Indian Institute of Technology, Tirupati

Department of Computer Science and Engineering

Development of a Completely Functional 2-D Ground Robot Using ROS and Simultaneous Localization and Mapping (SLAM)

Submitted in partial fulfillment of the requirements
for the degree of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by
Ayush Bilkhiwal
CS23M119

Supervisor:
Dr. Chalavadi Vishnu
Tanveer Ahmed

13-May-2025

Declaration

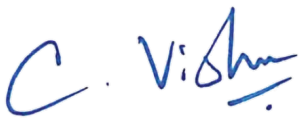
I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission to the best of my knowledge. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Place: Tirupati
Date: 13-05-2025

Signature
Ayush Bilkhiwal
CS23M119

BONAFIDE CERTIFICATE

This is to certify that the report titled Prepare a Completely Functional 2-D Ground Robot Using ROS and Simultaneous Localization and Mapping (SLAM), submitted by Ayush Bilkhiwal, to the Indian Institute of Technology, Tirupati, for the award of the degree of Master of Technology, is a bona fide record of the project work done by him under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.



Place: Tirupati
Date: 13-05-2025

Dr. Chalavadi Vishnu
Guide
Assistant Professor
Department of Computer Science and Engineering
IIT Tirupati- 517501

ACKNOWLEDGMENTS

Over the course of the entire semester, I would like to express my profoundest appreciation to my advisor, **Dr. Chalavadi Vishnu**, for his steadfast encouragement, invaluable counsel, and consistent support. I have been profoundly influenced by his guidance throughout my scholarly trajectory.I also extend my heartfelt thanks to **Mr.Tanveer Ahmed**, Senior Software Architect at BOSCH, for his technical support, practical insights, and valuable suggestions during the course of this project. I am writing to convey my heartfelt gratitude to the entire faculty of the Department of Computer Science and Engineering. Their unwavering commitment to teaching strategies and eagerness to impart wisdom have furnished me with priceless educational opportunities. Their dedication to achieving high standards has provided me with access to fresh prospects, expanding my horizons, and honing my expertise in the respective domain. Regardless of the situation, I was privileged to be met with constructive feedback from my senior and junior peers. Their advice and help have been very important to my academic growth. It was easy for everyone to work together and support each other in the department, which made learning a lot more fun. I am thankful for my family because they are always there for me and help me in every aspects. The things that kept me going were their unwavering faith in my abilities and support when things got tough. Finally, I want to say a huge thank you to everyone who helped me along the way in school. Your help and advice have been invaluable, and I am very thankful for the life-changing experiences and information I have gained.

ABSTRACT

This project presents the development of an autonomous 2D ground robot that integrates advanced sensor feedback and a ROS-based control system for precise mapping and navigation. The robot operates using Simultaneous Localization and Mapping (SLAM) algorithms, enabling it to construct real-time maps of its surroundings while navigating autonomously from one point to another. Unlike the initial simulation phase in the Gazebo environment, this project has transitioned to a real-world robotic system equipped with various sensors and actuators. The implementation is centered around XRCE-DDS communication, ensuring efficient and real-time data exchange between different system components, including the Robotics Control Board and various peripheral devices.

The robot is equipped with multiple sensors, including a LiDAR for environmental perception, an Inertial Measurement Unit (IMU) for orientation and motion tracking, and encoders for precise wheel odometry. Additionally, actuators such as a side brush and blower have been incorporated to enhance the robot's functionality for specific applications. The ROS-based control system facilitates motion planning, obstacle detection, and path optimization, ensuring the robot navigates efficiently while avoiding collisions. The SLAM navigation stack has been integrated to enable the robot to create an accurate map of its environment and use this map for localization and autonomous path execution. The use of XRCE-DDS communication protocol allows seamless data transmission between the different subsystems, improving reliability and real-time performance.

The scope of this project extends beyond just simulation and theoretical implementation. It involves the integration of both hardware and software components to build a fully functional robotic platform capable of operating in real-world conditions. The system has been tested in various environments to validate its performance, ensuring that the implemented navigation algorithms and sensor fusion techniques work as expected. Potential applications of this autonomous robotic system include logistics, disaster response, industrial automation, and further research in SLAM-based navigation techniques.

This report follows a structured approach, detailing the technological innovations and methodology employed in the project. It outlines the objectives, system design, implementation strategies, experimental validation, and key results. The document is aligned with the standard guidelines for thesis preparation and includes a chapter-wise breakdown of the project's development process. By integrating multiple sensors and real-time communication mechanisms, this project makes a significant contribution to the field of autonomous robotics, demonstrating a practical implementation of SLAM-based navigation in a real robotic system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The motivation for this project stems from the growing demand for autonomous systems across industries and research domains. Autonomous robots play a pivotal role in automating tasks in dynamic and uncertain environments, such as warehouses, disaster zones, and research facilities. The integration of Simultaneous Localization and Mapping (SLAM) technologies ensures these robots can navigate unknown terrains with precision, making them indispensable for logistics, exploration, and safety-critical applications.

This project aims to leverage advancements in robotics middleware, such as ROS2 Foxy, to provide a scalable and reliable framework for real-time navigation and mapping. The incorporation of XRCE-DDS communication addresses the need for low-latency and efficient data exchange, especially critical for embedded systems like the Jetson Nano. By combining these technologies, the robot is designed to operate with high accuracy and reliability, even in resource-constrained environments.

Furthermore, the project has academic significance, offering opportunities for hands-on learning in system integration, algorithm development, and hardware-software interfacing. It bridges the gap between theoretical knowledge and practical applications, equipping researchers and students with the expertise to contribute to the rapidly evolving robotics field. The outcome of this project is not just a functional robot but also a foundation for exploring future innovations in autonomous systems.

1.2 Problem Statement

Develop an autonomous 2-D ground robot capable of mapping and navigation by integrating sensor feedback from BOSCH's Power Control Board with Robotics Control Board through XRCE-DDS. Using this data, a SLAM algorithm will be developed to achieve accurate mapping and navigation, enabling the robot to move autonomously from point A to point B with high precision and reliability.

