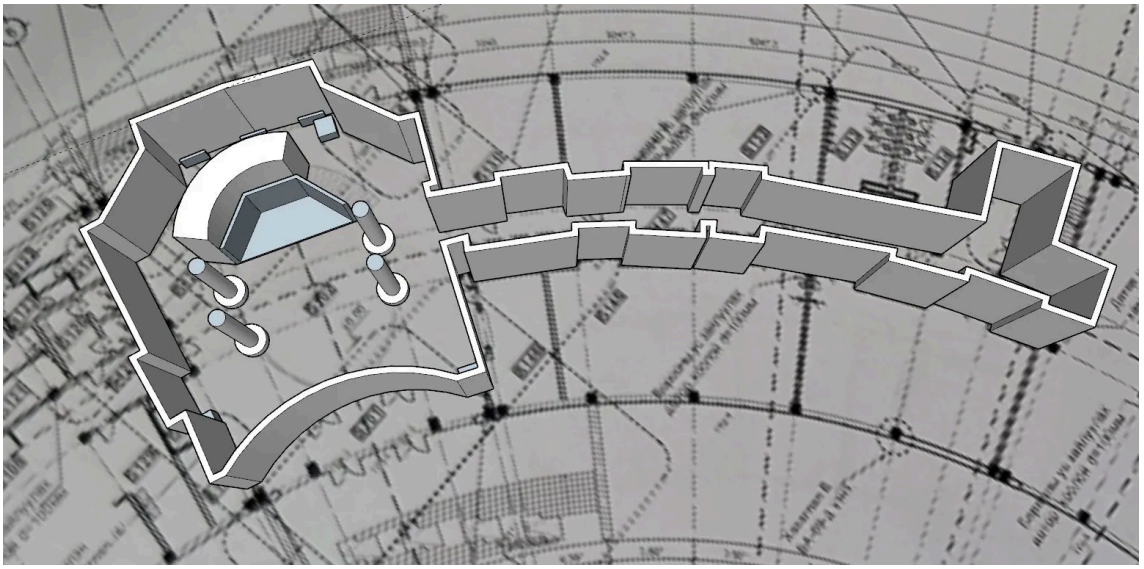


Figure 5.1 Sketchup 3D Reconstruction of the GMIT Main Hall and Corridors

5.1.2 CoppeliaSim Environment Import and Scaling

The reconstructed 3D model was exported from SketchUp as an OBJ file and imported into CoppeliaSim EDU to form the simulation environment. Upon import, the model appeared significantly smaller than the required simulation scale and needed to be resized. The default floor plane in CoppeliaSim had a size of 5 × 5 meters, which was not large enough to accommodate the full GMIT layout. Therefore both the floor and the imported model were resized using the Isometric Scaling function in CoppeliaSim. The floor was scaled up to 50 × 50 meters and the imported GMIT model was scaled to a bounding box size of approximately 7.38 × 22.77 × 55.14 meters as shown in Figure 5.2, which closely reflects the approximate real dimensions of the selected environment.

After scaling, the model was repositioned and rotated diagonally within the simulation space. This adjustment was necessary because the selected GMIT layout has a long and irregularly curved shape that did not fit within the simulation floor in its default orientation.

Finally the Respondable dynamic property of the imported model was enabled through the object dynamic properties settings in CoppeliaSim. This allowed the environment model to interact physically with the robot during simulation, ensuring that the robot could not pass through walls or obstacles during execution. The completed and scaled simulation environment is shown in Figure 5.2.

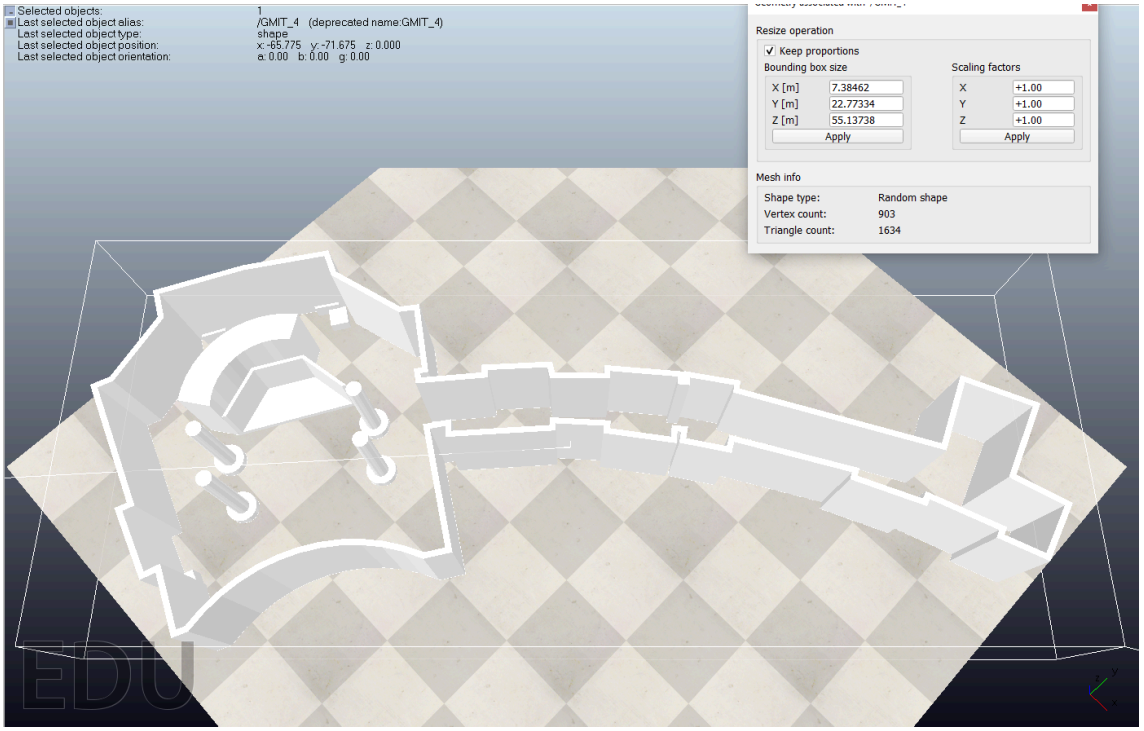


Figure 5.2 GMIT Simulation Environment and Its Dimensions in CoppeliaSim

5.2 Robot and Sensor Implementation

5.2.1 Pioneer P3DX Model

The Pioneer P3DX model was loaded directly from the CoppeliaSim built-in model library and used without modification to its physical dimensions, as the default model dimensions were already close to those of a realistic large indoor cleaning robot. The physical dimensions and drive parameters used in the simulation correspond to the real Pioneer P3DX specifications and were not modified, as the default values were appropriate for the scale of the selected environment. The robot uses a differential-drive structure with two main driving wheels and a free caster wheel for balance. The robot body dimensions are shown in Figure 5.3.

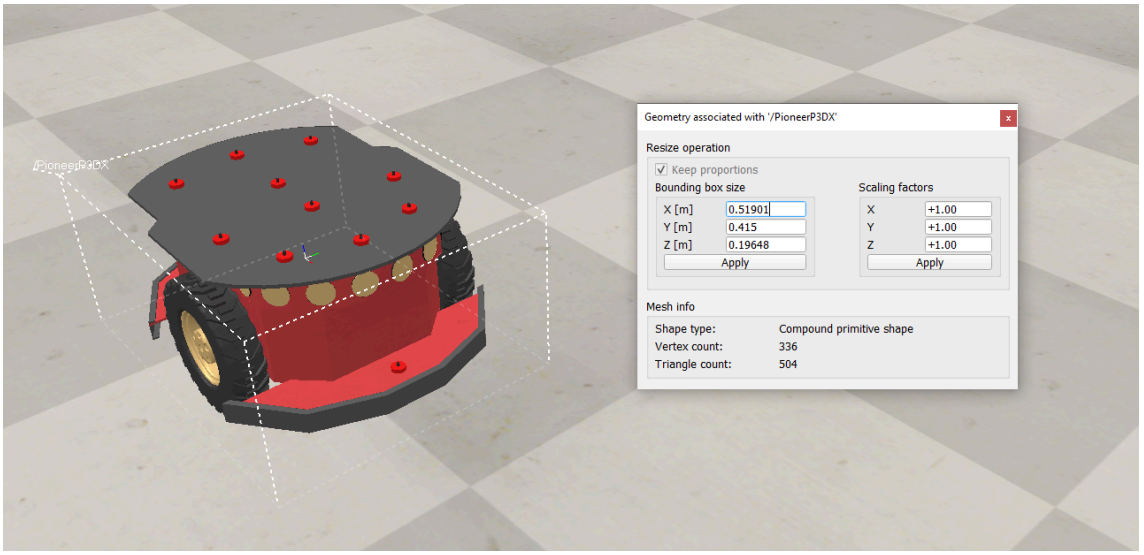
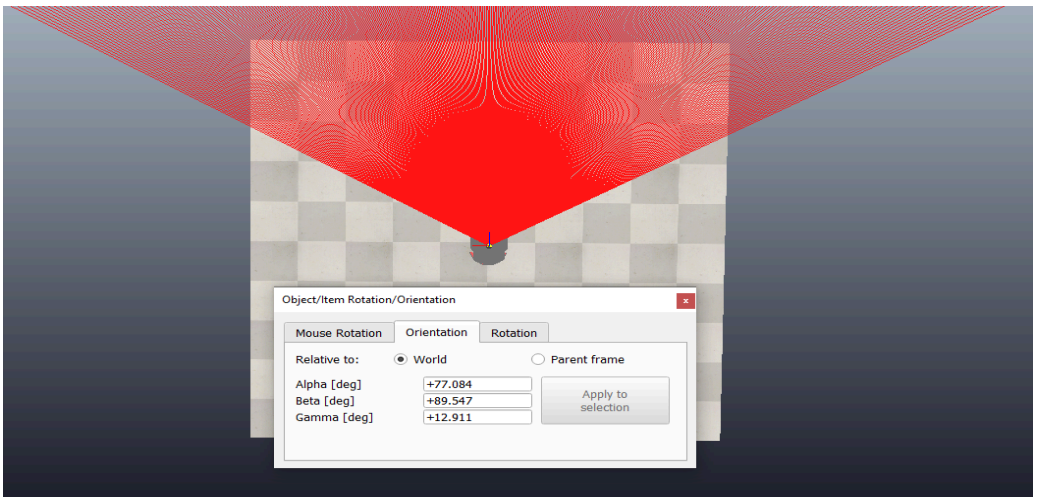


Figure 5.3 Pioneer P3DX Robot Model and Its Dimensions

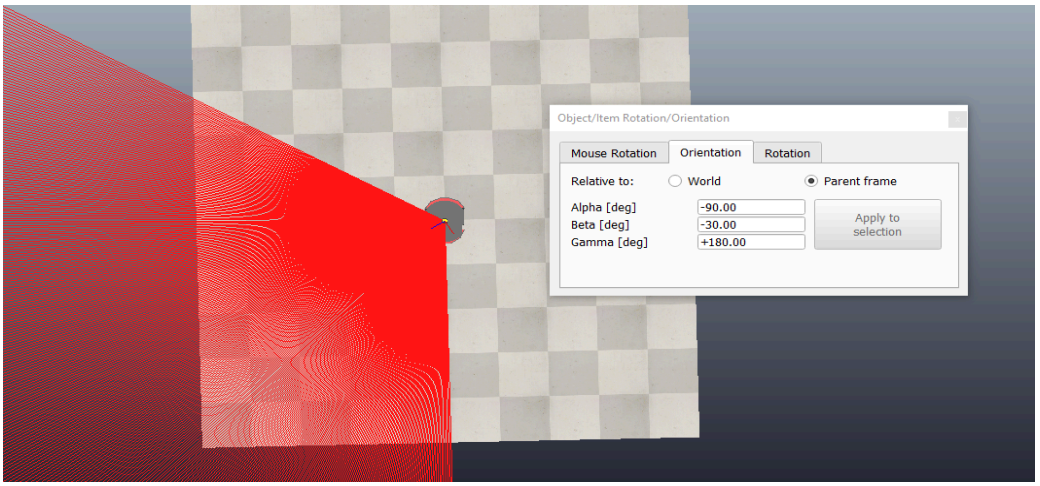
5.2.2 LiDAR Sensor Assembly

The default fastHokuyo LiDAR model available in the CoppeliaSim library provides a scanning range of 240 degrees, which was not sufficient for the full surrounding coverage required in this study. To achieve complete 360-degree scanning coverage, a third fastHokuyo sensor unit was added to the assembly. Each of the three sensors was assigned a 120-degree scanning sector, and their orientations were adjusted so that the three sectors covered the full 360-degree range around the robot without overlap or gap. The three sensors were mounted closely together at the top center of the robot body and grouped as a single object in the simulation scene tree under the parent object.

a) Sensor 1 (Front)



b) Sensor 2 (Left – Back)



c) Sensor 3 (Right – Back)

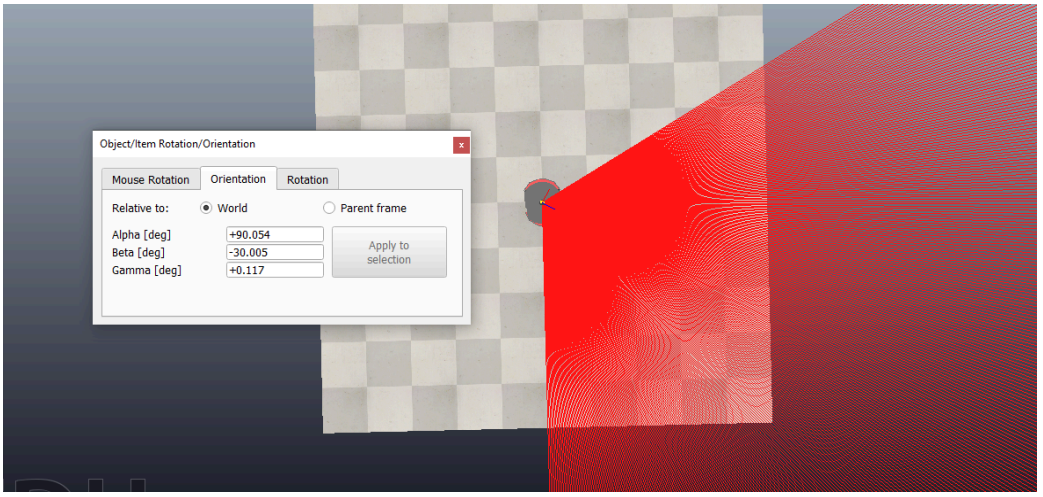


Figure 5.4 LiDAR Sensor Assembly

5.2.3 Sensing Script and LiDAR Data Pipeline

The built-in fastHokuyo sensing script was modified to support the three-sensor 360-degree LiDAR assembly used in this study. The maximum scan range was increased to 15 m and the output was changed from simple range values to X, Y and Z point coordinates in the robot body frame. The combined point data from all sensors was exported through a single CoppeliaSim string signal, named lidarData, so that the navigation and mapping script could use one unified data stream for wall-distance estimation, obstacle detection and occupancy grid generation.

5.3 Stage 1: Exploration and Map Export

The first simulation stage was implemented in CoppeliaSim to allow the robot to explore the selected GMIT indoor environment and generate an occupancy grid map. In this stage, the robot did not perform cleaning. The main purpose was to collect LiDAR data from the environment and create a usable map for the later MATLAB map processing and path planning stages.

The exploration script was organized using a state-based structure. Two main states were used: **STATE_EXPLORE** and **STATE_FINISHED**. During the exploration state, the robot moved through the environment using right-wall-following behavior while the occupancy grid map was updated. When the exploration was completed, the system changed to the finished state, where the robot stopped and exported the generated map. The main state definitions and navigation parameters are shown in Appendix A1.

During each simulation step, the robot received point data from the 360-degree LiDAR sensor assembly. The LiDAR data was read from the `lidarData` signal and converted into point coordinates in the robot body frame. This allowed the same sensor data to be used for navigation, obstacle detection, and map generation. The LiDAR data reading process is shown in Appendix A2.

The navigation logic used different parts of the LiDAR point cloud. Points in front of the robot were used to calculate the nearest front obstacle distance. This helped the robot detect walls or obstacles before collision. Points on the right side of the robot were used to estimate the right-wall distance. This distance was needed for maintaining a stable distance from the wall during boundary following. The front and right-side distance extraction logic is shown in Appendix A3.

The right-wall-following behavior was implemented by comparing the measured right-wall distance with the desired wall distance. If the robot moved too close to the wall, the wheel speeds were adjusted to turn the robot slightly away from the wall. If the robot moved too far from the wall, the robot turned slightly toward the wall. When an obstacle was detected in front of the robot, the robot entered a turning mode and turned left for a short time. This allowed the robot to follow corners and continue moving along the boundary. The main right-wall-following control logic is provided in Appendix A4.

Additional logic was included to make the exploration more reliable. A stuck recovery function was used when the robot's position changed very little for a certain time. In this

condition, the robot performed a short recovery movement to continue exploration. This was necessary because the selected GMIT environment includes long corridors, curved boundaries, and irregular wall geometry, where the robot could otherwise become trapped or stop progressing.