Chapter 2

Literature Review

Tian Bai the Multi-RRT algorithm is a modified version of the optimal sampling-based RRT approach, specifically designed to address the path planning challenge involving multiple waypoints in smart home environments. This algorithm generates several trees by employing a straightforward extension and connection technique, allowing efficient navigation through all designated waypoints. Simulation experiments show that Multi-RRT achieves faster convergence compared to standard and biased RRT methods, with reduced computational time and fewer nodes required. Its effectiveness was further demonstrated through real-world implementation on a TurtleBot2 within a smart home setup, where it successfully conducted search tasks informed by historical human location data. These results highlight the algorithm's reliability and efficiency for real-time mobile robot navigation.[1]

Zixiang Liu this work details the implementation of real-time localization and mapping for mobile robots using the FastSLAM algorithm, which leverages particle filters within the ROS framework. It addresses the challenges of simultaneous localization and mapping (SLAM) alongside local path planning by integrating an enhanced artificial potential field method to improve obstacle avoidance. The ROS platform effectively supports these capabilities, as demonstrated through simulation experiments conducted in the Gazebo environment. Enhancements to the potential field approach reduce the impact of excessive attractive forces, resulting in smoother and more stable navigation. The outcomes confirm the system's ability to avoid dynamic obstacles and perform efficient path planning, offering a reliable solution for mobile robot navigation in complex and dynamic settings.[2]

Kazi Mahmud this study introduces the DVR-1, an autonomous vacuum cleaning robot designed with a focus on affordability, lightweight construction, and effective obstacle avoidance. The robot operates in both manual and autonomous modes, employing four distinct path planning strategies: random walk, spiral, S-pattern, and wall-following. Equipped with bumper and cliff sensors, it ensures reliable detection and navigation around obstacles, while a cyclonic vacuum mechanism handles debris collection. Experimental evaluations conducted in a 12×12 square foot area revealed that the combined use of all path planning algorithms significantly improves cleaning coverage. With an overall production cost of approximately US$55, DVR-1 offers a cost-effective alternative to commercial models such as the iRobot Roomba and Neato.[3]

Rajesh Kannan this study examines the differences between the planned and actual trajectories of a differential drive robot utilizing the ROS navigation stack in a Gazebo simulation environment. The robot is tasked with reaching four predetermined destinations while navigating around dynamic obstacles, with its path coordinates recorded for performance evaluation. Results show that travel time increases as the number of obstacles rises—manual control required 2 minutes and 30 seconds in a clear environment, compared to 3 minutes and 26 seconds when four obstacles were present. The findings underscore the value of simulation-based analysis in detecting and addressing path planning issues prior to real-world deployment, and demonstrate the effectiveness of SLAM algorithms in supporting autonomous navigation.[4]

Jinming Yao this research proposes a hybrid path planning approach for indoor mobile robots, combining an optimized A* algorithm for global route planning with an enhanced artificial potential field method for local navigation. The refined A* algorithm incorporates jump point search and second-order Bessel curves, effectively minimizing sharp turns and producing smoother paths. This results in a pathfinding speed improvement of around 41.60% compared to conventional techniques. The proposed method was evaluated through both MATLAB simulations and physical testing, confirming its strong performance in navigating complex indoor environments and avoiding obstacles. The study highlights the critical role of precise localization in autonomous movement, supporting the broader goal of integrating mobile robots into everyday human settings.[5]

Cesar Cadena this review examines the progression, current developments, and future prospects of Simultaneous Localization and Mapping (SLAM), emphasizing its pivotal role in robotics over the last three decades. It outlines the conventional SLAM framework and explores persistent challenges such as ensuring robustness, achieving scalability, and effectively representing complex environments. The authors contend that SLAM remains an open problem, as its effectiveness is still influenced by factors like the type of robot, environmental conditions, and specific performance demands. Introducing the concept of the "robust-perception age," the paper underscores the need for systems with minimal failure rates and enhanced semantic understanding. Additionally, it highlights the promise of emerging sensors and deep learning techniques in advancing SLAM capabilities, concluding that continuous research is vital to develop dependable and practical environmental models for real-world applications.[6]

J. Shao introduced an improved back-EMF detection method for sensorless BLDC motor drives is proposed, enabling EMF sampling during both PWM on and off periods, thus removing the duty cycle limitation of earlier methods. This allows for more efficient operation at higher duty cycles without needing motor neutral sensing. Additionally, techniques relevant to automotive use—such as rotation detection and current sensing—are also introduced to enhance system robustness and control precision.[7]

AL Mamun presents an embedded system for motion control of omnidirectional mobile robots, specifically designed for the RoboCup small-size league. The system integrates a fuzzy-tuned proportional-integral (PI) path planner with a linear quadratic regulator (LQR)

controller, enhancing trajectory accuracy and energy efficiency. Simulation and experimental results demonstrate that the combined fuzzy-PI LQR controller outperforms traditional PI controllers, achieving a mean position deviation of 1.1% and orientation deviation of 3.2% during trajectory tracking. The research emphasizes the importance of adaptive control in dynamic environments, confirming that the proposed method significantly improves motion precision in omnidirectional robots, particularly under varying conditions.[8]

Chapter 3

Background Information

Autonomous robots have become integral to modern industries due to their ability to perform complex tasks in dynamic and unpredictable environments. Power control modules play a crucial role in enabling these robots to respond effectively to real-time challenges.Unlike traditional control systems, adaptive systems can modify their parameters based on feedback from the environment, enhancing precision and reliability.
Autonomous mobile robots (AMRs) are particularly valued for their ability to navigate without human intervention, relying on advanced sensing and control mechanisms. They are used across various sectors such as manufacturing, healthcare, agriculture, and logistics domain to perform tasks including carrying heavy objects, monitoring environments, and executing search and rescue missions. AMRs achieve these functionalities using a combination of environmental sensors like IMUs and wheel encoders, which provide critical data for motion tracking, localization, and navigation.
Studies emphasize the importance of sensor fusion techniques to integrate data from multiple sources, enabling accurate decision-making. The effective implementation of these techniques helps overcome challenges related to localization, obstacle avoidance, and system scalability. For example, IMUs and wheel encoders, combined with robust control algorithms, allow AMRs to operate reliably in dynamic and partially unknown environments. Moreover, actuators such as drive motors, blower motors enhance the robot's capability to perform physical tasks efficiently.

3.1 Experimental Setup

The experimental setup for this project involves the integration of hardware and software components to achieve autonomous navigation and mapping. The robot is designed to operate in dynamic environments using advanced technologies like ROS2 Foxy, SLAM algorithms, and XRCE-DDS communication. Below is a detailed description of the setup:

3.1.1 Hardware Setup

The hardware architecture serves as the foundation of the adaptive robotic system, acting as the backbone that connects and coordinates all key modules including sensors, actuators,

processing units, and communication interfaces.. At the heart of the architecture lies an embedded control platform based on a powerful STM32 microcontroller, supported by dedicated motor drivers, custom power circuitry, and safety mechanisms. To enable environment awareness and motion tracking, a variety of sensors such as an Inertial Measurement Unit (IMU), wheel encoders, cliff sensors, wall-following sensors, and collision switches are integrated into the system. These sensors provide the essential data required for localization, obstacle detection, and decision-making. The actuator system comprises high-performance drive motors, side brushes, bristle rollers, and a blower motor—all controlled via optimized BDC and BLDC motor driver ICs tailored for efficiency and reliability.
The adaptive robotic system centers around an STM32-based embedded control module, designed explicitly for robotic applications. It integrates multiple microcontrollers, including an STM32G4 as the primary controller for real-time control and adaptive decision-making. Additionally, an Infineon TLE9879 microcontroller manages sensor less BLDC motor commutation (blower motor), while a dedicated Renesas RA2E1 microcontroller handles user-interface components such as touch buttons and LED backlighting. The system incorporates essential sensors such as an IMU and quadrature encoders, high-performance DC motor drivers, and customized power management modules to ensure robust, efficient, and safe operation under dynamic conditions.

3.1.2 Embedded Processing Unit

The embedded processing unit is central to the system's operation, enabling robust real-time control and adaptive decision-making.

Parameter	**Specification**
Microcontroller	STM32G4 series, Arm® Cortex®-M4 CPU with FPU
Operating Frequency	Up to 170 MHz (213 DMIPS performance)
Flash Memory	512 KB with ECC support
SRAM	96 KB (Hardware parity check on first 32 KB)
GPIO	Up to 107 fast I/O ports
Voltage Input	11.5V to 17V DC
Onboard Power	5V/2A (Buck), 3.3V/800mA (LDO)

Table 3.1: STM32G4 Microcontroller Specifications

3.1.3 Integrated Sensors

The robotic system incorporates a wide array of sensors to ensure accurate perception, navigation, and safe operation. An Inertial Measurement Unit (IMU) is used to monitor orientation and motion, providing essential data for maintaining balance and direction. Wheel encoders enable precise tracking of displacement and speed, forming the basis for odometry. Additional sensors such as cliff detection modules, wall-following sensors, and collision switches help the robot avoid obstacles, navigate confined spaces, and prevent falls from ele-

vated surfaces. These sensors work in tandem to provide real-time environmental feedback, enabling the system to make informed decisions during operation.

3.1.3.1 IMU (Inertial Measurement Unit)

The IMU integrated into the adaptive robotic platform is the Bosch BMI088, a high-performance inertial measurement sensor designed specifically for applications requiring robust, accurate, and real-time orientation and motion tracking. The BMI088 IMU provides precise measurements of motion, orientation, and stability. It integrates a high-precision triaxial accelerometer and a high-performance triaxial gyroscope, essential for robotic navigation, stabilization, and adaptive control.

Sensor Overview

(a) IMU Sensor

(b) BMI connection with SPI

Figure 3.1: Cleaning components: Bristle roller and side brush motors

The BMI088 consists of two primary sensing components:

- 16-bit Digital Triaxial Accelerometer
- 16-bit Digital Triaxial Gyroscope

The sensor provides raw accelerometer data in gravitational force (g) and raw gyroscope data in degrees per second (°/s). Additionally, the integrated BSXlite sensor fusion library translates raw sensor data into robust orientation information. The BMI088 is known for its low noise and high stability, making it well-suited for dynamic mobile platforms. Its high resistance to vibration ensures accurate measurements even in motion-intensive environments. Communication with the microcontroller is typically established via SPI, allowing seamless integration into embedded systems. The compact form factor and low power consumption make it ideal for continuous real-time operation. Together, these features make the BMI088 a reliable choice for motion tracking in autonomous robotic systems.

3.1.3.2 Wheel Encoder Module

The wheel encoder module is an integral part of the adaptive robotic control system, designed to measure wheel rotation accurately, calculate velocity, and provide precise odometry

data. The wheel encoder utilizes quadrature encoder mode for robust measurement of wheel rotation direction and speed.

Purpose

- Provides accurate wheel odometry data essential for navigation and localization.
- Supplies real-time wheel velocity information (ticks per second) as feedback for precise motor control (PI control loops).
- Tracks absolute wheel rotations (in ticks) since system initialization, enabling total distance traveled estimation

The wheel encoders utilize hall-effect sensors integrated with drive wheels to provide quadrature signals. These signals are processed using hardware timer peripherals on the STM32G474 microcontroller, configured in quadrature encoder mode.

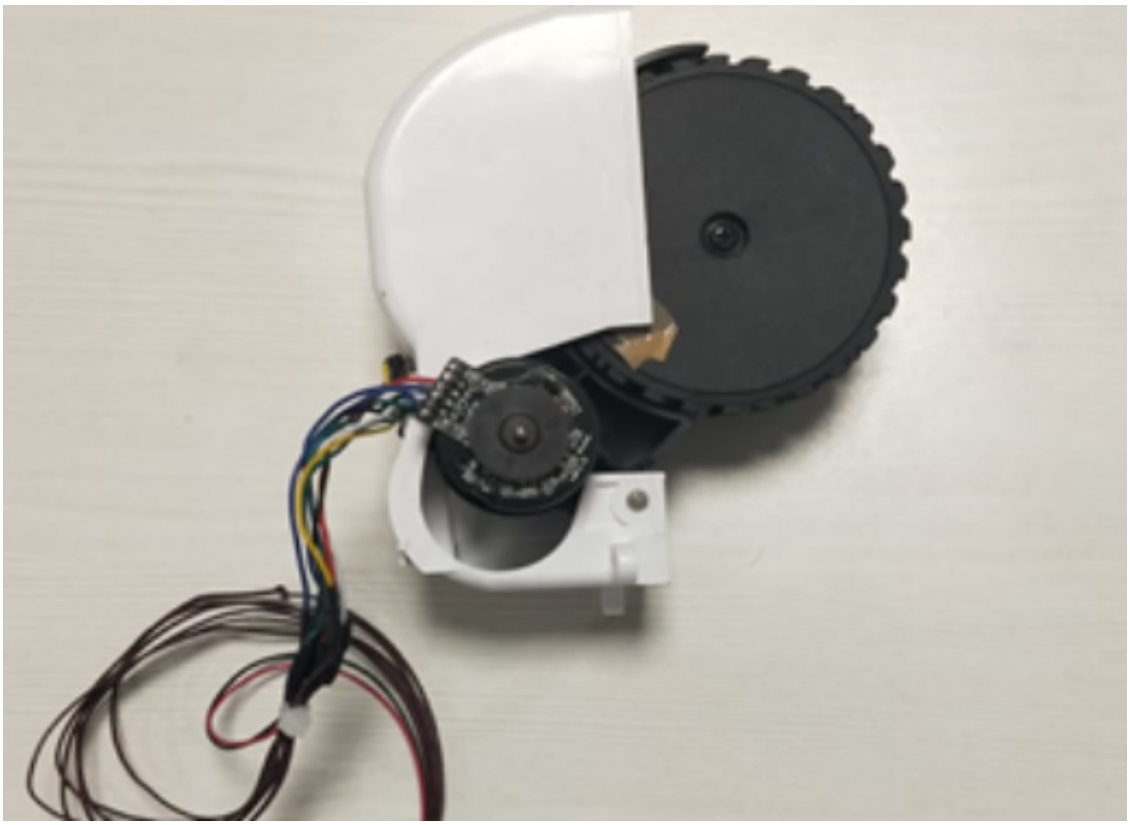


Figure 3.2: Wheel with encoder

3.1.4 Actuators

Actuators translate adaptive control signals from the microcontroller into precise mechanical movements, enabling the robot to execute navigation, cleaning, and other functional tasks effectively. This robotic system incorporates robust Bidirectional DC (BDC) motor drivers and a powerful Brushless DC (BLDC) motor driver for precise, efficient, and dynamic actuator control.