The occupancy grid map was updated during the sensing callback at a fixed update interval. The robot pose and LiDAR measurements were used together to update the grid. The robot's current position was marked as free, and each valid LiDAR ray was converted from the robot frame to the world frame. The cells along the ray were marked as free, while the endpoint of the ray was marked as occupied. In this way, the map gradually represented open floor areas, walls, and fixed obstacles. The occupancy grid update logic is shown in Appendix A6.

The exploration completion condition was not based only on returning near the starting position. If only this condition was used, the exploration would stop immediately at the beginning because the robot already starts at the initial position. Therefore, three conditions were checked together: minimum exploration time, minimum travelled distance, and return near the starting position. When all three conditions were satisfied, the boundary loop was considered complete. This completion logic is shown in Appendix A5.

After the exploration was completed, the robot entered the finished state. The final occupancy grid map was exported as a CSV file, and the related map metadata was exported as a separate text file. The map file stored free, occupied, and unknown cell values, while the metadata file stored information such as map resolution and grid boundaries. These files were used as the input for the MATLAB map processing stage. The CSV map export logic is provided in Appendix A7.

The main CoppeliaSim callbacks controlled the execution order of the simulation. The actuation callback updated and executed the current robot state, while the sensing callback updated the occupancy map at the selected interval. The cleanup callback stopped the robot at the end of the simulation. These callback functions are shown in Appendix A.8.

5.4 MATLAB Map Processing Pipeline

The raw occupancy map exported from CoppeliaSim was processed in MATLAB before the cleaning paths were generated. This step was necessary because the raw LiDAR-based map could contain small noise regions, disconnected free-space areas,

and narrow unreliable openings. If the path planning stage used the raw map directly, the generated path could pass too close to walls or through areas that were not safe for the robot.

The MATLAB script first loaded the exported occupancy grid and metadata files. The grid values were then separated into three categories: free, occupied, and unknown. Free cells represented areas where the robot could move, occupied cells represented walls and fixed obstacles, and unknown cells represented areas that were not observed during exploration. The map loading and cell classification logic is shown in Appendix B1.

After the raw map was loaded, the free-space map was cleaned using morphological operations. Small isolated free-space regions were removed because they were usually caused by mapping noise and were not useful for navigation. Erosion was then applied to remove narrow unreliable openings. After this step, the largest connected free-space region was selected as the main reachable area. This ensured that the following path planning steps used only the region that was actually connected and reachable by the robot.

Table 5.1 Parameters Used for Noise Removal and Free Region Extraction

Parameter	Value	Purpose
`smallFreeAreaCells`	50	Removes small isolated free-space regions
`narrowWidthCells`	3	Defines the erosion size used to remove narrow unreliable openings
`closeRaduisCells`	1	Defines the light smoothing size after reconstruction

After selecting the main reachable region, dilation was applied to recover the approximate original size of the free-space area after erosion. Light smoothing was also applied using closing and opening operations. This produced a cleaner final free-space map while still removing unreliable regions from the raw occupancy data. The free-space cleaning and main region extraction logic is shown in Appendix B2.

A safe navigation map was then generated from the cleaned map. Since the robot has a physical body size, the planned path cannot be placed directly beside walls or obstacles. Therefore, the occupied regions were inflated using the robot radius and an additional safety margin. This expanded the obstacle areas in the map and reduced the risk of the robot colliding with walls or fixed obstacles during path following.

Table 5.2 Parameters Used for Safe Transition Map Generation

Parameter	Value	Purpose
`robotWidthM`	0.41	Defines the physical width of the robot
`robotRadiusM`	0.205	Represents half of the robot width
`safetyMarginM`	0.05	Adds extra clearance from walls and obstacles
`inflationRadiusM`	0.255	Total obstacle expansion distance
`inflationCells`	Based on `mapResolution`	Converts the inflation distance into grid cell

The safe navigation map was created by dilating the obstacle map using the calculated inflation value. The remaining connected free region was then selected as the final safe navigation area. This map was used later for transition path generation and interior zig-zag coverage planning. The safe navigation map generation logic is shown in Appendix B3.

Edge polylines were extracted from the boundary between the final free-space region and the obstacle region. For each free cell, the script checked the neighboring cells. If a neighboring cell was not part of the free-space region, the shared side of the cell was saved as an edge segment. These edge segments were then connected into polylines and simplified to remove unnecessary repeated or nearly straight intermediate points. The edge extraction and polyline simplification logic is shown in Appendix B4 and Appendix B5.

The processed outputs were saved as MAT and CSV files for the next path planning stage. The file `exact_edge_map_meters_gmit.csv` stored the extracted edge polylines in meter coordinates. The file `edge_polylines_gmit.mat` stored the processed maps, safe navigation map, edge polylines, and map metadata. The file `coverage_cleanable_map_gmit.csv` stored the cleanable free-space area used for coverage planning and final coverage evaluation, while `coverage_meta_gmit.csv` stored the map resolution and grid boundary information. The export logic for these processed files is shown in Appendix B6.

5.5 Cleaning Path Generation

Cleaning path generation was performed in MATLAB after the map processing stage. The purpose of this stage was to convert the processed map outputs into a complete waypoint-based cleaning mission. The complete cleaning path was generated in three main steps: edge-cleaning path generation, interior zig-zag coverage generation, and final mission assembly. The final output of this stage was a single mission CSV file that could be imported into CoppeliaSim for execution.

The edge-cleaning paths were generated first from the extracted edge polylines. These polylines represented the boundary between the free-space region and the wall or fixed obstacle regions. Since the robot cannot follow directly on the wall boundary, each polyline was offset by a selected distance to create a robot-center path. Both left and right offset directions were tested, and the side with the better valid path length inside the free-space map was selected. This edge offset and validation logic is shown in Appendix C1.

Table 5.3 Parameters Used for Edge-Cleaning Generation

Parameters	Value	Purpose
`edgeOffset`	0.3 m	Defines the robot-center distance from the wall or obstacle edge
`sampleStep`	0.05 m	Defines the final waypoint spacing for edge-cleaning paths
`preSampleStep`	0.04 m	Defines the point spacing before offsetting the edge polyline
`minPathLength`	0.6 m	Removes very short edge-cleaning paths
`minPoints`	8	Removes path segments with too few points
`minClearance`	0.25 m	Defines the minimum clearance used for safe edge transfer paths

After the valid edge side was selected, the edge route was cleaned and resampled. Short segments, sharp spikes, and invalid points were removed. The route was then resampled using the selected waypoint spacing so that the robot could follow the path more smoothly during simulation. A validity check was also applied to ensure that most of the path points remained inside the free-space map. The cleaned edge paths were then exported as route waypoint data. This process is shown in Appendix C2.

The generated edge routes were not executed randomly. They were ordered to reduce unnecessary movement between separated edge-cleaning sections. The longest route was selected as the first route, and the following routes were selected using the nearest route endpoint. If reversing a route produced a shorter connection, the route direction was flipped. This allowed the edge-cleaning mission to move between route sections more efficiently.

Safe transfer paths were also generated between consecutive edge-cleaning routes. For each route pair, the end point of the current route and the start point of the next route were snapped to nearby safe points in the safe navigation map. A graph-based transfer path was then generated between these safe points. Short connector segments were also used to connect the cleaning route to the safe transfer path and from the safe transfer path to the next route. The edge route ordering and safe transfer generation logic is shown in Appendix C3.

The second part of the path planning stage was the interior zig-zag coverage generation. The safe navigation map was used as the planning area so that the generated lanes would stay away from inflated walls and obstacles. Parallel sweep lanes were generated across the free-space region. In this work, the sweep direction was set along the x-axis, so the planner generated horizontal sweep lanes. Each lane was divided into waypoint points using the selected sample step. This lane generation logic is shown in Appendix C4.

Table 5.4 Parameters Used for Interior Zig-Zag Coverage Generation

Parameters	**Value**	**Purpose**
`laneSpacingM`	0.35 m	Distance between adjacent zig-zag sweep lanes
`sweepSampleStep`	0.05 m	Point spacing along each sweep lane
`sweepDir`	x	Direction of sweep lane generation
`transferSampleStep`	0.05 m	Point spacing for transition paths between lanes
`minClearanceM`	0.2 m	Minimum clearance used when selecting safe transition points
`minLanePoints`	4	Minimum number of points required for a valid lane
`planDS`	3	Downsampling factor used for fast transition path planning

`clusterDistM`	1 m	Distance threshold used to group nearby lanes into regions

The generated sweep lanes were ordered before exporting the interior mission. First, the center point of each lane was calculated. Nearby lanes were then grouped into regions using the selected clustering distance. Inside each region, the lanes were ordered using a nearest-neighbor method. If reversing a lane reduced the distance from the previous lane, the lane direction was flipped. After ordering the lanes within each region, the regions themselves were also ordered to reduce long travel between separated areas.

After the lane ordering step, the interior mission was constructed. Each sweep lane was added as a cleaning segment. Between consecutive lanes, the planner selected safe points near the end of the current lane and the start of the next lane. A graph-based path was then generated between these safe points using the downsampled planning grid. This allowed the robot to move between interior lanes without crossing unsafe or occupied areas. The interior lane ordering and safe transfer mission generation logic is shown in Appendix C5.

The final step was to combine the edge-cleaning mission and the interior zig-zag mission into one complete mission. The edge mission was placed first because boundary cleaning was intended to be completed before the interior floor coverage. To prevent ID overlap, the mission IDs and route IDs of the interior mission were shifted after loading the edge mission. Then, a safe connector path was generated from the last point of the edge mission to the first point of the interior mission.

Table 5.5 Parameters Used for Final Mission Assembly

Parameters	**Value**	**Purpose**
`connectorSampleStep`	0.05 m	Resampling step for connector and return paths
`planDS`	3	Downsampling factor used for fast path planning
`minClearanceM`	0.2 m	Minimum clearance used when selecting safe connector points
`returnToStart`	`true`	Enables return-to-start movement
`returnMode`	`missionStart`	Sets the return target as the first mission point

A mission origin value was added to the final mission file to identify the source of each waypoint. Edge-cleaning points were marked as edge origin, interior-cleaning points were marked as interior origin, connector points were marked separately, and the return path was marked as return-to-start origin. After the edge and interior missions were connected, a final return-to-start path was generated from the last mission point back to the starting area. This return path was also generated using safe points and graph-based planning through the safe navigation map.

The final combined mission was exported as `final_cleaning_mission_gmit_safe.csv`. This file contained the ordered waypoint list for the complete cleaning task, including edge-cleaning routes, safe edge transfers, the edge-to-interior connector, interior zig-zag lanes, interior transfers, and the return-to-start path. The final mission combination, return-to-start generation, and CSV export logic are shown in Appendix C6.

5.6 Stage 2: Mission Execution

The second simulation stage was implemented in CoppeliaSim to execute the final cleaning mission generated from MATLAB. In this stage, the robot did not perform exploration or map generation. Instead, it loaded the final waypoint mission file and followed the planned path in the correct order. The main purpose of this stage was to test whether the robot could complete the full cleaning mission and measure the final cleaning coverage.

At the beginning of the simulation, the script loaded three main files: the final cleaning mission CSV file, the cleanable coverage map, and the coverage metadata file. The final mission file contained the ordered waypoints for edge cleaning, edge transitions, the edge-to-interior connector, interior cleaning, interior transitions, and the return-to-start path. The coverage map and metadata were loaded so that the covered floor area could be calculated during simulation. The main execution parameters, including robot dimensions, wheel properties, controller settings, coverage width, and coordinate transformation values, are shown in Appendix D1.

The mission CSV file was read and converted into structured waypoint data. Each row contained the mission ID, segment type, route ID, mission origin, and x-y waypoint coordinates. After loading, the rows were grouped by mission ID so that each mission segment could be treated as a separate path. This made it possible for the robot to

complete one path segment and then switch to the next segment in the correct order. The mission loading and path grouping logic is shown in Appendix D2.

Each mission segment was classified according to its mission origin and segment type. This classification separated the path into edge-cleaning routes, edge transitions, interior-cleaning routes, interior transitions, the connector path, and the return-to-start path. Different movement speeds were assigned depending on the path type. Cleaning routes were allowed to move faster than connector or transition paths, while connector and return movements used a lower speed for safer motion. The path classification and speed selection logic is shown in Appendix D3.

Before execution, the loaded waypoint paths were transformed into the CoppeliaSim coordinate system. Duplicate consecutive points were removed, and each path was resampled using a fixed spacing. This helped produce smoother waypoint sequences for the pure pursuit controller. After this preparation, the mission started from the first path segment, and the robot followed each segment one by one until all planned mission sections were completed.

The robot followed the mission path using a pure pursuit-based controller. During every actuation step, the robot pose was read from the simulation. The controller then searched the current path and selected a lookahead point ahead of the robot. This lookahead point was selected based on the closest point on the path and the predefined lookahead distance. Using a point ahead of the robot helped the robot follow the general path shape smoothly instead of only reacting to the nearest waypoint. The lookahead point selection logic is shown in Appendix D4.

After the lookahead point was selected, the controller calculated the required linear and angular velocity. The target point was first transformed into the robot local coordinate frame. Then, the controller calculated the path curvature needed to move toward the lookahead point. A heading correction term was also added to reduce heading error. When the robot approached the end of a path segment, the speed was reduced using the slowdown radius. This reduced overshooting before switching to the next mission segment. The pure pursuit control and velocity calculation logic is shown in Appendix D5.

The Pioneer P3DX model uses a differential-drive motion structure, so the calculated linear and angular velocities were converted into left and right wheel velocities. If both wheel velocities were similar, the robot moved forward. If the wheel velocities were different, the robot turned. This allowed the same control method to follow straight

zig-zag lanes, curved edge-cleaning paths, transfer paths, connector paths, and the return-to-start path. The wheel velocity conversion logic is also shown in Appendix D5.

The main execution loop was handled inside the CoppeliaSim actuation callback. During each actuation step, the script checked whether the mission had already finished. If the mission was still active, the script selected the current path segment, calculated the lookahead point, checked the distance to the segment end, and generated the wheel commands. When the robot reached the end of the current path segment within the goal tolerance, the script stopped the robot briefly and switched to the next path segment. This process continued until all path segments were completed. The main mission execution callback is shown in Appendix D6.

Cleaning coverage was tracked during the sensing callback. The robot position was used to mark cleanable map cells as covered. The effective cleaning width was applied around the robot position, and all cleanable cells inside this area were marked as covered. Interpolation was used between the previous and current robot positions so that covered cells were not missed when the robot moved between simulation steps. The number of covered cells was updated continuously during the mission. The coverage tracking logic is shown in Appendix D7.

The final coverage percentage was calculated by comparing the number of covered cleanable cells with the total number of cleanable cells in the coverage map. During simulation, the interface displayed the current coverage percentage and the current mission mode, such as edge cleaning, edge transition, interior cleaning, connector, or return-to-start. After the final path segment was completed, the robot stopped, the mission state changed to finished, and the final coverage percentage was recorded as the main performance result for Chapter 6.

This stage completed the full simulation workflow of the developed control system. The planned mission was generated externally in MATLAB, imported into CoppeliaSim, followed by the robot using pure pursuit control, and evaluated using coverage tracking. Therefore, the second simulation stage verified whether the generated edge-cleaning, interior zig-zag, transfer, connector, and return paths could be executed successfully by the simulated differential-drive robot.

6. Results

6.1 Stage 1: Exploration Results

6.1.1 Boundary Following Behavior

The boundary following exploration was carried out in the first CoppeliaSim simulation stage. The robot successfully traversed the full boundary of the GMIT main hall and corridor environment using the right-wall following strategy. The complete exploration path is shown in Figure 6.1.

As observed in Figure 6.1, the robot traced the outer boundary of the environment as well as the boundaries around internal fixed obstacles, covering the full perimeter of the selected area. The exploration required approximately 40 to 60 minutes of real time to complete, with the simulation time running at approximately 20 to 25 minutes. The low simulation frame rate caused by continuous occupancy map updates at 0.05 m resolution resulted in a slow step-by-step execution.

One to two stuck events were observed during the exploration, occurring near sharp boundary corners where the robot became temporarily trapped. In both cases the stuck detection and recovery mechanism successfully detected the condition and applied a short recovery motion, after which the robot resumed normal wall following behavior without requiring any manual intervention or script modification.

The robot returned to its starting position after completing the full boundary loop, satisfying all three exploration completion conditions: minimum elapsed simulation time, minimum accumulated travel distance, and proximity to the starting pose. The occupancy map was then automatically exported as a CSV file for the MATLAB processing stage.