3.1.4.1 Bidirectional DC Motor (BDC) Drivers

BDC motor drivers are responsible for controlling the direction and speed of DC motors used in the robot's drive system, side brushes, and bristle rollers. These drivers allow the motors to run forward or backward and support braking modes for precise control. Integrated driver ICs such as DRV8874, DRV8218, and DRV8251A are used for their built-in current regulation, fault protection, and ease of control through PWM signals. They ensure smooth motor operation even under variable loads. Their reliable performance and compact integration make them ideal for real-time robotic applications.

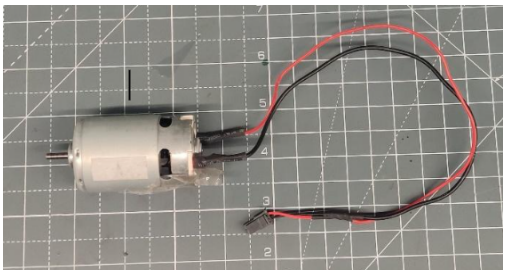
Motor Driver	**Voltage**	**Current (Continuous/Peak)**	**Operating Modes**	**Application Description**
Motor Driver 1 & 2	11.5V to 17V DC	1A / 2.5A	Forward, Reverse, Brake (Fast/Slow Decay)	Drive wheels. Robust H-bridge driver with adjustable current limiting, offering precise control of wheel movement and braking capabilities.
Motor Driver 3	11.5V to 17V DC	0.5A / 1A	Forward, Reverse, Brake (Fast/Slow Decay)	Side Brushes. Compact motor driver providing accurate directional control and braking for small to medium load applications.
Motor Driver 4	11.5V to 17V DC	3A / 7A	Forward, Brake	Bristle Roller. High-current motor driver suited for demanding applications, enabling high-torque operation with forward and brake control.

Table 3.2: Specifications of Motor Driver Modules Used in the Robotic System

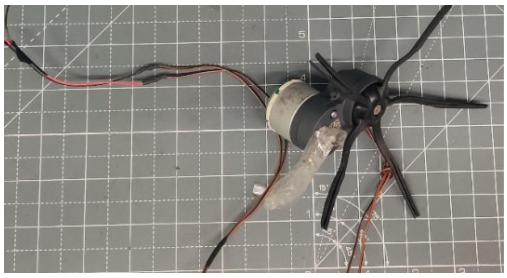
The bristle roller is used to sweep and agitate dirt or debris from the floor surface, making it easier to collect. It rotates at high speed to lift dust particles, especially from textured or carpeted surfaces. The side brush, on the other hand, is designed to clean edges and corners where the main roller cannot reach. It helps pull in dirt from walls and tight spots into the main cleaning path.. Both components are driven by dedicated motors controlled via embedded drivers to ensure precise speed and direction. Their operation can be adjusted based on cleaning mode or floor type. This flexibility allows the robot to adapt to different environments and maintain consistent cleaning performance.

3.1.4.2 BLDC Motor Driver (Brushless DC Motor):

The robotic system integrates a Brushless DC (BLDC) motor driven by an Infineon TLE9879 microcontroller, using an external three-phase half-bridge. The BLDC motor provides continuous, efficient power specifically for the blower mechanism, utilizing the advanced Field-



(a) Bristle Roller Motor



(b) Side Brush Motor

Figure 3.3: Cleaning components: Bristle roller and side brush motors

Oriented Control (FOC) method for precise speed control.
Technical Highlights:

- Operating Voltage: 11.5V to 17V DC
- Power Rating: 100 W continuous
- Controller: Infineon TLE9879 with external 3-phase half-bridge
- Speed Control Technique: Field-Oriented Control (FOC)

The use of sensor less FOC technique ensures precise motor control, smooth operation, and energy efficiency, significantly improving overall system performance and reliability.

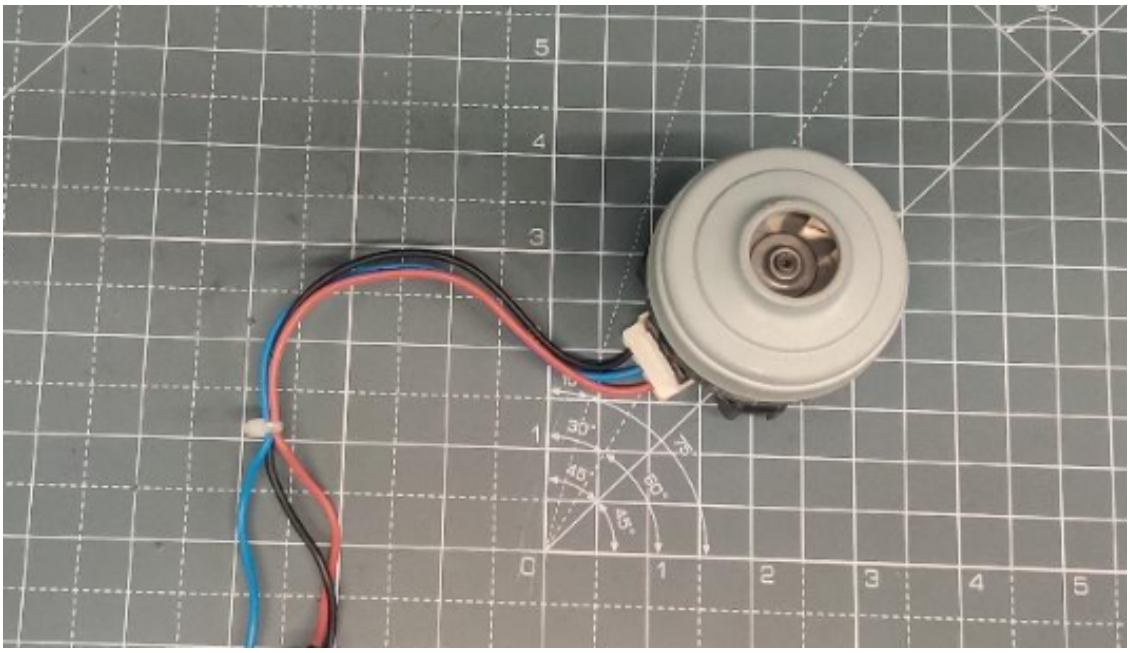


Figure 3.4: BLDC Motor

3.1.5 Custom Control and Power Module

The robotic system integrates a specialized Custom Control and Power Module designed to effectively manage, regulate, and distribute power to various components. It operates within an input voltage range of 11.5 V to 17 V DC, providing stable output voltages through dedicated regulators:

- 5 V DC supply: High-efficiency Buck converter (2A) powering primary electronics.
- 3.3 V DC supply: Low Dropout (LDO) regulator (800mA) for microcontrollers and sensitive circuitry.