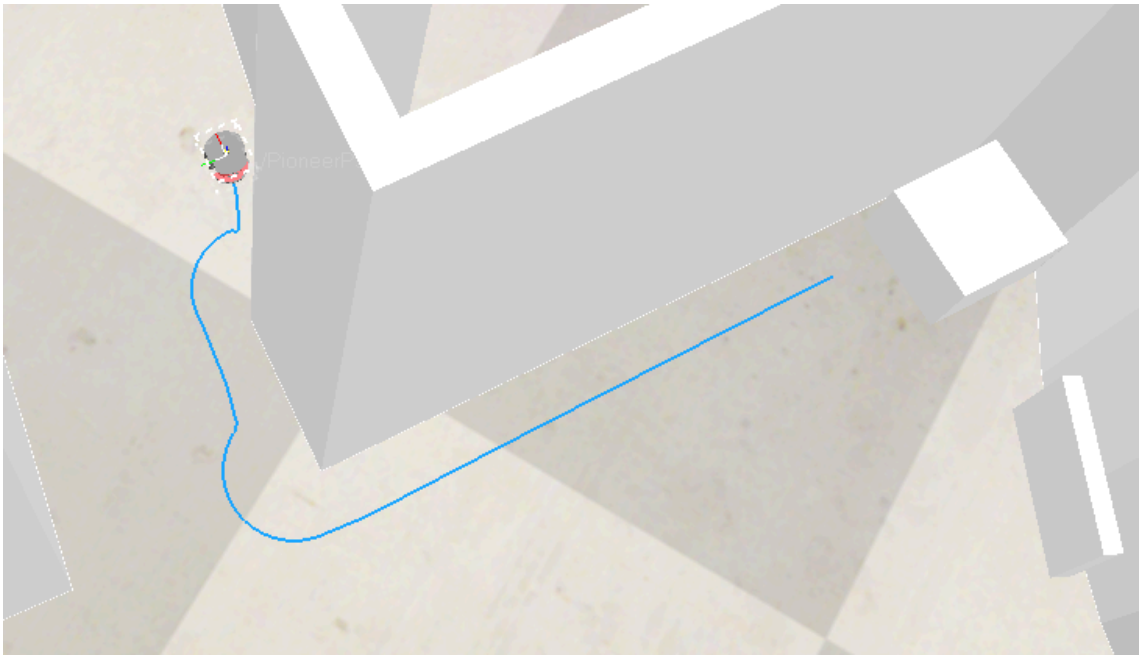


Figure 6.1 Robot Exploration Track in CoppeliaSim — Boundary Following Result

6.1.2 Generated Occupancy Map Quality

The occupancy map generated during the exploration stage is shown in Figure 6.2. The map displays the free space regions in white and occupied or unknown regions in black, representing the GMIT main hall and corridor layout as detected by the LiDAR sensor during boundary following exploration.

The generated map successfully captured the overall shape of the selected environment, including the curved corridor structure, the main hall area, and the internal fixed circular column obstacles visible as black filled regions within the free space. The long curved corridor extending toward the upper right of the map and the wider main hall area at the lower left were both clearly represented with well defined boundaries.

The map quality was considered sufficient for the subsequent MATLAB processing stage. The free space region was largely continuous and well connected throughout the environment, with clear separation between free and occupied areas along most boundary sections. Some minor noise and irregular boundary sections were observed near sharp corners and narrow areas, which were expected given the discretization of the LiDAR measurements into a 0.05 m grid resolution. Some minor noise and irregular boundary sections were observed near sharp corners and narrow areas.

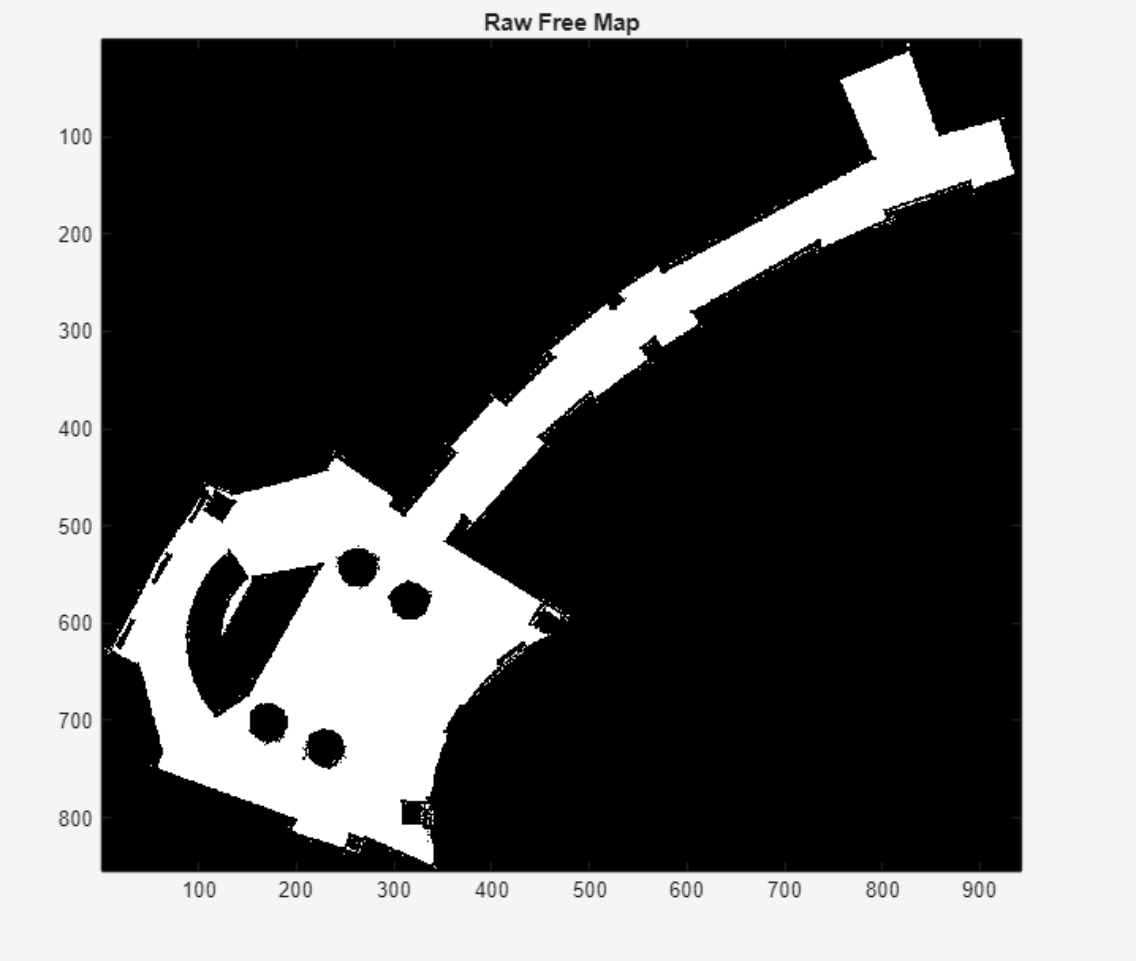


Figure 6.2 Raw Occupancy Map Generated from Exploration Stage

6.2 MATLAB Map Processing Results

6.2.1 Noise Removal and Free Region Extraction Result

The raw occupancy map imported from CoppeliaSim was processed through the noise removal and free region extraction pipeline in MATLAB. The eroded free map and the final cleaned free map are shown in Figure 6.3 and Figure 6.4 respectively.

As shown in Figure 6.3, the erosion step temporarily reduced the free space region and removed narrow unreliable connections between areas. This allowed the largest connected region representing the main reachable floor area to be selected without including disconnected or unreliable free space islands. After the main region was selected, dilation was applied to recover the approximate original size of the free space. The resulting final main free map shown in Figure 6.4 shows a clean and continuous free space representation of the GMIT main hall and corridor layout, with all internal fixed obstacles clearly preserved as non-free regions. Compared to the raw

map, small isolated free regions and weak narrow connections were successfully removed, producing a more reliable map for the subsequent path planning stages.

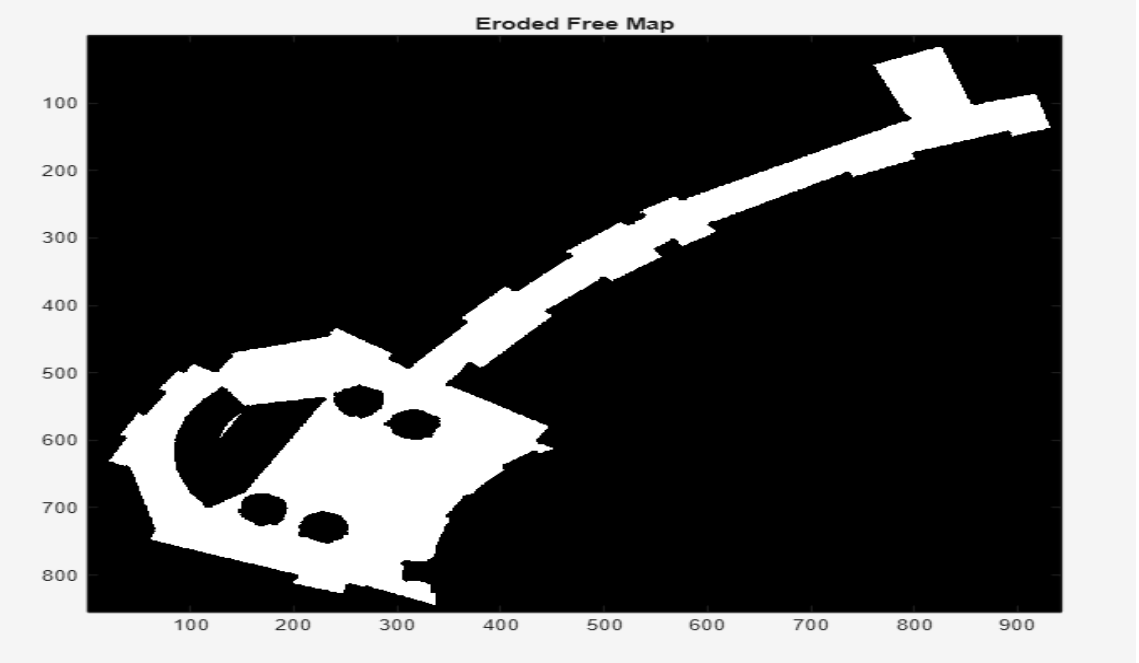


Figure 6.3 Eroded Free Map After Noise Removal

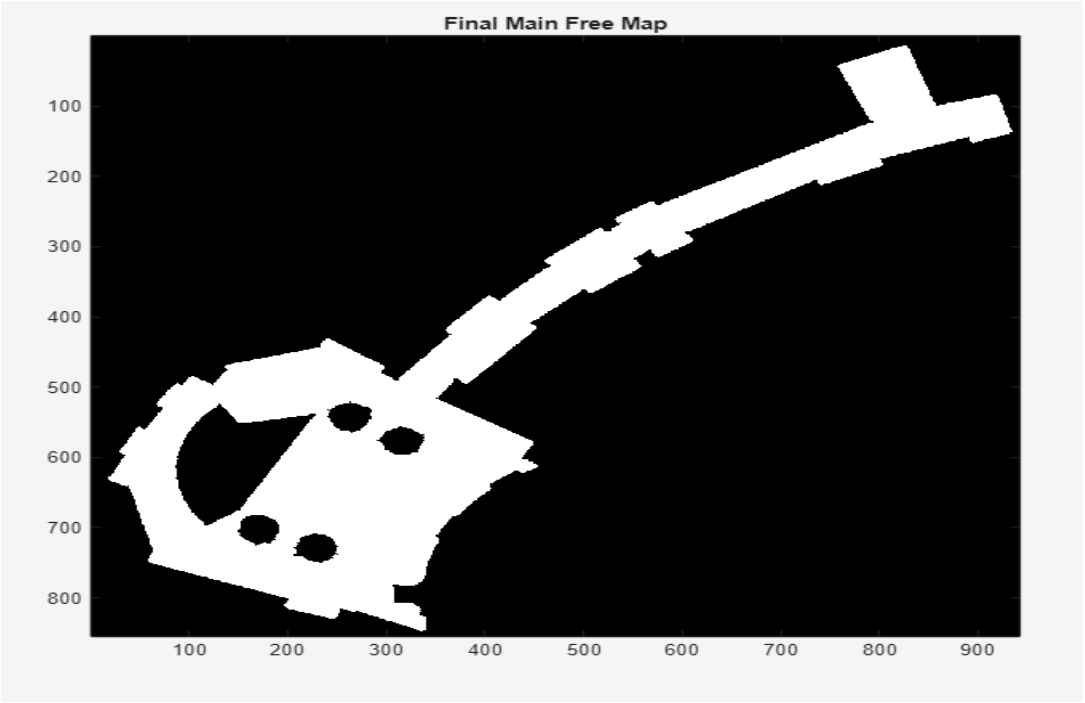


Figure 6.4 Final Main Free Map After Region Extraction

6.2.2 Safe Navigation Map Result

The safe navigation map generated from the cleaned free map is shown in Figure 6.5. The obstacle regions were expanded by an inflation radius of 0.255 m, which accounts for the robot's physical width of 0.41 m and an additional safety margin of 0.05 m. This expansion reduced the available free space around all wall boundaries and internal obstacle boundaries, ensuring that the robot center could move through the remaining free area without risking contact with walls or obstacles during transition movements.

As shown in Figure 6.5, the cyan edge polylines overlay the safe navigation map, showing the boundary between the safe free space and the inflated obstacle regions. The overall shape of the environment was well preserved after inflation, with the main corridor and hall areas remaining navigable. Narrow sections of the environment were reduced more noticeably due to the inflation.

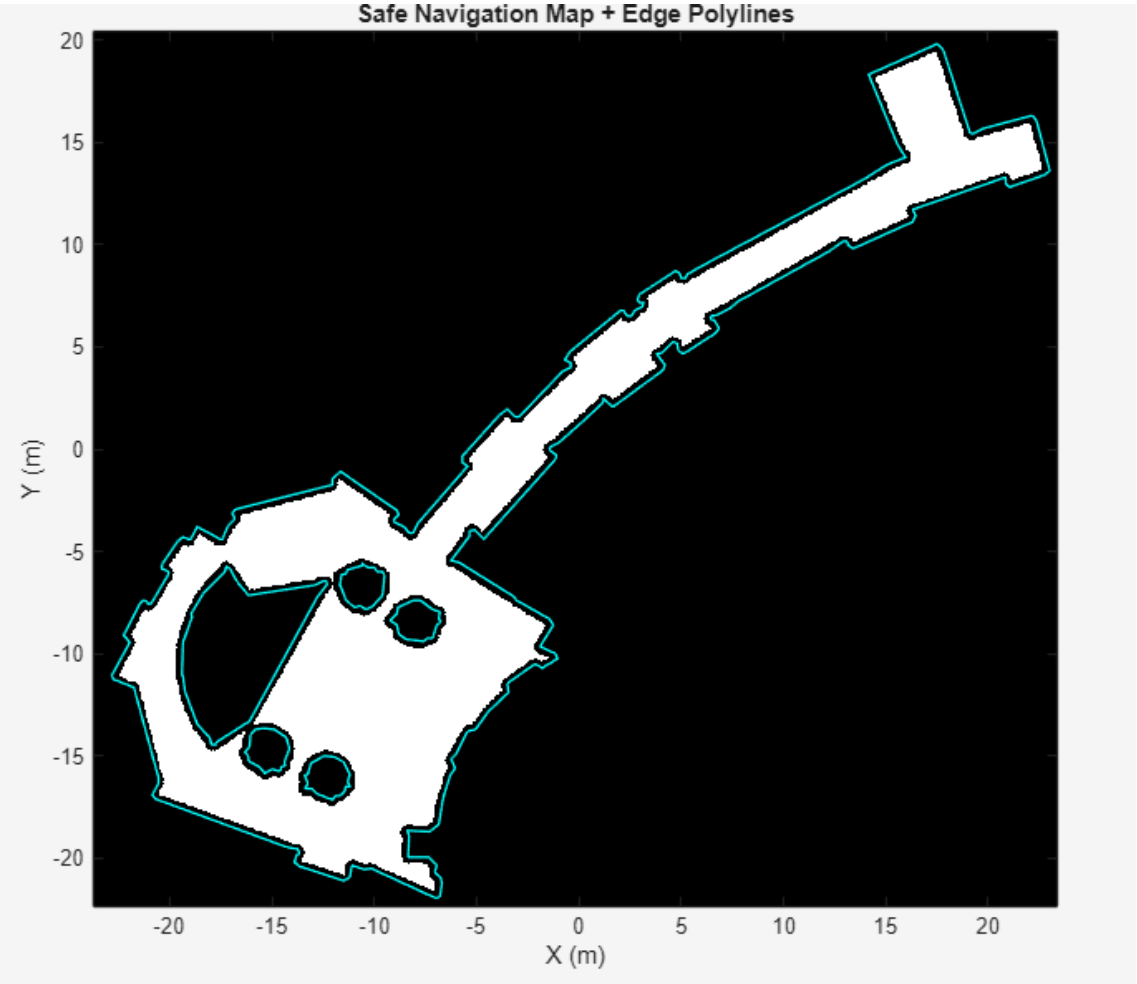


Figure 6.5 Safe Navigation Map with Edge Polylines Overlay

6.2.3 Edge Extraction Result

The extracted edge polylines generated from the final main free map are shown in Figure 6.6. The polylines represent the boundaries between the free space region and all non-free regions, including the outer environment boundary and the boundaries around each internal fixed obstacle.

As shown in Figure 6.6, the edge extraction successfully captured the full outer boundary of the GMIT main hall and corridor layout, including the curved corridor section, the rectangular extension at the upper right, and the irregular shaped main hall at the lower left. The boundaries around all five internal circular column obstacles were also clearly extracted as individual closed polylines. The extracted polylines closely followed the actual environment geometry, providing an accurate boundary representation suitable for edge cleaning path generation in the subsequent stage.

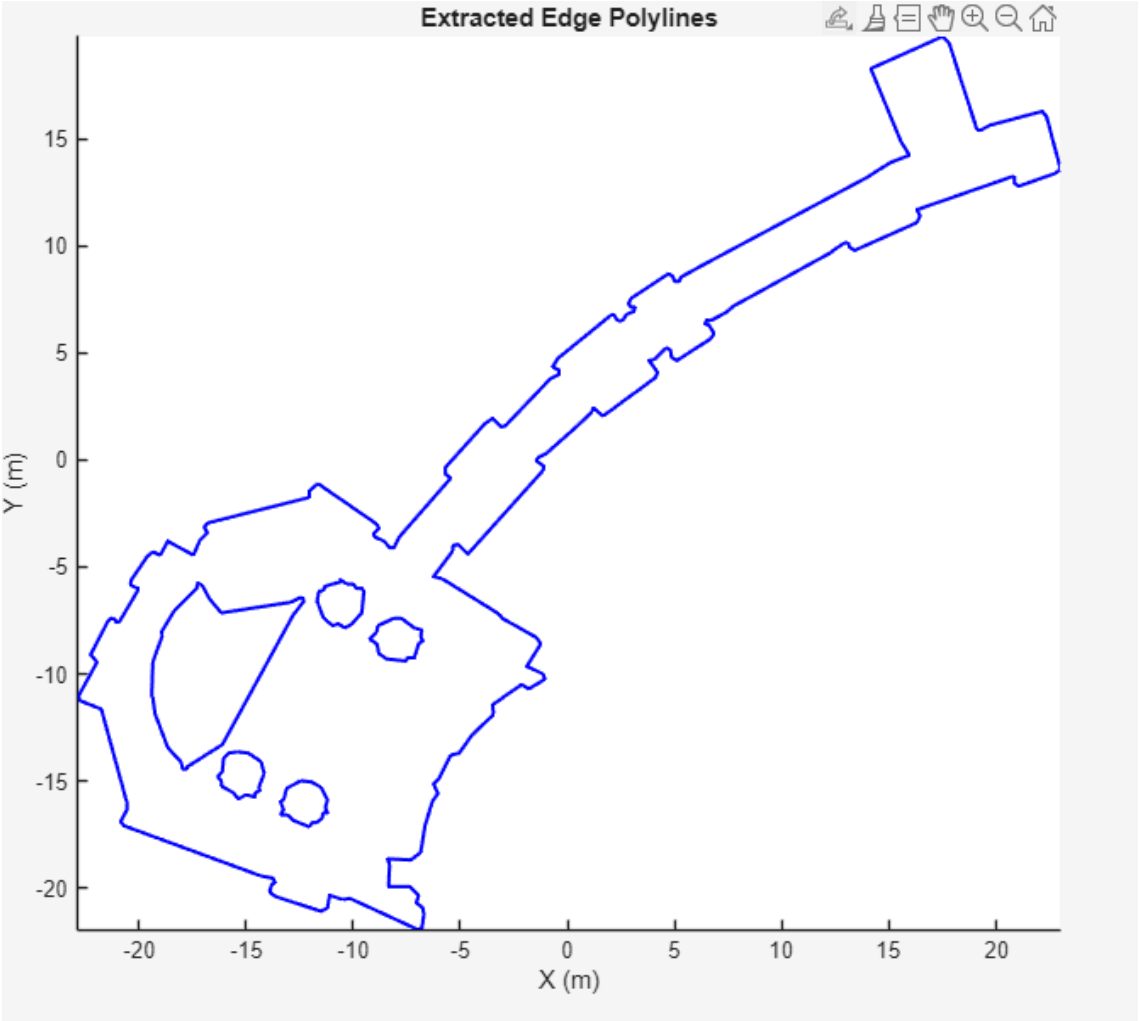


Figure 6.6 Extracted Edge Polylines of GMIT Environment

6.3 Cleaning Path Planning Results

6.3.1 Edge Cleaning Paths

The generated edge-cleaning mission is shown in Figure 6.7. The blue path follows the offset boundary of the selected GMIT first-floor environment, while the cyan lines show the safe transfer paths between separated edge-cleaning routes.

The result shows that the edge-cleaning path was generated around the outer wall boundary and the internal fixed obstacles. The path did not lie directly on the wall edge, but was placed inside the free-space region so that the robot centre could travel with clearance from walls and obstacles.

Some boundary sections were separated into different routes because the environment contains internal obstacles and irregular shapes. These separated routes were connected using safe transfer paths. The generated edge-cleaning mission was then used as the first part of the final cleaning mission.

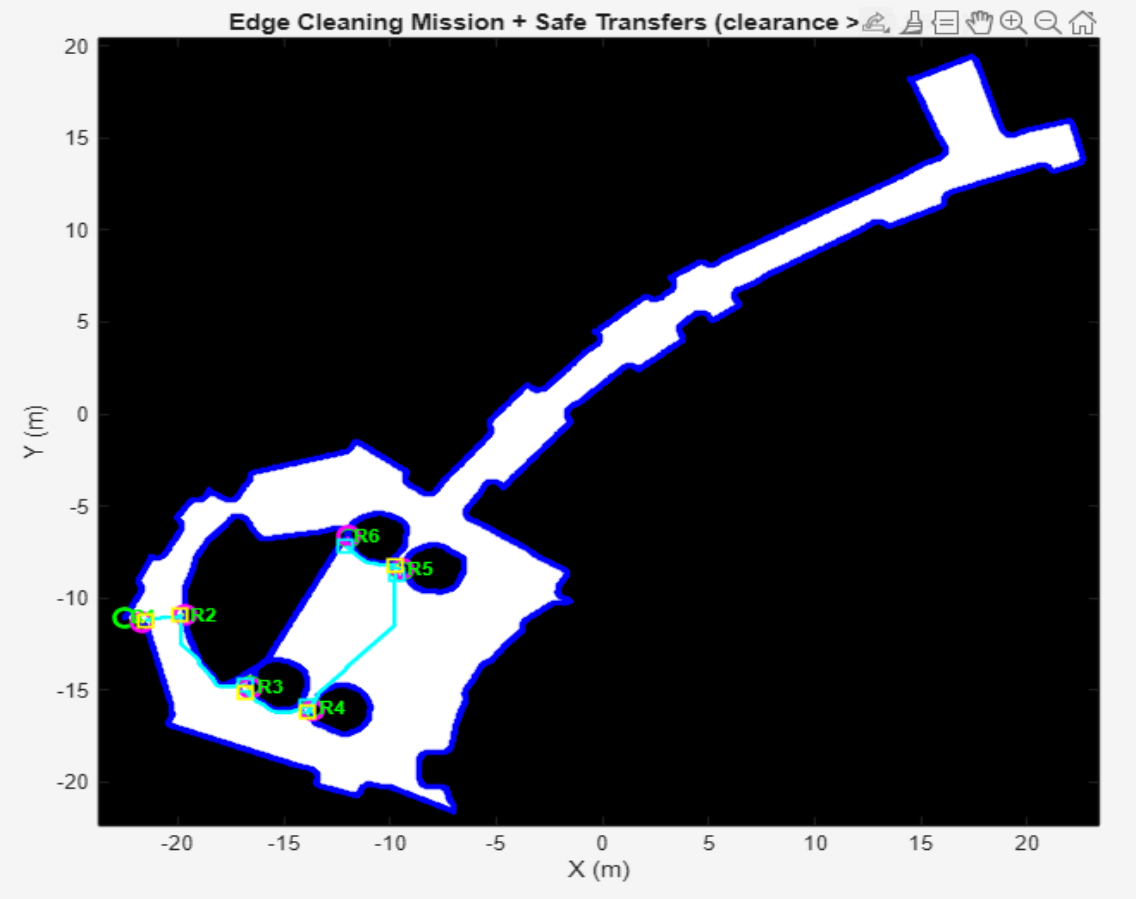


Figure 6.7 Generated Edge-Cleaning Mission with Safe Transfer Paths

6.3.2 Interior Zig-Zag Coverage Paths

The generated interior zig-zag coverage paths are shown in Figure 6.8. The blue sweep lines covered the main free-space region of the GMIT hall and corridor area, while the cyan lines show the transfer paths between separated sweep sections.

The result shows that the zig-zag pattern covered both the large hall area and the long corridor section in an organized way. The path was divided around internal obstacles and boundary changes, so the generated lanes remained inside the valid free-space region.The repeated parallel sweep structure covered the main interior floor area after the edge-cleaning stage.

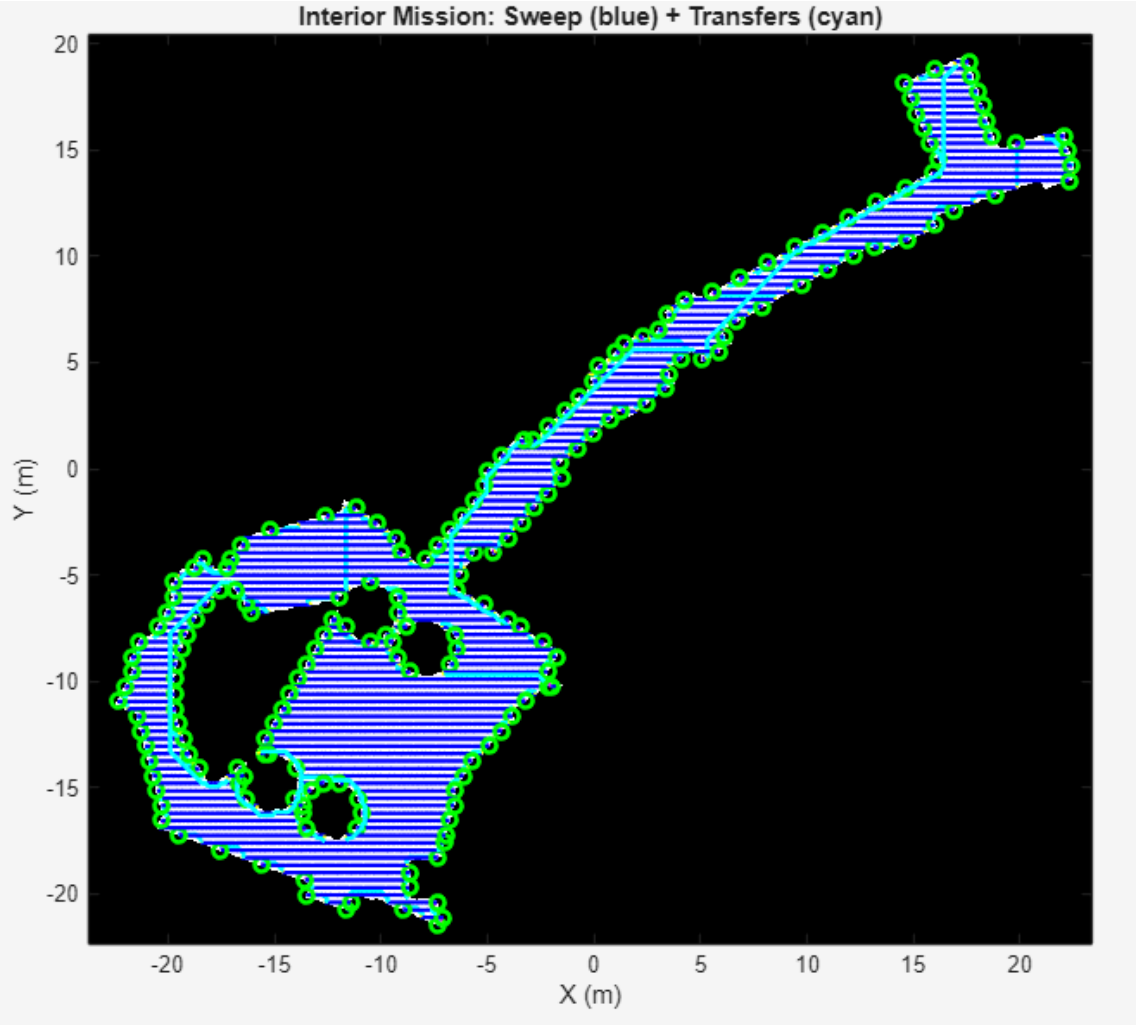


Figure 6.8 Generated Interior Zig-Zag Coverage Paths and Transfers

6.3.3 Final Combined Cleaning Mission

The final combined cleaning mission is shown in Figure 6.9. The mission combines the edge-cleaning path, safe connector path, interior zig-zag coverage path, and return-to-start movement into one complete route.

As shown in the figure, the final mission covers the boundary regions and the interior floor area within the selected GMIT first-floor environment. The edge path follows the outer wall and internal obstacle boundaries, while the interior path fills the remaining open floor area with parallel sweeping lanes. The safe connector and return path allow the separate parts of the mission to be executed as one continuous sequence.

The final combined mission was used as the waypoint input for the second CoppeliaSim simulation stage.

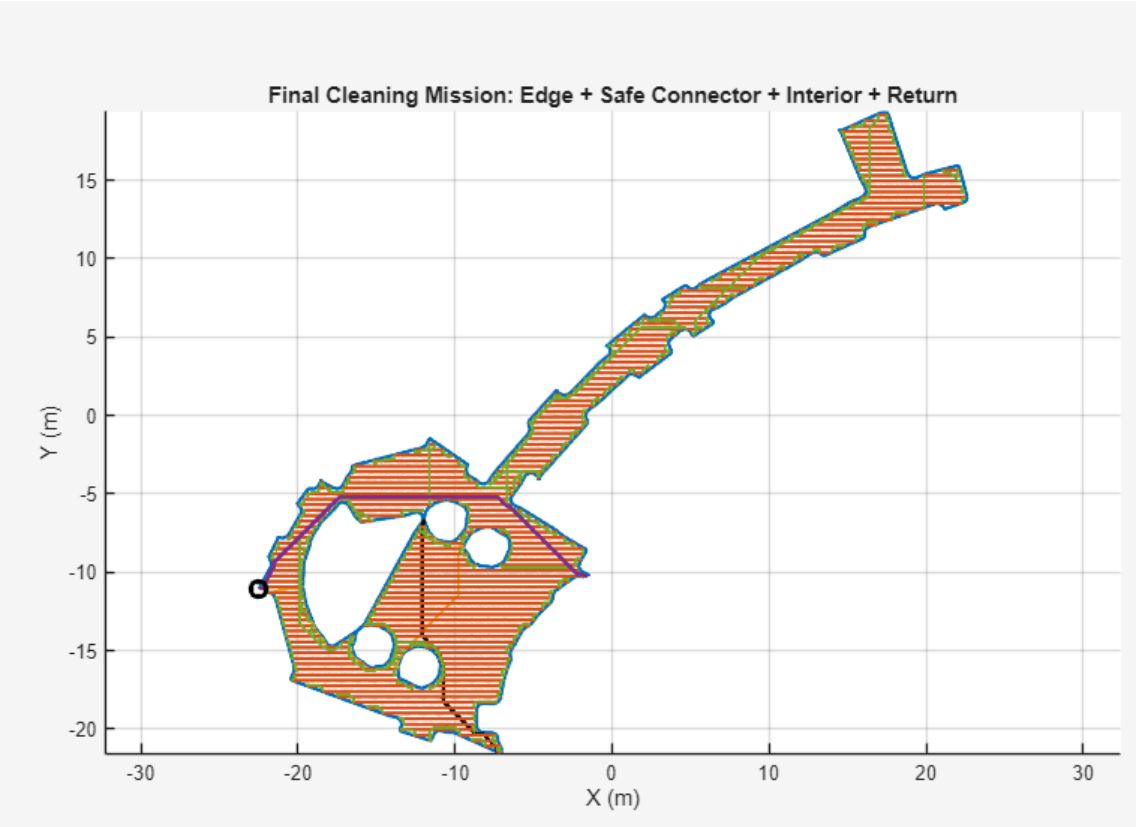


Figure 6.9 Final Combined Cleaning Mission Path

6.4 Stage 2 Mission Execution Results

6.4.1 Path Following Performance

The path-following performance during the final mission execution is shown in Figure 6.10, Figure 6.11, and Figure 6.12. In these figures, the blue path represents the planned edge-cleaning path, the orange parallel lines represent the planned interior zig-zag coverage path. Figure 6.10 shows the general path-following behavior during the final mission. The robot followed the planned route inside the cleanable region without major deviation. This result shows that the planned mission path could be imported and executed in CoppeliaSim using the pure pursuit controller.