The module also monitors critical voltages, including battery levels, motor voltages, and internal reference voltages, ensuring reliable operation. An Under-voltage Lockout (UVLO) feature is implemented to maintain system safety by preventing operation during low-voltage conditions.
Safety mechanisms integrated into this module include:

- 12 A fuse dedicated to motor power protection.
- Adjustable 4 A eFuse for external module protection.
- Dedicated safety switches for controlled isolation of motor power.
- Robust EMI filtering and ESD protection circuits.

These protective features significantly enhance reliability, durability, and overall system performance.

3.1.6 User Interface Components

The hardware integrates intuitive user interface components that enhance user interaction and provide real-time system feedback through touch inputs and dynamic LED indicators.
Capacitive Touch Buttons
The system includes three capacitive touch buttons, managed by a dedicated Renesas RA2E1 microcontroller. These buttons allow seamless, contactless user input for functions such as mode selection, start/stop commands, or manual overrides. The touch inputs are responsive, reliable, and durable, designed for consistent performance in everyday operation.
LED Indicators
Each touch button is paired with an integrated RGB LED, capable of displaying a wide range of colors using PWM (Pulse Width Modulation) control. These LEDs are also controlled by the RA2E1 microcontroller via I^2C, and are used to indicate different system states or modes.This combination of touch input and multicolor LED feedback provides a user-friendly interface, ensuring operators can easily monitor and interact with the robot's status in real-time.

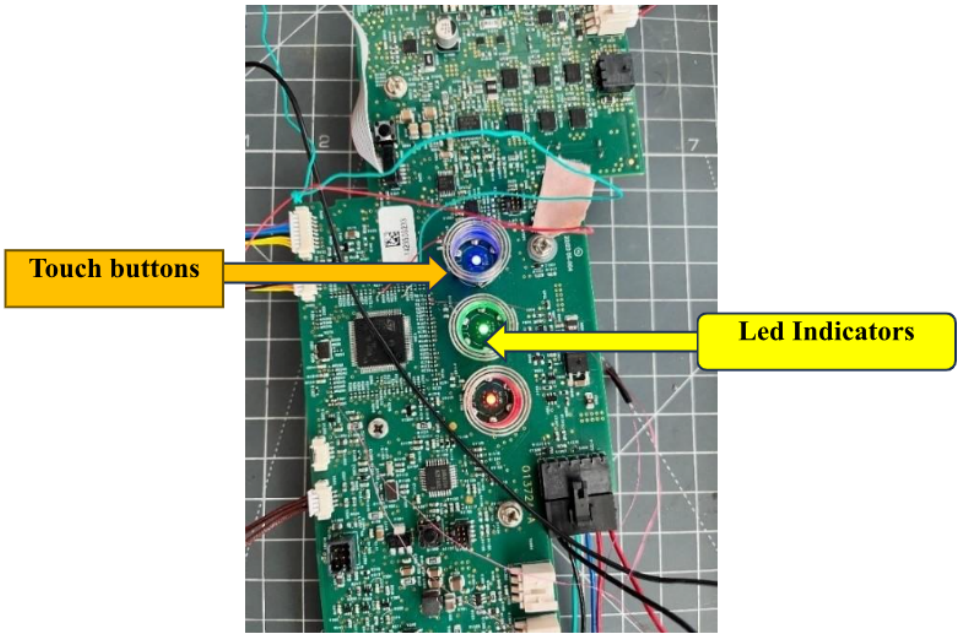


Figure 3.5: Touch buttons and LED Indicators

3.1.7 XRCE-DDS Communication Interface

The robotic system utilizes XRCE-DDS (eXtremely Resource Constrained Environment – Data Distribution Service) as a lightweight and efficient communication protocol for interfacing between the embedded controller (STM32) and a higher-level processing unit such as the Jetson Nano.XRCE-DDS enables real-time communication between microcontrollers and microprocessors by allowing the embedded device to act as a ROS 2-compatible node, capable of publishing and subscribing to topics over a UART interface.

Functionality:
Acts as a ROS 2 publisher and subscriber, enabling exchange of real-time data such as sensor readings, motor feedback, control commands, and system states. Supports communication of lightweight, structured messages between ROS 2 nodes and the embedded system, compliant with DDS standards. This integration allows seamless two-way communication between the low-level hardware and high-level autonomy stack, enabling tasks like navigation, mapping, and decision-making to be executed on the Jetson Nano while the STM32 handles real-time motor control and sensor.

3.1.7.1 Application Layer

The Application Layer sits at the top of the software architecture and is responsible for orchestrating the overall behavior and decision-making of the robotic system. It manages

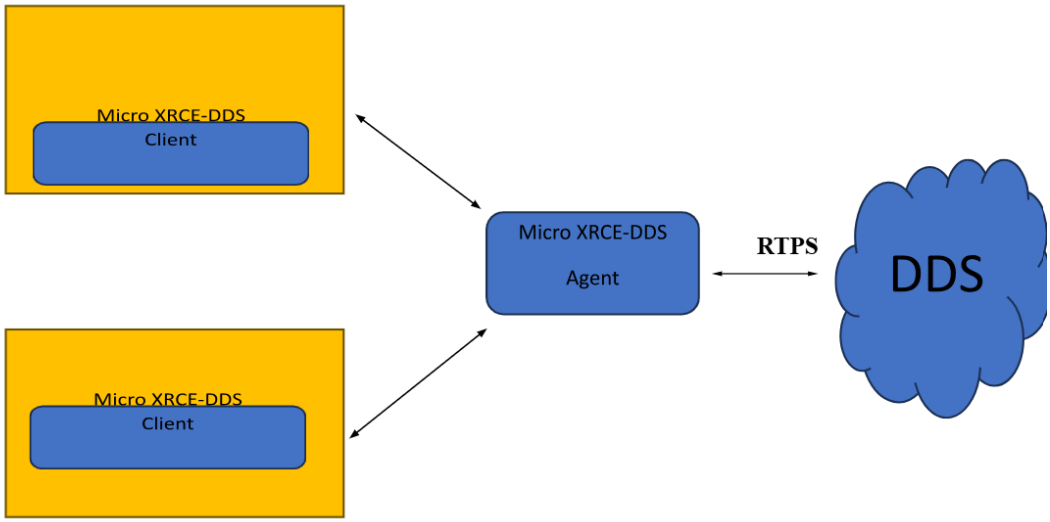


Figure 3.6: Block diagram of XRCE-DDS Communication

high-level logic, system states, and coordinates the actions of various functional subsystems. Key components of this layer include