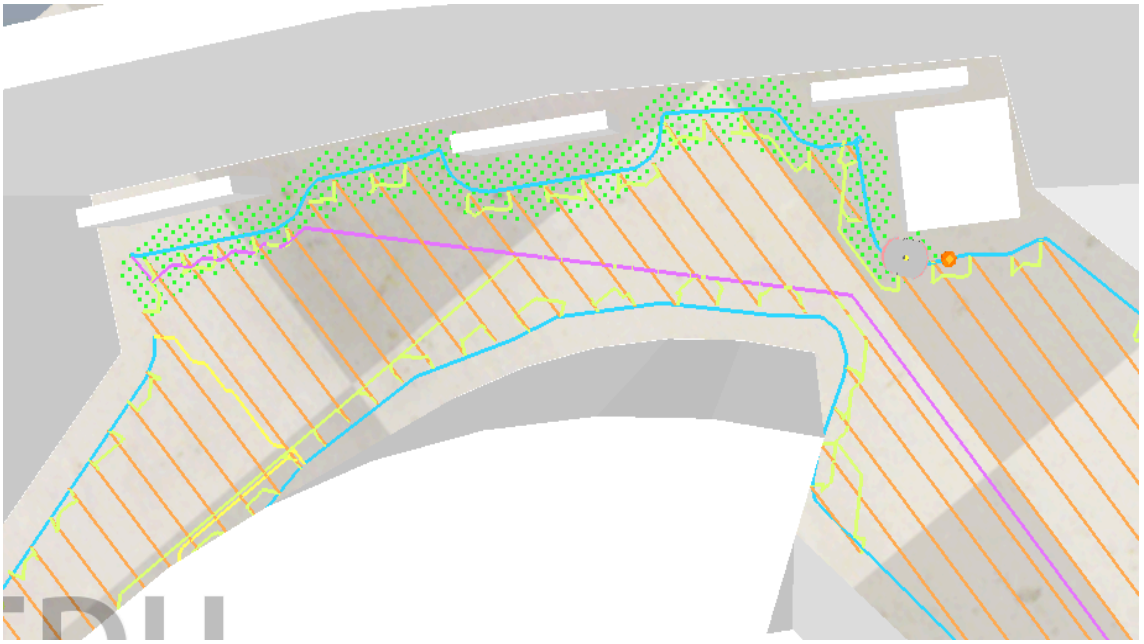


Figure 6.10 Robot Path Following Behavior Result

Figure 6.11 shows the robot path-following result after the edge-cleaning section. The robot was able to move around the outer boundary and internal obstacle regions while remaining inside the valid free-space area.

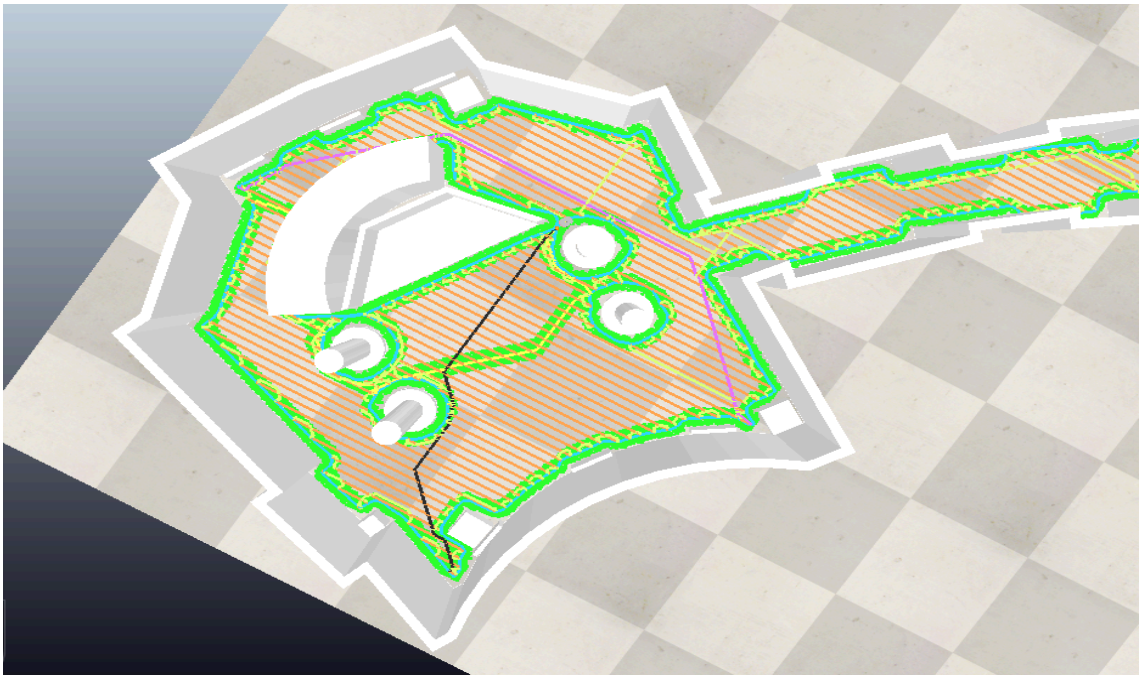


Figure 6.11 Edge-Cleaning Path-Following Result

Figure 6.12 shows the robot following the interior zig-zag coverage path. The robot movement followed the general direction of the planned parallel sweep lanes. Tracking was smoother along straight lane sections, while small deviations appeared during direction changes and connector movements between lanes.

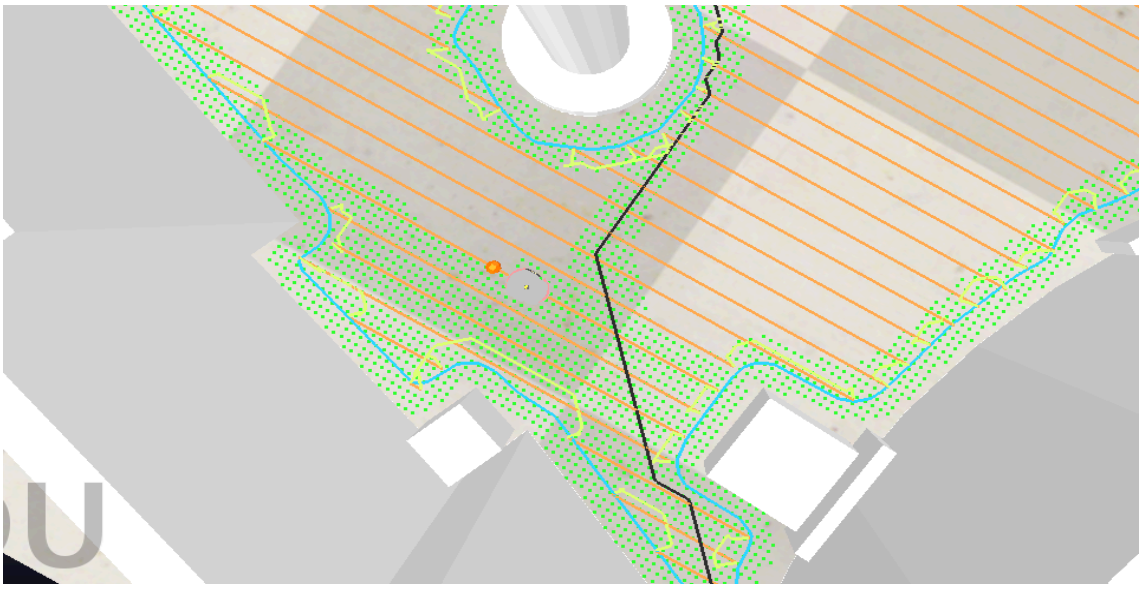


Figure 6.12 Interior Zig-Zag Path-Following Behavior

The robot followed both the edge-cleaning path and the interior zig-zag path during the final mission execution.

6.4.2 Transition Movement Results

The transition movement result is shown in Figure 6.13. In the figure, the yellow path represents the transition route between separated cleaning sections, while the blue and orange paths represent the edge-cleaning and interior zig-zag paths respectively.

The result shows that the transition path connected the cleaning sections without passing through walls or internal obstacles. The robot was able to move from one planned section to the next and continue the mission normally.

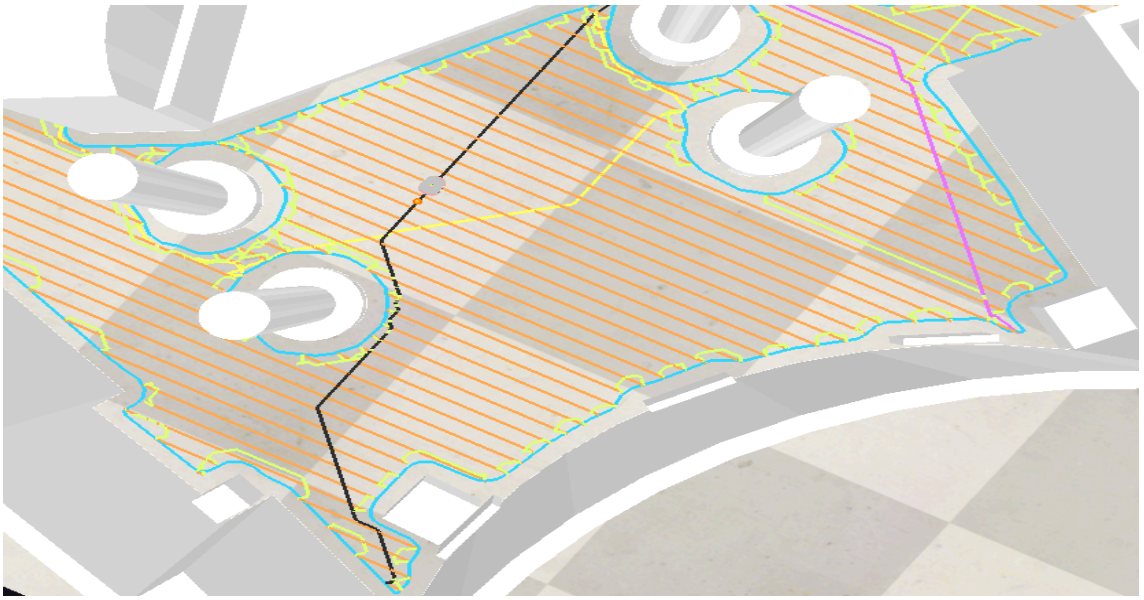


Figure 6.13 Transition Movement Between Cleaning Path Sections

6.4.3 Return to Start Result

The return-to-start result is shown in Figure 6.14. After completing the interior zig-zag coverage section, the robot followed the return path back toward the starting area.

The result shows that the robot returned to the initial mission region without collision or stopping. The final position was located near the starting area.

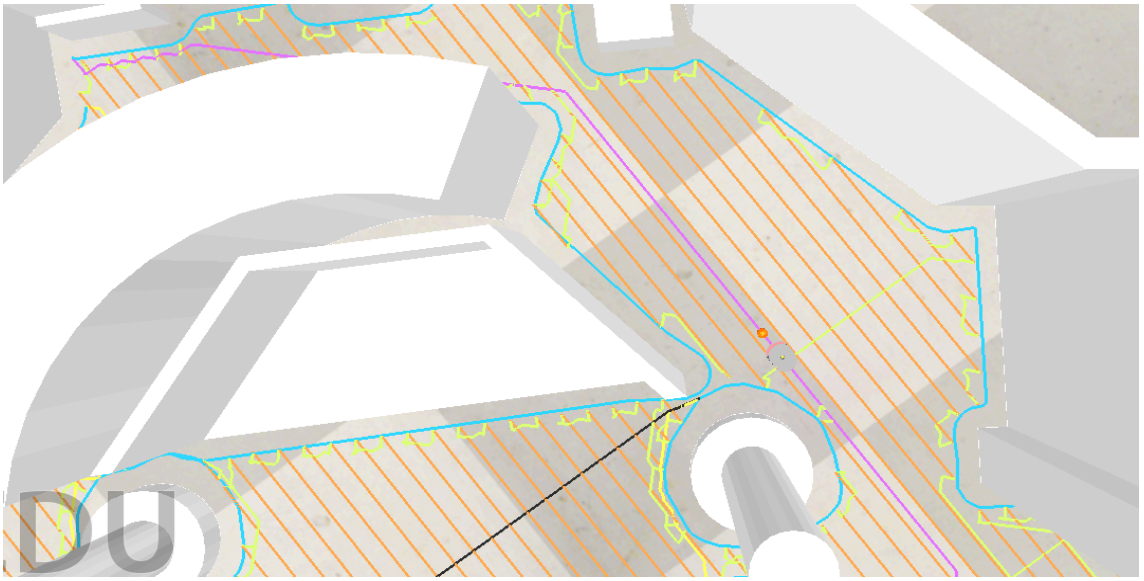


Figure 6.14 Return-to-Start Movement Result

6.4.4 Final Coverage Percentage

The final cleaning coverage result is shown in Figure 6.15. The simulation finished with the coverage map and covered cells displayed over the selected cleanable area.

The final coverage percentage reached **99.4%** at the end of the mission. This value was recorded after the robot completed the edge-cleaning path, interior zig-zag coverage path, transition movements, and return-to-start movement.

Most of the cleanable floor area was marked as covered in the final visualization, while only small uncovered regions remained near some boundary and obstacle-side areas.

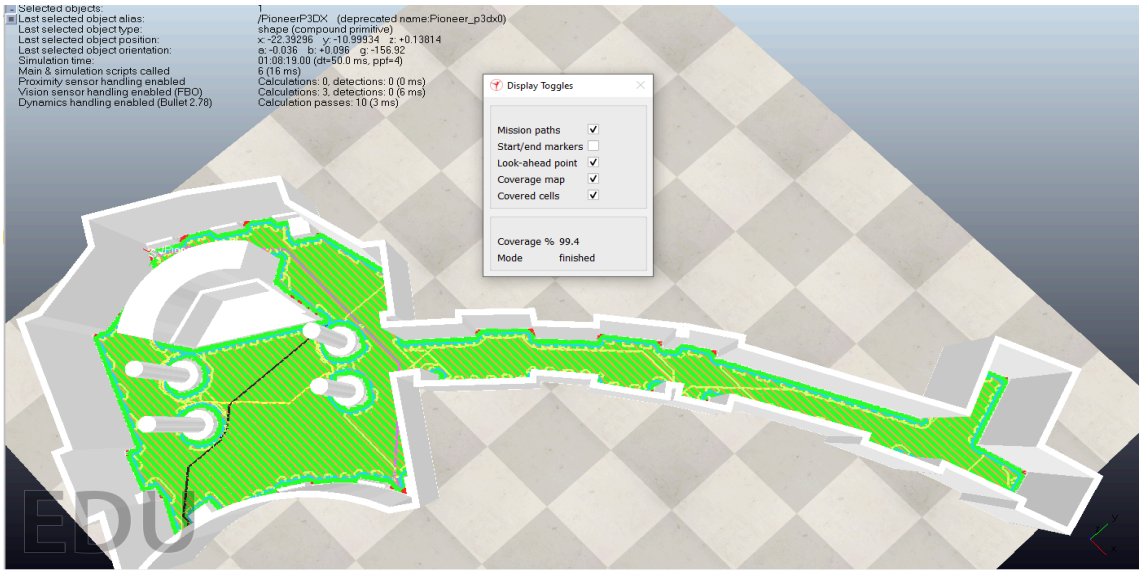


Figure 6.15 Final Cleaning Coverage Visualization After Mission Completion

7. Discussion

7.1 Interpretation of Simulation Results

The simulation results demonstrate that under ideal conditions, where the map and execution environment are identical, the developed two-stage control system achieves the primary objective of the study. Because all stages were validated within the same CoppeliaSim environment, the 99.4 percent coverage result reflects the best-case performance of the path planning and execution pipeline. Real-world deployment would introduce additional sources of error including sensor noise, localization drift, and environmental changes that are not captured in this simulation. These factors are expected to reduce coverage performance and should be addressed in future physical validation.

The sequential completion of all mission stages — boundary exploration, edge cleaning, interior zig-zag coverage, transition movements, and return to start — without manual intervention confirms that the individual components of the system are correctly integrated and that the state-based execution logic functions as intended. The stuck recovery mechanism during exploration and the segment switching logic during execution both operated correctly in all observed cases, indicating that the control logic handles the irregular geometry of the selected environment without requiring environment-specific tuning.

The high coverage percentage obtained in the simulation is the result of the combined cleaning approach adopted in this study. Edge cleaning ensured that boundary regions near walls and fixed obstacles were reached, while the closely spaced zig-zag lanes with 0.35 m lane spacing and 0.50 m coverage width tracking ensured that the interior floor area was systematically covered with adequate overlap between adjacent lanes. The small uncovered regions that remained at the end of the mission, corresponding to approximately 0.6 percent of the cleanable floor area, were primarily located near sharp corners and narrow obstacle-side areas where the physical dimensions of the robot body prevented complete coverage regardless of the path planning strategy.

7.2 Navigation and Path Tracking Performance

The pure pursuit path tracking controller performed well throughout the mission execution stage. Along straight lane sections and smooth curved boundary segments, the robot followed the planned path closely without significant deviation. This is

consistent with the known behavior of the pure pursuit controller, which provides stable and smooth tracking along paths with gradually varying curvature when the lookahead distance is appropriately tuned relative to the path geometry [11].

One important distinction between the simulation results and expected physical performance is that the simulation provides ground-truth robot pose at every execution step. This eliminates localization error entirely during the mission execution stage. In a physical implementation using wheel odometry alone, positional drift would accumulate over the full mission distance of approximately 55 meters. Based on typical differential-drive odometry error rates of 1 to 3 percent of distance travelled, a drift of 0.5 to 1.5 meters could be expected by the end of the mission without correction. This level of drift would cause visible deviations from planned zig-zag lanes and could leave uncleaned strips between lanes. A LiDAR-based localization correction method such as scan matching against the pre-built occupancy map would be required to maintain acceptable path tracking accuracy in physical deployment.

Small tracking deviations were observed during sharp direction changes and transitions between cleaning segments. These deviations are expected behavior for the pure pursuit method and do not represent a failure of the controller. The speed reduction applied when the robot approached the end of each path segment helped reduce overshoot during segment transitions and contributed to smooth continuation of the mission. The goal tolerance of 0.12 m combined with the index-based endpoint detection condition ensured reliable segment switching without premature transitions.

The boundary exploration stage also performed as expected. The right-wall following strategy successfully guided the robot around the full perimeter of the GMIT environment, including the curved corridor sections and the boundaries around all five internal fixed column obstacles. One to two stuck events were observed near sharp boundary corners, but the stuck detection and recovery mechanism resolved these situations automatically in both cases without requiring external intervention. The three-condition exploration completion criterion, requiring minimum elapsed time, minimum travel distance, and proximity to the starting position, provided a reliable method for detecting boundary loop completion in the selected environment.

7.3 Map Processing and Path Planning Analysis

The MATLAB map processing pipeline successfully converted the raw occupancy map exported from CoppeliaSim into a clean and structured representation suitable for path planning. The erosion-dilation approach used in the noise removal and free region

extraction stage effectively removed small isolated free-space islands and narrow unreliable connections from the raw map, while preserving the main connected reachable floor area. The resulting cleaned free map provided a reliable basis for edge extraction and coverage planning.

The boundary following strategy produces a map in which interior free-space quality depends on the accuracy of long-range LiDAR measurements cast from boundary positions. In the simulation environment, LiDAR measurements are geometrically exact up to the grid discretization error, so this effect is limited. In physical deployment with a real LiDAR sensor, angular resolution limitations and surface reflectivity variations would introduce additional noise into long-range measurements, potentially degrading interior map quality beyond what the morphological processing pipeline can reliably correct. A supplementary interior traversal pass after the boundary exploration, or the use of a higher-resolution grid, could improve interior map quality in future implementations.

The safe navigation map generated by expanding obstacle regions by an inflation radius of 0.255 m ensured that transition paths between cleaning sections maintained adequate clearance from walls and obstacles. This inflation radius was derived from the physical robot width of 0.41 m plus a safety margin of 0.05 m, providing a conservative clearance for robot movement without excessively reducing the available navigable area. In narrow sections of the environment, the inflation noticeably reduced the available free space, but the main corridor and hall areas remained navigable throughout.

The edge polyline extraction algorithm successfully captured the full outer boundary of the GMIT environment and the boundaries around all five internal circular column obstacles. The extracted polylines closely followed the actual environment geometry, providing accurate boundary representations for edge-cleaning path generation. The path generation stages, including edge offset computation, zig-zag lane generation, and transition path planning using A* search through the safe navigation map, produced a complete and executable cleaning mission without manual adjustment.

7.4 Coverage Performance Analysis

The 99.4 percent final coverage result exceeds the 95 percent success threshold and confirms that the combined edge and zig-zag strategy is effective for the selected environment under simulation conditions. The remaining 0.6 percent of uncovered area is concentrated near sharp boundary corners and narrow obstacle-side regions where

the geometric path offset places the robot center too far from the wall for the coverage width to reach the adjacent boundary cells. This is a geometric limitation of offset-based edge cleaning rather than a path following failure. In the target physical platform, side brushes extending beyond the robot body would reduce this residual area by increasing the effective cleaning reach near walls. Quantitatively, reducing the edge offset from 0.3 meters to 0.25 meters in the path planning stage would bring the robot center closer to the boundary and could recover a portion of the uncovered corner cells, at the cost of a reduced safety margin from walls. This tradeoff should be evaluated during physical deployment when the actual wall clearance tolerance of the platform is known [1].

7.5 Proposed Hardware Deployment Discussion

The proposed hardware deployment architecture described in Section 3.3 outlines a viable path for transferring the simulation-validated control concept to a physical cleaning robot platform. The selected hardware combination of a repurposed Roborock cleaning robot and a Raspberry Pi 3B single-board computer provides a low-cost and practically accessible deployment target. The Roborock platform already includes a rotating 360-degree LiDAR assembly, differential-drive motors, a battery system, and drive electronics, all of which are compatible with the sensing and actuation requirements of the developed system.

The simulation workflow maps directly to the proposed physical architecture. The boundary exploration and occupancy mapping stage would run on the Raspberry Pi 3B during the first physical operation, generating a map of the target environment. The MATLAB map processing and path planning stages would be performed offline on an external computer, producing the cleaning mission CSV file. The mission execution stage, implemented as a Lua script in the simulation, would be re-implemented as a lightweight control program running on the Raspberry Pi 3B to send wheel velocity commands based on pure pursuit computations.

Hardware assembly is currently in progress. However, no physical validation results are available within the scope of this thesis. The hardware deployment section should therefore be understood as a proposed architecture and implementation roadmap rather than a validated deployment. Physical validation including LiDAR data pipeline testing, motor control integration, and real-environment coverage measurement is planned as the immediate next stage of this work following thesis submission [12].

7.6 System Limitations

Several limitations of the current system should be noted. First, the entire system was validated in simulation only. The simulation provides ground-truth robot pose during mission execution, which means localization error is completely absent from the results. In physical deployment, differential-drive odometry drift would accumulate over the full mission distance without correction. Additionally, real-world sensor noise, motor speed variation, and surface friction effects are not represented in the CoppeliaSim physics environment. A LiDAR-based position correction method integrated into the mission execution stage would be required before physical deployment. The performance of the path tracking controller and coverage quality in physical deployment may differ from the simulation results.

Second, the exploration and cleaning mission assume a static environment. Dynamic obstacles such as people walking through the environment during cleaning are not detected or avoided in the current system. Adding a reactive obstacle avoidance layer to the mission execution stage would be necessary for safe deployment in an actively used building.

Third, the rectangular body shape of the Pioneer P3DX robot model used in the simulation has a higher probability of contacting walls during turning maneuvers in narrow boundary areas compared with round-shaped commercial cleaning robots. This limitation contributed to the small residual uncovered areas near sharp corners observed in the final coverage result.

Fourth, no battery or power modeling is included in the current system. For physical deployment, the cleaning mission would need to be evaluated against the battery capacity of the Roborock platform, and a docking and recharging procedure would need to be incorporated for environments where a single battery charge is insufficient to complete the full mission.

Future work directions arising from the limitations identified above include: physical hardware validation of the control system on the proposed Roborock platform with real LiDAR and motor integration; addition of a LiDAR-based localization correction method to compensate for odometry drift during long missions; integration of a reactive dynamic obstacle avoidance layer for safe operation in occupied environments; extension of the system to multi-room environments through automatic room segmentation and doorway detection; and optimization of the mission path length to reduce total cleaning time while maintaining coverage above the defined threshold.

7.7 Comparison with Related Work

The system developed in this study addresses a problem that has been approached in various forms in the existing literature. Autonomous cleaning robots operating in indoor environments have been studied in the context of both commercial platforms and research-level systems, with most work focusing on either navigation and mapping or coverage path planning as separate problems [1], [2]. The contribution of this study lies in combining both components into a single deployable workflow that is validated in a realistic and geometrically complex environment.

In terms of mapping approach, the use of LiDAR-based occupancy grid mapping for indoor environments is a well-established method [3], [4]. However, most related implementations either assume a pre-built map or rely on a full SLAM system running continuously during mission execution. The system developed in this study takes a different approach by separating the exploration and execution stages entirely. The occupancy map is generated during a dedicated boundary following exploration run and then processed offline in MATLAB before any cleaning path is planned. This separation reduces the computational demand during mission execution and makes the approach more suitable for low-cost hardware platforms.

For coverage path planning, the literature consistently identifies combined boundary and interior coverage strategies as the most effective approach for structured indoor environments [8], [9]. The system developed in this study follows this same principle by combining edge-cleaning paths along all boundary regions with interior zig-zag coverage paths over the main floor area. The final coverage result of 99.4 percent demonstrates that this strategy is effective even in a non-rectangular environment with curved boundaries and internal fixed obstacles, which is a more demanding test case than the simple rectangular rooms commonly used in related work [8], [9].

For path tracking, the pure pursuit controller selected in this study is a simple and practical choice that has been validated in similar indoor navigation contexts [11]. Related work on autonomous cleaning robot deployment using ROS-based architectures confirms that the combination of differential-drive motion control and pre-planned waypoint execution is a viable approach for physical hardware platforms [12]. The proposed hardware deployment architecture described in this study is consistent with this direction, with the main difference being the use of a repurposed commercial cleaning platform rather than a purpose-built research robot.

Overall, the main distinction of this study from related work is the focus on a large and irregularly shaped real building environment rather than a simplified test space, and the use of an offline planning pipeline that does not require a full SLAM system during mission execution. These characteristics make the developed system a practical and accessible approach that is directly transferable to low-cost hardware, as described in the proposed deployment architecture.

8. Conclusion

This thesis presented the development and simulation-based validation of a control system for an autonomous cleaning robot designed for large indoor environments. The target environment was the main hall and corridor area of the GMIT first floor, which provided a realistic and challenging test case with curved boundaries, irregular shapes, and fixed internal obstacles.

The developed system follows a structured two-stage simulation framework. In the first stage, the Pioneer P3DX robot equipped with a 360-degree LiDAR assembly autonomously explored the reconstructed GMIT environment using right-wall following navigation and generated an occupancy grid map. In the second stage, the exported map was processed in MATLAB through noise removal, free-space extraction, safe map generation, and edge polyline extraction. A complete cleaning mission was then planned in MATLAB, combining boundary edge-cleaning paths, interior zig-zag coverage paths, safe transition routes, and a return-to-start path. The mission was executed in CoppeliaSim using a pure pursuit path tracking controller, with cleaning coverage tracked continuously throughout execution.

The system achieved a final cleaning coverage of 99.4 percent, exceeding the defined success threshold of 95 percent. The robot successfully followed both the edge-cleaning and interior zig-zag paths, completed all transition movements without contacting walls or obstacles, and returned to the starting position at mission completion. These results demonstrate that the combined edge and zig-zag coverage strategy, supported by the offline MATLAB processing pipeline and pure pursuit execution, is effective for large and irregularly shaped indoor environments.

In addition to the simulation results, a proposed hardware deployment architecture was described for transferring the developed control concept to a physical Roborock-based cleaning platform with a Raspberry Pi 3B controller. Hardware assembly is currently in

progress, and physical integration and validation are planned as the next stage of this work.

The results of this thesis establish a validated simulation framework and control architecture that provides a practical foundation for the subsequent hardware integration phase. The separation of exploration, planning, and execution into independent stages, combined with the lightweight pure pursuit controller, makes the system suitable for deployment on low-cost embedded hardware and supports the long-term goal of developing an accessible and effective autonomous cleaning solution for large public indoor spaces.

9. References

1. Nandan BJ, Koti V. A Comprehensive Review on Autonomous Mobile Robot Technologies for Indoor Cleaning Applications: Path Planning, SLAM, Obstacle Avoidance, Energy Optimization [Internet]. Scribd; 2025 [cited 2026 Apr 29]. Available from: https://www.scribd.com/document/944996136/AMR-Technologies-for-Indoor-Cleaning-Applications
2. Paneru B, Basnet N, Shrestha S, Giri R, Kshatri DB. Autonomous navigation of a mobile robot in indoor environments. Zerone Scholar. 2016 Nov;1(1):3-8. Available from: https://scispace.com/pdf/autonomous-navigation-of-a-mobile-robot-in-indoor-4pjet8g67y.pdf
3. Brindza J, Kajánek P, Erdélyi J. Lidar-Based Mobile Mapping System for an Indoor Environment. Slovak Journal of Civil Engineering [Internet]. 2022 Jun 1 [cited 2026 Apr 29];30(2):47–58. Available from: https://www.researchgate.net/publication/361936436_Lidar-Based_Mobile_Mapping_System_for_an_Indoor_Environment
4. Stefanini E, Enrico Ciancolini, Alessandro Settimi, Pallottino L. Efficient 2D LIDAR-Based Map Updating For Long-Term Operations in Dynamic Environments. 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 2022 Oct 23;832–9. Available from: https://www.researchgate.net/publication/362067800_Efficient_2D_LIDAR-Based_Map_Updating_For_Long-Term_Operations_in_Dynamic_Environments
5. Elfes A. Using occupancy grids for mobile robot perception and navigation. Computer. 1989 Jun;22(6):46-57. doi:10.1109/2.30720. Available from: https://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/integrated4/elfes_occup_grids.pdf

6. Thrun S, Burgard W, Fox D. Probabilistic robotics. Cambridge, MA: MIT Press; 2005.
7. MathWorks. Occupancy grids [Internet]. Natick, MA: The MathWorks, Inc.; [cited 2026 Apr 29]. Available from: https://www.mathworks.com/help/robotics/ug/occupancy-grids.html
8. Galceran E, Carreras M. A survey on coverage path planning for robotics. Robot Auton Syst. 2013 Dec;61(12):1258-76. doi:10.1016/j.robot.2013.09.004.
9. Jayalakshmi KP, Nair VG, Sathish D. A comprehensive survey on coverage path planning for mobile robots in dynamic environments. IEEE Access. 2025;13:60158-85. doi:10.1109/ACCESS.2025.3556446.
10. Adriansyah A, Suwoyo H, Tian Y, Deng C. Improving wall-following robot performance using PID-PSO controller. J Teknol. 2019;81(3):119-26. doi:10.11113/jt.v81.13098.
11. Morales J, Martínez JL, Martínez MA, Mandow A. Pure-pursuit reactive path tracking for nonholonomic mobile robots with a 2D laser scanner. EURASIP J Adv Signal Process. 2009;2009:935237. doi:10.1155/2009/935237
12. Roy A, Anandakrishnan S, Mohasin BR, Benny C, Shiras RM, Chakrayudhan M. ROS-SLAM powered autonomous vacuum cleaner. Int J Multidiscip Res. 2025;7(2). doi:10.36948/ijfmr.2025.v07i02.38351.
13. Coppelia Robotics. CoppeliaSim user manual [Internet]. Coppelia Robotics; [cited 2026 Apr 29]. Available from: https://manual.coppeliarobotics.com/
14. MathWorks. MATLAB documentation [Internet]. Natick, MA: The MathWorks, Inc.; [cited 2026 Apr 29]. Available from: https://www.mathworks.com/help/matlab/index.html
15. MathWorks. Image Processing Toolbox documentation [Internet]. Natick, MA: The MathWorks, Inc.; [cited 2026 Apr 29]. Available from: https://www.mathworks.com/help/images/index.html

Appendices

Appendix A: CoppeliaSim Navigation Simulation Code

A.1 — State Machine and Main Parameters

```
STATE_EXPLORE = 1
STATE_FINISHED = 2
currentState = STATE_EXPLORE

forwardSpeed = 0.9
turnSpeed = 0.65

desiredRightDist = 0.5
frontBlockDist = 0.6
rightWallDetectDist = 1.2

maxScanDistance = 15
mapResolution = 0.05
mapUpdateInterval = 0.25
mapRayStep = 0.05

minExploreTime = 45
minTravelDistance = 20
returnDistThreshold = 0.8
returnYawThreshold = 1.2
```

A.2 — LiDAR Data Reading

```
function readLidarPoints()
	local packed = sim.getStringSignal('lidarData')
	if not packed then
		return nil
	end

	local data = sim.unpackFloatTable(packed)
	if not data or #data < 3 then
		return nil
	end

	local points = {}
	for i = 1, #data, 3 do
		local x = data[i]
		local y = data[i + 1]
		local z = data[i + 2]

		if x ~= nil and y ~= nil and z ~= nil then
			points[#points + 1] = {x = x, y = y, z = z}
		end
	end

	return points
end
```

A.3 — Front and Right Wall Distance Extraction

```
function getFrontMinDistance(points)
```

```
	local minDist = maxScanDistance
	local found = false

	for i = 1, #points do
		local p = points[i]
		if p.x > 0 and math.abs(p.y) < frontWidth then
			local d = math.sqrt(p.x * p.x + p.y * p.y)
			if d < minDist then
				minDist = d
				found = true
			end
		end
	end

	return minDist, found
end

function getRightWallDistanceRaw(points)
	local minDist = maxScanDistance
	local found = false

	for i = 1, #points do
		local p = points[i]
		if p.y < 0 and p.x > rightXMin and p.x < rightXMax then
			local lateralDist = math.abs(p.y)
			if lateralDist < minDist then
				minDist = lateralDist
				found = true
			end
		end
	end

		return minDist, found
end
```

A.4 — Right-Wall Following Control Logic

```
function boundaryFollowStep()
	local t = sim.getSimulationTime()

	updateStuckStatus()

	if recoveryMode == 1 then
		if t < recoveryEndTime then
			setWheelSpeeds(-0.4, 0.6)
			return
		else
			recoveryMode = 0
		end
	end

	if turnMode == 1 then
		if t < turnModeEndTime then
			turnLeft(turnSpeed)
			return
		else
			turnMode = 0
		end
	end
```

```
        local frontBlocked, frontMin = isFrontBlocked()
        local rightDetected, rightRaw = isRightWallDetected()

        if frontBlocked then
                turnMode = 1
                turnModeEndTime = t + 0.8
                turnLeft(turnSpeed)
                return
        end

        if not rightDetected then
                setWheelSpeeds(0.85, -0.55)
                return
        end

        local rightDist, found = getWallDistance()

        if found then
                local error = rightDist - desiredRightDist

                if math.abs(error) < 0.05 then
                        moveForward(forwardSpeed)
                        return
                end

                local correction = 1.2 * error
                correction = clamp(correction, -0.22, 0.22)

                setWheelSpeeds(forwardSpeed + correction, forwardSpeed -
                correction)
        else
                setWheelSpeeds(forwardSpeed + 0.10, forwardSpeed - 0.10)
        end
end
```