- **System State Machine:** Governs the robot's overall operation flow, transitioning between states such as idle, cleaning, docking, or error-handling based on inputs and conditions.
- **Safety manager:** Continuously monitors critical safety parameters and ensures immediate response to events like emergency stops, sensor faults, or unsafe conditions.
- **Drive units manager:** Handles motion planning and coordination of drive units, managing speed and direction based on control inputs or autonomous navigation logic.
- **Power manager:** Supervises the robot's power system, managing power-saving states, battery status, and critical power-related decisions.
- **Blower manager, Side brush manager, Bristle roller manager:** Control the operational logic and cleaning cycles of respective actuators, ensuring synchronized and efficient cleaning operations.

This layer encapsulates the robot's intelligent behavior and translates user commands, sensor data, and control policies into real-time coordinated actions. It communicates directly with the control layer to execute the decisions at the hardware level.

3.1.7.2 Control Layer

The Control Layer serves as the intermediate bridge between the high-level application logic and the underlying hardware interfaces. It is responsible for implementing real-time control

logic and managing precise execution of commands issued by the application layer.This layer translates abstract system-level instructions into actionable control signals for actuators and peripheral components, ensuring synchronized and responsive behavior.
Key modules within this layer include:

- **Illumination control:** Controls the RGB LED indicators and touch button backlighting. It reflects system states such as active, idle, error, or mode transitions through visual feedback, enhancing user interaction and system visibility.
- **Drive units control:** Manages the real-time behavior of the wheel motors by processing speed and direction commands. It generates appropriate PWM signals and direction control based on input from the drive units manager, ensuring smooth and adaptive navigation.

By handling these time-critical operations, the Control Layer plays a vital role in delivering responsive and coordinated system behavior, while isolating higher-level decision-making from hardware-level execution.

3.2 Software Environment

The software stack utilizes advanced frameworks and tools to ensure seamless integration and operation:

- **ROS2 Foxy**: This middleware framework enables modular development, efficient communication, and real-time operation of the robot's control system.Facilitates communication between software nodes and hardware components through XRCE-DDS.
- **Gazebo Simulation**: A virtual simulation environment used for testing SLAM algorithms, navigation, and obstacle avoidance in a controlled setting before real-world implementation.
- **SLAM Algorithms**: Open-source SLAM solutions such as Google Cartographer are employed for accurate mapping and localization.
- **Programming Languages**: The system is implemented using Python and C++ for high-performance and flexibility.

3.2.1 Software Architecture

The software architecture of the autonomous room-cleaning robot is built on top of the Robot Operating System 2 (ROS2), specifically the Foxy Fitzroy distribution. This version was chosen because it offers long-term support (LTS) and brings the advantages of real-time communication, modular design, and enhanced performance over its predecessor, ROS1. The ROS2 ecosystem provides the backbone for the entire software system, enabling seamless integration between various components, both in simulation and on the actual robot.

The architecture is designed in a layered manner to simplify development, testing, and debugging. At its core lies a perception subsystem, which is responsible for gathering data

from a wide array of onboard sensors including a 2D LIDAR, an Inertial Measurement Unit (IMU), wheel encoders, capacitive touch sensors, cliff detection sensors, and wall-following proximity sensors. These sensors are critical for understanding the robot's surrounding environment and its own state within it. The data collected from the perception layer is published to ROS2 topics, making it accessible to higher-level nodes such as those responsible for localization and navigation.

Above the perception layer is the planning layer, which plays a central role in computing how the robot should move to achieve its objectives. This is implemented using the ROS2 Navigation Stack (Nav2), which offers a modular set of nodes responsible for global path planning, local path adjustment, obstacle avoidance, and behavior tree-based decision-making. The planning layer consumes sensor data and map information to compute a safe and efficient trajectory for the robot to follow as it cleans a room or navigates from one point to another.

The control layer translates the planned motion into actual movement. Based on the computed paths and velocity commands, it publishes control signals—typically as $\text{geometry}_m sgs/Twistmessages|thataretransmittedtotheSTM32-basedmotorcontroller.$
.

For testing, debugging, and live monitoring, the entire system is visualized through RViz2, which is tightly integrated into the ROS2 ecosystem. RViz2 allows for real-time visualization of sensor streams, robot model pose, planned paths, and detected obstacles. This visualization is instrumental during both the simulation phase and real-world testing, as it enables the developer to inspect and verify system behavior.

Before deployment on hardware, all functionalities were rigorously tested in a simulated environment built in Gazebo 11. The simulation includes the robot's full kinematic model described through a URDF (Unified Robot Description Format) file, which includes both the physical dimensions and the joint constraints of the robot. 3D mesh files are used to provide a more accurate visual and collision representation of the robot within the simulation environment. Sensor plugins mimic real sensor behavior, and the robot is spawned in a virtual room setup that resembles real-world environments.

The final integration stage involves mapping all these components—sensor drivers, control nodes, navigation stack, and visualization tools—to the actual hardware. This includes setting up ROS2 nodes on a host computer that communicates with the STM32 microcontroller via a serial or USB interface. Configuration files, launch files, and parameter servers ensure that the entire system is brought up in a deterministic and robust manner. In the real-world robot, the software stack operates reliably, allowing the robot to navigate autonomously, avoid obstacles, and clean rooms efficiently.

This layered and modular architecture ensures that each component of the software can be tested and improved independently while maintaining smooth interoperation. The use of ROS2 Foxy, together with Gazebo, RViz, and custom firmware, results in a powerful, flexible, and real-time-capable system that effectively bridges the gap between simulation and physical deployment.