A.5 — Exploration Completion Condition

```
function checkBoundaryLoopCompletion()
        if boundaryLoopCompleted then
                return true
        end

        local currentTime = sim.getSimulationTime()
        local elapsed = currentTime - exploreStartTime
        local backNearStart, distToStart, yawError = isBackNearStart()

        if elapsed > minExploreTime and travelDistance > minTravelDistance and
        backNearStart and not isStuckNow then
                boundaryLoopCompleted = true
                return true
        end

        return false
end
```

A.6 — Occupancy Grid Map Update

```
function updateMap()
```

```
        local rx, ry, ryaw = getPose()
        local points = readLidarPoints()
        if not points then
                return
        end

        markFree(rx, ry)

        local c = math.cos(ryaw)
        local s = math.sin(ryaw)

        for i = 1, #points, mapPointStride do
                local p = points[i]
                local wx = rx + p.x * c - p.y * s
                local wy = ry + p.x * s + p.y * c

                markLineFree(rx, ry, wx, wy)
                markObstacle(wx, wy)
        end
end
```

A.7 — CSV Map Export

```
function exportMapToCsv(basePath)
        local gxMin, gxMax, gyMin, gyMax = getDisplayGridBounds()

        local csvPath = basePath .. '/occupancy_mapgmit.csv'
        local metaPath = basePath .. '/occupancy_metagmit.txt'

        local f = io.open(csvPath, 'w')
        if not f then
                print('Failed to open CSV file: ' .. csvPath)
                return
        end

        for gy = gyMax, gyMin, -1 do
                local row = {}
                for gx = gxMin, gxMax do
                        local key = makeGridKey(gx, gy)
                        local v = -1

                        if mapData[key] == 1 then
                                v = 1
                        elseif mapData[key] == 0 then
                                v = 0
                        end

                        row[#row + 1] = tostring(v)
                end
                f:write(table.concat(row, ',') .. '\n')
        end
        f:close()

        local m = io.open(metaPath, 'w')
        m:write('mapResolution=' .. tostring(mapResolution) .. '\n')
        m:write('gxMin=' .. tostring(gxMin) .. '\n')
        m:write('gxMax=' .. tostring(gxMax) .. '\n')
        m:write('gyMin=' .. tostring(gyMin) .. '\n')
        m:write('gyMax=' .. tostring(gyMax) .. '\n')
```

```
        m:close()
end
```

A.8 — Main Simulation Callbacks

```
function sysCall_actuation()
        updateState()
        runCurrentState()
end

function sysCall_sensing()
        local t = sim.getSimulationTime()
        if t - lastMapUpdateTime >= mapUpdateInterval then
                updateMap()
                lastMapUpdateTime = t
        end
end

function sysCall_cleanup()
        stopRobot()
end
```

Appendix B: Occupancy Map Processing in MATLAB

B.1 Occupancy Map Loading and Cell Classification

```
M = readmatrix(INPUT_CSV);
meta = readMetaFile(INPUT_META);

mapResolution = meta.mapResolution;
gxMin = meta.gxMin;
gxMax = meta.gxMax;
gyMin = meta.gyMin;
gyMax = meta.gyMax;

[rows, cols] = size(M);

free_map_raw = (M == 0);
obstacle_map_raw = (M == 1);
unknown_map = (M == -1);
known_map = ~unknown_map;
```

B.2 Free-Space Cleaning and Main Region Extraction

```
free0 = free_map_raw & known_map;
free0 = bwareaopen(free0, smallFreeAreaCells, 4);

seNarrow = strel('disk', narrowWidthCells, 0);
free_eroded = imerode(free0, seNarrow);

CC = bwconncomp(free_eroded, 4);
if CC.NumObjects == 0
        error('No free region left after erosion.');
end

[~, idxLargest] = max(cellfun(@numel, CC.PixelIdxList));
main_eroded = false(size(free_eroded));
main_eroded(CC.PixelIdxList{idxLargest}) = true;
```

```
main_recovered = imdilate(main_eroded, seNarrow);
main_recovered = main_recovered & free0;

if closeRadiusCells > 0
       seSmall = strel('disk', closeRadiusCells, 0);
       main_recovered = imclose(main_recovered, seSmall);
       main_recovered = imopen(main_recovered, seSmall);
       main_recovered = main_recovered & free0;
end

CC2 = bwconncomp(main_recovered, 4);
if CC2.NumObjects == 0
   error('No free region after reconstruction.');
end

[~, idxLargest2] = max(cellfun(@numel, CC2.PixelIdxList));
main_free_map = false(size(main_recovered));
main_free_map(CC2.PixelIdxList{idxLargest2}) = true;

obstacle_map = known_map & ~main_free_map;
```

B.3 Safe Navigation Map Generation

```
seInflate = strel('disk', inflationCells, 0);
nav_obstacle_map = imdilate(obstacle_map, seInflate);
nav_free_map = known_map & ~nav_obstacle_map & main_free_map;

CC3 = bwconncomp(nav_free_map, 4);
if CC3.NumObjects == 0
       nav_free_map = main_free_map;
       nav_obstacle_map = known_map & ~nav_free_map;
else
       [~, idxLargest3] = max(cellfun(@numel, CC3.PixelIdxList));
       nav_main = false(size(nav_free_map));
       nav_main(CC3.PixelIdxList{idxLargest3}) = true;
       nav_free_map = nav_main;
       nav_obstacle_map = known_map & ~nav_free_map;
end
```

B.4 Edge Segment Extraction

```
maxSeg = rows * cols * 4;
segments = zeros(maxSeg, 4);
segCount = 0;

for r = 1:rows
       for c = 1:cols
              if ~main_free_map(r,c)
                     continue;
              end

              [xL,  xR,  yB,  yT]  =  cellEdgesMeters(r,  c,  gxMin,  gyMax,
              mapResolution);

              if r == 1 || ~main_free_map(r-1, c)
                     segCount = segCount + 1;
                     segments(segCount,:) = [xL, yT, xR, yT];
              end
```

```
            if r == rows || ~main_free_map(r+1, c)
                segCount = segCount + 1;
                segments(segCount,:) = [xL, yB, xR, yB];
            end

            if c == 1 || ~main_free_map(r, c-1)
                segCount = segCount + 1;
                segments(segCount,:) = [xL, yB, xL, yT];
            end

            if c == cols || ~main_free_map(r, c+1)
                segCount = segCount + 1;
                segments(segCount,:) = [xR, yB, xR, yT];
            end
        end
end

segments = segments(1:segCount,:);
```

B.5 Polyline Connection and Simplification

```
polylines = connectSegmentsToPolylines(segments, 1e-9);

cleaned_polylines = {};
    for k = 1:numel(polylines)
    P = polylines{k};

    if size(P,1) < minPoints
        continue;
    end

    P = removeDuplicateConsecutivePoints(P);
    P = simplifyRDP(P, epsilonRDP);
    P = simplifyPolyline(P, angleTol);
    P = removeDuplicateConsecutivePoints(P);

    if polylineLength(P) < minLengthM
        continue;
    end

    if size(P,1) >= 2
        cleaned_polylines{end+1} = P;
    end
end

polylines = cleaned_polylines;
```

B.6 Processed Map and Edge Export

```
exportRows = [];
for k = 1:numel(polylines)
    P = polylines{k};
    ids = k * ones(size(P,1), 1);
    exportRows = [exportRows; [ids, P]];
end

writematrix(exportRows,fullfile(OUTPUT_DIR,'exact_edge_map_meters_gmit.csv');
```

```
save(fullfile(OUTPUT_DIR, 'edge_polylines_gmit.mat'), ...
        'polylines', ...
        'main_free_map', 'free_map_raw', ...
        'obstacle_map', 'obstacle_map_raw', ...
        'unknown_map', ...
        'nav_free_map', 'nav_obstacle_map', ...
        'robotWidthM', 'robotRadiusM', 'safetyMarginM', ...
        'inflationRadiusM', 'inflationCells', ...
        'mapResolution', 'gxMin', 'gxMax', 'gyMin', 'gyMax');

coverage_cleanable_map = double(main_free_map);
writematrix(coverage_cleanable_map,fullfile(OUTPUT_DIR,'coverage_cleanable_ma
p_gmit.csv'));

coverage_meta = table(mapResolution, gxMin, gxMax, gyMin, gyMax, rows, cols,
...
'VariableNames',{'mapResolution','gxMin','gxMax','gyMin','gyMax','rows','cols
'});

writetable(coverage_meta, fullfile(OUTPUT_DIR, 'coverage_meta_gmit.csv'));
```

Appendix C: Edge and Zig-Zag Path Generation Code

C.1 Edge-Cleaning Path Offset and Validation

```
for k = 1:numel(polylines)
        P = polylines{k};
        P = sanitizePolyline(P);

        if size(P,1) < 2
                continue;
        end

        isClosed = norm(P(1,:) - P(end,:)) < 1e-9;

        P = removeShortSegments(P, minSegLen, isClosed);
        P = simplifyRDP_general(P, rdpTolerance, isClosed);
        P = removeSharpSpikes(P, spikeAngleDeg, isClosed);
        P = resamplePolyline(P, preSampleStep, isClosed);

        P_left = offsetPolylineGeneral(P, +edgeOffset, isClosed);
        P_right = offsetPolylineGeneral(P, -edgeOffset, isClosed);

        segsL = extractValidSegments(P_left, main_free_map, gxMin, gyMax, ...
                mapResolution, minPoints, minPathLength, minSegLen,
                maxGapBridge);

        segsR = extractValidSegments(P_right, main_free_map, gxMin, gyMax, ...
                mapResolution, minPoints, minPathLength, minSegLen,
                maxGapBridge);

        scoreL = totalSegmentScore(segsL);
        scoreR = totalSegmentScore(segsR);

        if scoreL >= scoreR
                chosenSegs = segsL;
                chosenSide = +1;
        else
```

```
            chosenSegs = segsR;
            chosenSide = -1;
        end
end
```

C.2 Edge Route Cleaning, Resampling, and Export

```
for s = 1:numel(chosenSegs)
        Q = chosenSegs{s};

        Q = sanitizePolyline(Q);
        Q = removeShortSegments(Q, minSegLen, false);
        Q = resamplePolyline(Q, sampleStep, false);

        if smoothWindow > 1
                Q = smoothOpenPolyline(Q, smoothWindow);
        end

        Q = simplifyRDP_general(Q, finalRdpTolerance, false);
        Q = removeSharpSpikes(Q, spikeAngleDeg, false);
        Q = removeShortSegments(Q, minSegLen, false);
        Q = sanitizePolyline(Q);

        if size(Q,1) < minPoints
                continue;
        end

        validMask = validPointMask(Q, main_free_map, gxMin, gyMax,
        mapResolution, occupancyMargin);

        validRatio = nnz(validMask) / numel(validMask);
        if validRatio < 0.90
                continue;
        end

        L = polylineLength(Q);
        if L < minPathLength
                continue;
        end

        edgePaths{end+1} = Q;
        pathMeta = [pathMeta; [k, L, size(Q,1), chosenSide]];
end

exportRows = [];
for i = 1:numel(edgePaths)
        P = edgePaths{i};
        ids = i * ones(size(P,1),1);
        exportRows = [exportRows; [ids, P]];
end

writematrix(exportRows, pathCsvFile);
```

C.3 Edge Mission Ordering and Safe Transfer Generation

```
orderedRoutes   = {};
orderedRouteIds = [];
used = false(numel(edgePaths_clean),1);
```

```
if useRobotStart
	currentXY = robotStartXY;
else
	lengths = zeros(numel(edgePaths_clean),1);
	for i = 1:numel(edgePaths_clean)
		lengths(i) = polylineLength(edgePaths_clean{i});
	end

	[~, firstIdx] = max(lengths);
	firstRoute = edgePaths_clean{firstIdx};

	orderedRoutes{1} = firstRoute;
	orderedRouteIds(1) = firstIdx;
	used(firstIdx) = true;
	currentXY = firstRoute(end,:);
end

while nnz(used) < numel(edgePaths_clean)
	bestIdx = -1;
	bestDist = inf;
	bestReverse = false;

	for i = 1:numel(edgePaths_clean)
		if used(i)
			continue;
		end

		P = edgePaths_clean{i};

		d1 = norm(currentXY - P(1,:));
		d2 = norm(currentXY - P(end,:));

		if d1 < bestDist
			bestDist = d1;
			bestIdx = i;
			bestReverse = false;
		end

		if d2 < bestDist
			bestDist = d2;
			bestIdx = i;
			bestReverse = true;
		end
	end

	P = edgePaths_clean{bestIdx};
	if bestReverse
		P = flipud(P);
	end

	orderedRoutes{end+1} = P;
	orderedRouteIds(end+1) = bestIdx;
	used(bestIdx) = true;
	currentXY = P(end,:);
end

for i = 1:numel(orderedRoutes)
	cleanPath = orderedRoutes{i};
	cleanRouteId = orderedRouteIds(i);
```

```
        missionId = missionId + 1;

        missionRows = [missionRows; ...
                [missionId * ones(size(cleanPath,1),1),
                1 * ones(size(cleanPath,1),1),
                cleanRouteId * ones(size(cleanPath,1),1),
                cleanPath]];

        if i < numel(orderedRoutes)
                nextPath = orderedRoutes{i+1};
                nextRouteId = orderedRouteIds(i+1);

                routeEndXY = cleanPath(end,:);
                routeStartXY = nextPath(1,:);

                [safeFromXY, okFrom] = snapWorldToNearestSafe(routeEndXY,
                nav_free_map, nav_dist, minClearance, gxMin, gyMax,
                mapResolution);

                [safeToXY, okTo] = snapWorldToNearestSafe(routeStartXY,
                nav_free_map, nav_dist, minClearance, gxMin, gyMax,
                mapResolution);

                if okFrom
                        conn1 = sanitizePolyline([routeEndXY; safeFromXY]);
                else
                        conn1 = [];
                end

                if okFrom && okTo
                        transferPath = astarWorld(safeFromXY, safeToXY,
                        nav_free_map, gxMin, gyMax, mapResolution, connType);

                        if ~isempty(transferPath)
                                transferPath = sanitizePolyline(transferPath);
                                transferPath = resamplePolyline(transferPath,
                                transferSampleStep, false);
                        end
                else
                        transferPath = [];
                end

                if okTo
                        conn2 = sanitizePolyline([safeToXY; routeStartXY]);
                else
                        conn2 = [];
                end
        end
end
```

C.4 Interior Sweep Lane Generation

```
if strcmpi(sweepDir,'x')
        laneYs = (yBBmin + laneSpacingM/2) : laneSpacingM : yBBmax;

        for li = 1:numel(laneYs)
                laneY = laneYs(li);
                laneRow = yToRow(laneY, gyMax, mapResolution);
```

```
            if laneRow < 1 || laneRow > mapRows
                continue;
            end

            segs = contiguousTrueRuns(nav_free_map(laneRow,:));

            for s = 1:size(segs,1)
                x1 = colToX(segs(s,1), gxMin, mapResolution);
                x2 = colToX(segs(s,2), gxMin, mapResolution);

                nPts = max(2, round((x2 - x1) / sweepSampleStep) + 1);
                lane = [linspace(x1, x2, nPts)', laneY * ones(nPts,1)];

                lane = filterLanePoints(lane, nav_free_map,
                gxMin, gyMax, mapResolution);

                if size(lane,1) >= minLanePoints
                    sweepLanes{end+1} = lane;
                end
            end
        end
end
```