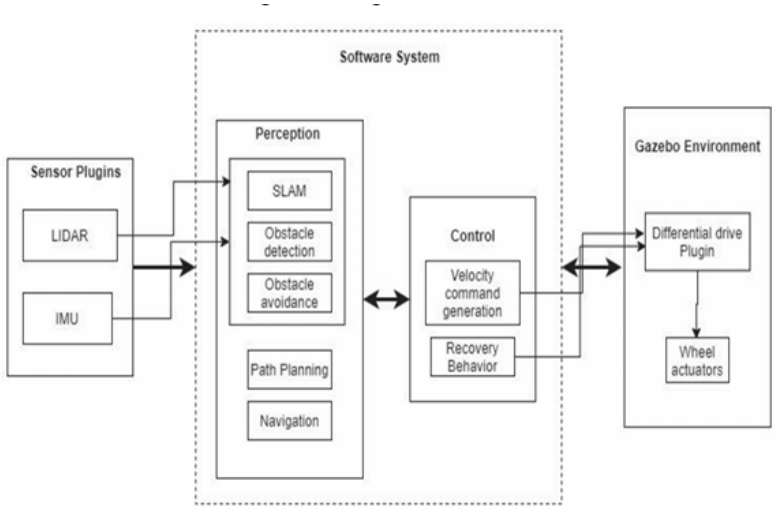


Figure 3.7: System Architecture

3.2.2 Simulation Environment: Gazebo and URDF Integration

Before deploying the autonomous robot on physical hardware, the software stack was comprehensively tested in a virtual environment built using Gazebo 11. Gazebo served as a dynamic, physics-based 3D simulator that provided a realistic testing ground where the robot's behavior, sensor integration, and autonomous navigation pipeline could be evaluated under near-real-world conditions.

At the core of the simulation environment was the robot model, meticulously designed using the Unified Robot Description Format (URDF). This model defined every structural and functional element of the robot, including its base chassis, wheels, sensors, and actuators. The use of modular ¡xacro¿ macros made it easy to maintain and extend the model. Each link and joint in the robot was defined with realistic mass, inertia, and physical constraints, ensuring that the simulated motion closely matched what would be expected in the real world.

To create a visually and functionally accurate representation, 3D mesh files (in STL and DAE formats) were included in the URDF. These meshes depicted the true structure of the robot, including design details like motor housing, sensor placement, and bumpers. While these meshes enhanced the visual realism in Gazebo, separate simplified collision models were used to ensure efficient physics calculations. The result was a robot that not only moved like the real one but also looked the part. The robot was spawned into a customized simulation world designed to resemble indoor environments. This virtual space included walls, narrow passages, furniture-like obstacles, and textured flooring, replicating the kind of layout that the robot would later encounter during real cleaning tasks. The virtual world was defined using the Simulation Description Format (SDF), and additional elements like lighting and material friction were adjusted to match realistic indoor conditions.

Sensor simulation was a crucial part of this environment. LIDAR was emulated using Gazebo plugins that published laser scan data on the /scan topic. An IMU plugin provided orientation and angular velocity data, while simulated wheel encoders were configured to produce odometry readings based on wheel rotation. These virtual sensors replicated the

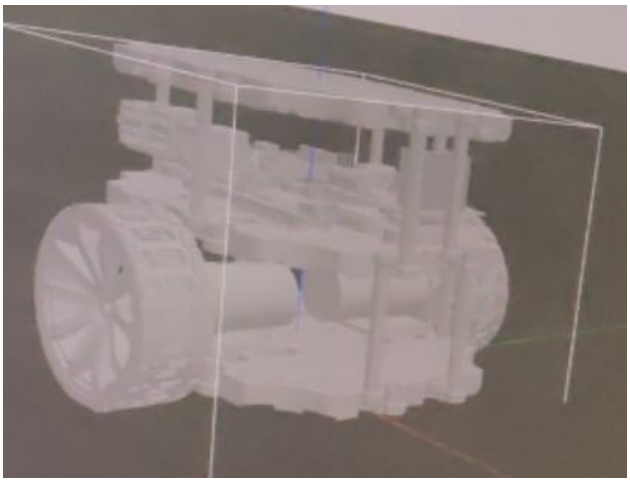


Figure 3.8: Real-time visualization of robot odometry in RViz.

characteristics of their physical counterparts, including measurement noise, frequency, and field of view. This made it possible to develop and validate the perception and localization components without needing the physical robot online.

To visualize the robot's operation and perform system-level debugging, RViz2 was used in parallel with the simulation. Through RViz, live sensor data such as laser scans, odometry, TF frames, and the robot's 3D model could be visualized in real-time. One of the most valuable uses of RViz during development was observing the robot's localization status—comparing its perceived position against the ground truth provided by Gazebo. It also helped verify the correctness of the transform tree and ensured all components were communicating as expected.

In addition to visualization, ROS2 launch files were developed to automate the startup of the entire simulation pipeline. These included the robot spawn process, sensor plugin initialization, map loading, and controller activation. Using these launch files, repeatable tests could be conducted with minimal setup, enabling consistent benchmarking of navigation behavior under different configurations.

This simulation-first development strategy proved invaluable. It allowed early detection of integration issues, safe testing of edge-case scenarios, and confident tuning of parameters like obstacle inflation radius, velocity smoothing, and localization particle count. Once the robot demonstrated stable performance in simulation—including successful navigation, accurate localization, and reliable obstacle avoidance—it was ready for hardware testing.

3.2.3 RViz Configuration and Visualization

RViz2 played an integral role in the development, testing, and debugging of the autonomous room-cleaning robot. As a powerful 3D visualization tool integrated within the ROS2 ecosystem, RViz served as the central interface for real-time inspection of robot states, sensor data, navigation plans, and coordinate transformations. During both the simulation and hardware phases, RViz was used to validate the correctness of data flow across the system and to intuitively understand the robot's internal decision-making processes.

The RViz setup was carefully configured to reflect the state of the robot and its environment with maximum clarity. The 3D model of the robot, based on the same URDF used in Gazebo, was visualized in RViz using the RobotModel display type. This allowed for the real-time observation of link movements and joint states. The Transform (TF) display helped verify that all coordinate frames—from the robot's base frame (base link) to the sensors (lidar frame, imuframe, etc.)—were being published and updated correctly. Misalignments or missing frames could be quickly identified through this interface, making it easier to resolve issues related to sensor integration or robot description. One of the most informative as-