C.5 Interior Mission Generation with Safe Transfers

```
nL = numel(sweepLanes);
laneCenters = zeros(nL,2);

for i = 1:nL
    laneCenters(i,:) = mean(sweepLanes{i},1);
end
labels = zeros(nL,1);
clusterId = 0;

for i = 1:nL
    if labels(i) ~= 0
        continue;
    end

    clusterId = clusterId + 1;
    labels(i) = clusterId;
    queue = i;

    while ~isempty(queue)
        cur = queue(1);
        queue(1) = [];

        for j = 1:nL
            if labels(j) == 0
                d = norm(laneCenters(cur,:) - laneCenters(j,:));
                if d < clusterDistM
                    labels(j) = clusterId;
                    queue(end+1) = j;
                end
            end
        end
    end
end
```

```
regionLaneLists = cell(clusterId,1);
regionStartXY = zeros(clusterId,2);
regionEndXY = zeros(clusterId,2);

for cid = 1:clusterId
	idx = find(labels == cid);
	regionUsed = false(numel(idx),1);
	regionOrder = [];

	regionCenters = laneCenters(idx,:);
	[~, localStartIdx] = min(regionCenters(:,2) + 1e-3 *
	regionCenters(:,1));
	cur = idx(localStartIdx);

	while numel(regionOrder) < numel(idx)
		regionOrder(end+1) = cur;

		localIdx = find(idx == cur, 1, 'first');
		regionUsed(localIdx) = true;
		curEnd = sweepLanes{cur}(end,:);

		bestJ = -1;
		bestDist = inf;
		bestFlip = false;

		for k = 1:numel(idx)
			if regionUsed(k)
				continue;
			end

			j = idx(k);
			P = sweepLanes{j};

			d1 = norm(curEnd - P(1,:));
			d2 = norm(curEnd - P(end,:));

			if d1 < bestDist
				bestDist = d1;
				bestJ = j;
				bestFlip = false;
			end

			if d2 < bestDist
				bestDist = d2;
				bestJ = j;
				bestFlip = true;
			end
		end

		if bestJ == -1
			break;
		end

		if bestFlip
			sweepLanes{bestJ} = flipud(sweepLanes{bestJ});
		end

		cur = bestJ;
	end
```

```
        regionOrdered = sweepLanes(regionOrder);
        regionLaneLists{cid} = regionOrdered;
        regionStartXY(cid,:) = regionOrdered{1}(1,:);
        regionEndXY(cid,:) = regionOrdered{end}(end,:);
end

orderedRegions = [];
usedRegion = false(clusterId,1);

[~, firstRegion] = min(regionStartXY(:,2) + 1e-3 * regionStartXY(:,1));
orderedRegions(end+1) = firstRegion;
usedRegion(firstRegion) = true;

while nnz(usedRegion) < clusterId
        curRegion = orderedRegions(end);
        curEnd = regionEndXY(curRegion,:);

        bestR = -1;
        bestDist = inf;
        bestFlipRegion = false;

        for r = 1:clusterId
                if usedRegion(r)
                        continue;
                end

                d1 = norm(curEnd - regionStartXY(r,:));
                d2 = norm(curEnd - regionEndXY(r,:));

                if d1 < bestDist
                        bestDist = d1;
                        bestR = r;
                        bestFlipRegion = false;
                end

                if d2 < bestDist
                        bestDist = d2;
                        bestR = r;
                        bestFlipRegion = true;
                end
        end

        if bestFlipRegion
                regionLaneLists{bestR} = flip(regionLaneLists{bestR});
                for k = 1:numel(regionLaneLists{bestR})
                        regionLaneLists{bestR}{k} =
                        flipud(regionLaneLists{bestR}{k});
                end
        end
        orderedRegions(end+1) = bestR;
        usedRegion(bestR) = true;
end

finalLanes = {};
for k = 1:numel(orderedRegions)
        rid = orderedRegions(k);
        finalLanes = [finalLanes, regionLaneLists{rid}];
end
```

```
sweepLanes = finalLanes;

missionRows = [];
missionId = 0;

for i = 1:numel(sweepLanes)
	lane = sweepLanes{i};

	missionId = missionId + 1;
	missionRows = [missionRows; buildRows(missionId, 1, i, lane)];

	if i == numel(sweepLanes)
		break;
	end

	nextLane = sweepLanes{i+1};
	laneEndXY = lane(end,:);
	laneStartXY = nextLane(1,:);

	[safeFromXY, okFrom] = snapSafe(laneEndXY, nav_free_map, nav_dist,
	minClearanceM, gxMin, gyMax, mapResolution);

	[safeToXY, okTo] = snapSafe(laneStartXY, nav_free_map, nav_dist,
	minClearanceM, gxMin, gyMax, mapResolution);

	if okFrom && okTo
		tp = fastPath(safeFromXY, safeToXY, plan_free, G,
		planRows, planCols, planDS, mapRows, mapCols,
		mapResolution, gxMin, gyMax);

		tp = resample(transferPath, transferSampleStep);

		if size(tp,1) >= 2
			missionId = missionId + 1;
			missionRows = [missionRows; buildRows(missionId, 3,
			i+1,tp)];
		end
	end
end

writematrix(missionRows, outputCsvFile);
```

C.6 Final Mission Combination, Return-to-Start, and Export

```
edgeData = readmatrix(edgeCsvFile);
intData = readmatrix(interiorCsvFile);

edgeMissionMax = max(edgeData(:,1));
edgeRouteMax = max(edgeData(:,3));

intDataShifted = intData;
intDataShifted(:,3) = intDataShifted(:,3) + edgeRouteMax;
intDataShifted(:,1) = intDataShifted(:,1) + edgeMissionMax + 1;

edgeLastPoint = edgeData(end,4:5);
intFirstPoint = intDataShifted(1,4:5);

[safeFromXY, okFrom] = snapSafe(edgeLastPoint, nav_free_map, nav_dist,
minClearanceM, gxMin, gyMax, mapResolution);
```

```
[safeToXY, okTo] = snapSafe(intFirstPoint, nav_free_map, nav_dist,
minClearanceM, gxMin, gyMax, mapResolution);

connectorPath = fastPath(safeFromXY, safeToXY, plan_free, G,
planRows, planCols, planDS, mapRows, mapCols,
mapResolution, gxMin, gyMax);

connectorPath = sanitizePoly([edgeLastPoint; connectorPath; intFirstPoint]);
connectorPath = resample(connectorPath, connectorSampleStep);

connectorSeg = [
       (edgeMissionMax + 1) * ones(size(connectorPath,1),1), ...
       3 * ones(size(connectorPath,1),1), ...
       (edgeRouteMax + 1) * ones(size(connectorPath,1),1), ...
       connectorPath
];

edgeOrigin = ones(size(edgeData,1),1);
connectorOrigin = 3 * ones(size(connectorSeg,1),1);
intOrigin = 2 * ones(size(intDataShifted,1),1);

edgeData6 = [edgeData(:,1:3), edgeOrigin, edgeData(:,4:5)];
connectorSeg6 = [connectorSeg(:,1:3), connectorOrigin, connectorSeg(:,4:5)];
intData6 = [intDataShifted(:,1:3), intOrigin, intDataShifted(:,4:5)];

finalData = [
       edgeData6;
       connectorSeg6;
       intData6
];

if returnToStart
       finalEndPoint = finalData(end,5:6);

       if strcmpi(returnMode, 'missionStart')
              returnGoalPoint = finalData(1,5:6);
       elseif strcmpi(returnMode, 'custom')
              returnGoalPoint = returnTargetXY;
       end

       returnMissionId = max(finalData(:,1)) + 1;
       returnRouteId   = max(finalData(:,3)) + 1;

       [safeReturnFromXY, okR1] = snapSafe(finalEndPoint, nav_free_map,
       nav_dist, minClearanceM, gxMin, gyMax, mapResolution);
       [safeReturnToXY, okR2] = snapSafe(returnGoalPoint, nav_free_map,
       nav_dist,minClearanceM, gxMin, gyMax, mapResolution);

       returnPath = fastPath(safeReturnFromXY, safeReturnToXY,
              plan_free, G, planRows, planCols, planDS,
              mapRows, mapCols, mapResolution, gxMin, gyMax);

       returnPath = sanitizePoly([finalEndPoint; returnPath;
       returnGoalPoint]);
       returnPath = resample(returnPath, connectorSampleStep);
       returnOrigin = 4 * ones(size(returnPath,1),1);
```

```
        returnSeg6 = [
                returnMissionId*ones(size(returnPath,1),1),
                3*ones(size(returnPath,1),1),
                returnRouteId*ones(size(returnPath,1),1),
                returnOrigin,
                returnPath
        ];
        finalData = [
                finalData;
                returnSeg6
        ];
end

writematrix(finalData, outputCsvFile);
```

Appendix D: CoppeliaSim Cleaning Simulation Code

D.1 Mission Execution Parameters

```
csvFile = 'C:/Users/Kirada/Documents/final_cleaning_mission_gmit_safe.csv'
mapPath = 'C:/Users/Kirada/Documents/coverage_cleanable_map_gmit.csv'
metaPath = 'C:/Users/Kirada/Documents/coverage_meta_gmit.csv'
robotLength = 0.51
robotWidth = 0.41
robotHeight = 0.19

wheelRadius = 0.0975
wheelBase = 0.331

minSpeed = 0.08
maxLinearSpeed = 0.90
maxOmega = 3.8

lookAhead = 0.50
headingGain = 1.8
goalTolerance = 0.12
slowDownRadius = 0.15

minPointSpacing = 0.02
resampleStep = 0.05

coverageWidth = 0.5
interpSteps = 5

worldOffsetX = 0.0
worldOffsetY = 0.0
worldYawDeg = 0.0
flipY = false
```

D.2 Mission CSV Loading and Path Grouping

```
function loadMissionCsv(path)
        local f = io.open(path, 'r')
        if not f then
                return nil
        end

        local rows = {}
```

```
        for line in f:lines() do
                line = trim(line)
                if line ~= '' then
                        local cols = splitCsvSimple(line)

                        if #cols >= 6 then
                                local mission_id = tonumber(cols[1])
                                local segment_type = tonumber(cols[2])
                                local route_id = tonumber(cols[3])
                                local mission_origin = tonumber(cols[4])
                                local x_m = tonumber(cols[5])
                                local y_m = tonumber(cols[6])

                                if mission_id and segment_type and route_id and
                                mission_origin and x_m and y_m then

                                        rows[#rows + 1] = {
                                                mission_id = mission_id,
                                                segment_type = segment_type,
                                                route_id = route_id,
                                                mission_origin = mission_origin,
                                                x = x_m,
                                                y = y_m
                                        }
                                end
                        end
                end
        end

        f:close()
        return rows
end

function groupRowsIntoPaths(rows)
        local grouped = {}
        local orderedIds = {}

        for i = 1, #rows do
                local r = rows[i]
                local id = r.mission_id

                if grouped[id] == nil then
                        grouped[id] = {
                                mission_id = id,
                                segment_type = r.segment_type,
                                route_id = r.route_id,
                                mission_origin = r.mission_origin,
                                points = {}
                        }
                        orderedIds[#orderedIds + 1] = id
                end

                grouped[id].points[#grouped[id].points + 1] = {r.x, r.y}
        end

        table.sort(orderedIds)
```

```
        local out = {}
        for i = 1, #orderedIds do
                out[#out + 1] = grouped[orderedIds[i]]
        end

        return out
end
```

D.3 Path Classification and Speed Selection

```
function classifyMissionPath(missionOrigin, segmentType)
        if missionOrigin == 1 then
                if segmentType == 1 then
                        return 1
                else
                        return 2
                end
        elseif missionOrigin == 2 then
                if segmentType == 1 then
                        return 3
                else
                        return 4
                end
        elseif missionOrigin == 3 then
                return 5
        elseif missionOrigin == 4 then
                return 6
        else
                if segmentType == 1 then
                        return 3
                else
                        return 4
                end
        end
end

function getPathSpeed(pathObj)
        if pathObj.drawClass == 5 or pathObj.drawClass == 6 then
                return 0.28
        end

        if pathObj.segment_type == 1 then
                if pathObj.mission_origin == 1 then
                        return 0.35
                elseif pathObj.mission_origin == 2 then
                        return 0.40
                else
                        return 0.32
                end
        else
                return 0.28
        end
end
```

D.4 Pure Pursuit Lookahead Point Selection

```
function findLookAheadPoint(path, rx, ry, lookDist)
        local closestIdx = 1
        local bestD2 = math.huge
```

```
        for i = 1, #path do
                local dx = path[i][1] - rx
                local dy = path[i][2] - ry
                local d2 = dx * dx + dy * dy
                if d2 < bestD2 then
                        bestD2 = d2
                        closestIdx = i
                end
        end

        if closestIdx >= #path - 1 then
                local distToEnd = dist2Dxy(rx, ry, path[#path][1],
                path[#path][2])
                return path[#path], closestIdx, distToEnd
        end

        local accum = 0.0

        for i = closestIdx, #path - 1 do
                local p1 = path[i]
                local p2 = path[i + 1]
                local segLen = dist2D(p1, p2)

                if accum + segLen >= lookDist then
                        local remain = lookDist - accum
                        local t = 0.0
                        if segLen > 1e-9 then
                                t = remain / segLen
                        end

                        local lx = p1[1] + t * (p2[1] - p1[1])
                        local ly = p1[2] + t * (p2[2] - p1[2])

                        local distToEnd = distanceAlongPathFrom(path, i + t)
                        return {lx, ly}, closestIdx, distToEnd
                end

                accum = accum + segLen
        end

        local endPt = path[#path]
        local distToEnd = dist2Dxy(rx, ry, endPt[1], endPt[2])
        return endPt, closestIdx, distToEnd
end
```

D.5 Pure Pursuit Control and Wheel Velocity Conversion

```
function purePursuitControl(rx, ry, rtheta, tx, ty, gx, gy, distToEnd,
targetSpeed, headingGain, maxOmega)

        local dx = tx - rx
        local dy = ty - ry

        local c = math.cos(rtheta)
        local s = math.sin(rtheta)

        local localX = c * dx + s * dy
        local localY = -s * dx + c * dy
```

```
        local Ld = math.sqrt(localX * localX + localY * localY)
        if Ld < 1e-6 then
                return 0, 0
        end

        local kappa = 2.0 * localY / (Ld * Ld)
        local v = targetSpeed

        if distToEnd < slowDownRadius then
                local a = clamp(distToEnd / slowDownRadius, 0.0, 1.0)
                v = minSpeed + a * (targetSpeed - minSpeed)
        end

        v = clamp(v, minSpeed, maxLinearSpeed)

        local goalHeading = math.atan2(ty - ry, tx - rx)
        local headingErr = wrapToPi(goalHeading - rtheta)

        local w = v * kappa + headingGain * headingErr
        w = clamp(w, -maxOmega, maxOmega)

        return v, w
end

function setBaseVelocity(v, omega)
        local vL = v - 0.5 * wheelBase * omega
        local vR = v + 0.5 * wheelBase * omega

        local wL = vL / wheelRadius
        local wR = vR / wheelRadius

        setWheelSpeeds(wL, wR)
end

function setWheelSpeeds(leftVel, rightVel)
        sim.setJointTargetVelocity(leftMotor, leftVel)
        sim.setJointTargetVelocity(rightMotor, rightVel)
end
```

D.6 Main Mission Execution Callback

```
function sysCall_actuation()
        if missionFinished then
                currentDisplayMode = 'finished'
                setWheelSpeeds(0, 0)
                updateUiLabels()
                return
        end

        if activePathIdx > #paths then
                missionFinished = true
                currentDisplayMode = 'finished'
                setWheelSpeeds(0, 0)
                updateUiLabels()
                return
        end
```

```
    local currentPath = paths[activePathIdx]
    local P = currentPath.points

    if not P or #P < 2 then
        goToNextPath()
        updateUiLabels()
        return
    end

    local rx, ry, rtheta = getRobotPose2D()

    local cfg = {
        speed = getPathSpeed(currentPath),
        lookAhead = lookAhead,
        headingGain = headingGain,
        maxOmega = maxOmega
    }

    currentDisplayMode = className(currentPath.drawClass)

    local lookPt, closestIdx, distToEnd = findLookAheadPoint(P, rx, ry,
    cfg.lookAhead)

    if showLookAheadPoint and lookAheadDraw then
        sim.addDrawingObjectItem(lookAheadDraw, nil)
        sim.addDrawingObjectItem(lookAheadDraw, {lookPt[1], lookPt[2],
        pathDrawZ + 0.02})
    end

    local endPt = P[#P]
    local distToGoal = dist2Dxy(rx, ry, endPt[1], endPt[2])
    local nearEndIndex = (#P - closestIdx) <= 6

    if nearEndIndex and distToGoal < goalTolerance then
        goToNextPath()
        updateUiLabels()
        return
    end

    local v, w = purePursuitControl(
        rx, ry, rtheta,
        lookPt[1], lookPt[2],
        endPt[1], endPt[2],
        distToEnd,
        cfg.speed, cfg.headingGain, cfg.maxOmega
    )

    setBaseVelocity(v, w)
    updateUiLabels()
end
```

D.7 Coverage Tracking

```
function markCoveredAroundRobot(robotX, robotY)
    local half = coverageWidth / 2.0

    local minX = robotX - half
    local maxX = robotX + half
    local minY = robotY - half
```

```
        local maxY = robotY + half

        local cMin = math.max(1, math.floor(minX / mapResolution - gxMin) + 1)
        local cMax = math.min(cols, math.floor(maxX / mapResolution - gxMin) +
        1)

        local rMin = math.max(1, math.floor(gyMax - maxY / mapResolution) + 1)
        local rMax = math.min(rows, math.floor(gyMax - minY / mapResolution) +
        1)

        for r = rMin, rMax do
                for c = cMin, cMax do
                        if cleanableMap[r][c] == 1 and not covered[r][c] then
                                covered[r][c] = true
                                coveredCount = coveredCount + 1

                                if showCoveredCells and greenDraw and
                                shouldDrawCell(r, c) then

                                        local x, y = gridToWorld(r, c)
                                        sim.addDrawingObjectItem(greenDraw, {x, y,
                                        coverageDrawZ})
                                end
                        end
                end
        end
end

function sysCall_sensing()
        local p = sim.getObjectPosition(robot, -1)
        local robotX = p[1]
        local robotY = p[2]

        if lastRobotX ~= nil and lastRobotY ~= nil and interpSteps > 0 then
                for i = 0, interpSteps do
                        local t = i / interpSteps
                        local xi = lastRobotX + t * (robotX - lastRobotX)
                        local yi = lastRobotY + t * (robotY - lastRobotY)
                        markCoveredAroundRobot(xi, yi)
                end
        else
                markCoveredAroundRobot(robotX, robotY)
        end

        lastRobotX = robotX
        lastRobotY = robotY

        updateUiLabels()
end
```