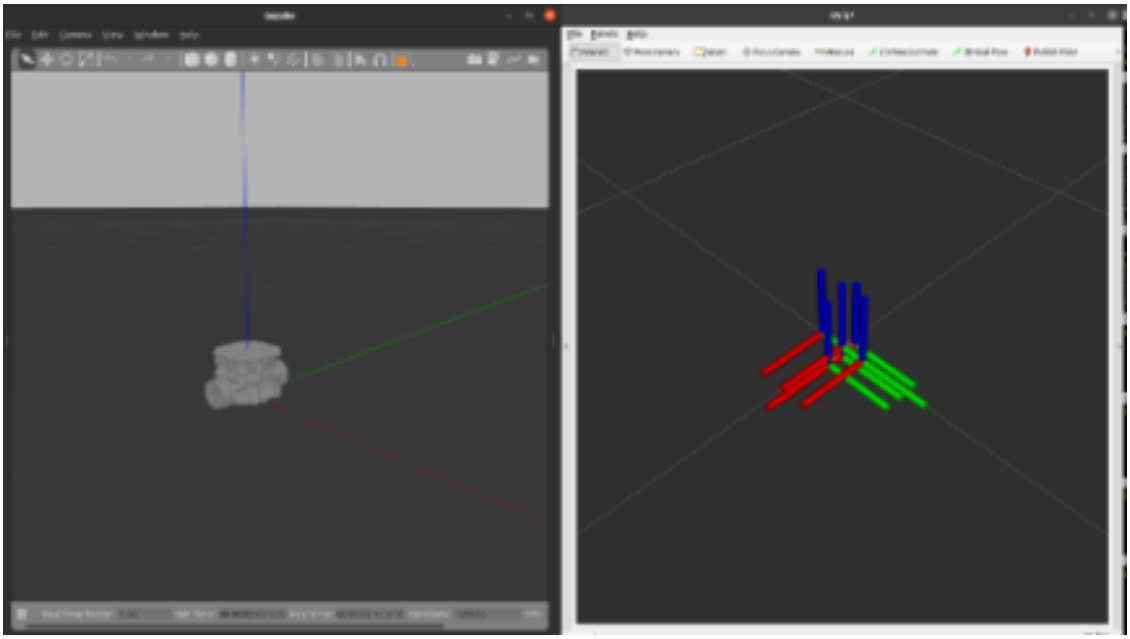


Figure 3.9: Real-time visualization of robot in Gazebo.

pects of the RViz interface was the real-time display of sensor data. The laser scan display provided a live feed of LIDAR readings, rendering the 2D scan as a radial sweep of colored points around the robot. This visualization allowed the developer to observe how the robot perceived its surroundings—walls, obstacles, and open spaces were clearly visible, even in the absence of physical hardware. In parallel, the Odometry and IMU displays showed the robot's pose and orientation, both critical for diagnosing localization performance.

Localization itself was verified using RViz by overlaying the robot's estimated pose on a known map. The AMCL particle cloud was rendered to show how the robot refined its position estimate over time. Initially dispersed, the particles gradually converged as sensor data accumulated, indicating that the robot was successfully localizing itself in the environment. This process was especially useful for debugging during the hardware transition phase, where issues such as map misalignment or inconsistent sensor data could cause localization failure.

Navigation feedback was also displayed in RViz. Once a navigation goal was issued—either from the Nav2 Goal tool in RViz or programmatically—the global planner's output was rendered as a blue path on the map, representing the planned trajectory to the target location. The local planner's real-time adjustments were visualized as shorter trajectory updates, often bending around nearby obstacles. These overlays provided immediate insight into whether

the planner was operating correctly and how it was adapting to changes in the environment.

Costmaps were another essential visualization tool. The global costmap outlined known obstacles and boundaries in the static map, while the local costmap captured dynamic obstacles detected through LIDAR in the robot's vicinity. Both were displayed as translucent grids with color-coded costs, offering a clear understanding of how the robot was reasoning about its safety buffer and obstacle avoidance. This was especially valuable during fine-tuning of inflation radius, obstacle persistence, and recovery behaviors.

In RViz, custom configurations were saved as .rviz files, which preserved all display settings, fixed frames, color schemes, and topic subscriptions. These configurations allowed for quick reloading of the complete visualization environment across development sessions. Whether observing the robot in Gazebo or monitoring its operation on the physical floor, the same RViz setup provided consistent and reliable feedback.

The ability to visually confirm every component of the system—robot pose, velocity, sensor coverage, path planning, and obstacle mapping—greatly accelerated development. It not only made debugging intuitive but also enabled confidence in system behavior, especially during transitions between simulation and real-world deployment. RViz was more than just a debugging tool; it became an integral part of the development pipeline, bridging the gap between abstract system data and real-world robotic perception.

3.2.4 TF Tree Management

One of the foundational elements of the robot's software architecture is the Transform (TF) tree, which defines the spatial relationships between various coordinate frames in the system. TF management in ROS2 is handled through the tf2 library, which provides real-time, time-stamped transforms between different frames of reference. For an autonomous robot that relies heavily on accurate localization, sensor fusion, and navigation, maintaining a coherent and correctly broadcast TF tree is essential.

The base of the transform tree begins at the map frame, which represents a fixed, world-aligned coordinate system used by the localization system. From this static frame, a dynamic transform is broadcast to the odom frame, which tracks the robot's motion over time using odometry. This transform is typically maintained by the Extended Kalman Filter (EKF) or directly by the navigation stack's localizers. The odom to base_link transform represents the robot's actual movement through space as estimated by wheel encoders and IMU data.

The base_link frame serves as the central reference for the robot body, located at the geometric center of the robot. All sensor and actuator frames are positioned relative to this. For instance, the lidar_frame is mounted forward and slightly elevated, representing the position of the LIDAR scanner. Similarly, imu_frame is typically placed near the center of mass to minimize rotational bias. Other frames, such as those for cliff sensors or bumpers, are added to represent their physical locations on the chassis.

RViz was instrumental in verifying that all transforms were being broadcast correctly and updated in real time. During development, any break in the TF tree—such as a missing static transform or a misconfigured child-parent relationship—would be immediately flagged in the RViz TF display. These issues often led to downstream failures in localization, sensor alignment, or navigation planning, and were resolved by systematically inspecting and adjusting the robot's URDF and node configurations.

An example of the final TF tree for the robot looks like a well-structured hierarchy rooted at map, branching down through odom, base_link, and outward to each sensor frame. The robot_state_publisher node, launched with the robot's URDF, was responsible for continuously publishing joint states and broadcasting corresponding transforms, allowing all nodes to reference sensor data within a consistent coordinate framework.

This consistent and complete TF tree enabled accurate integration of sensor data, ensured alignment between the robot's physical and perceived state, and allowed critical packages like AMCL and Nav2 to perform reliably during autonomous operations.

3.2.5 Topic Graph Analysis and System Communication

In addition to the spatial coordination provided by the TF tree, the robot's internal communication relies on a well-structured ROS2 topic graph. This graph illustrates the flow of data between nodes—sensors, controllers, perception algorithms, and planners—via ROS2's publisher-subscriber model. The topic graph not only outlines how data moves through the system but also reveals interdependencies and bandwidth utilization, which are crucial for maintaining real-time performance.

During development, the ROS2 CLI tool (ros2 topic list, ros2 topic info, and ros2 node info) was used frequently to inspect active topics, check message types, and identify which nodes were publishing or subscribing to which streams. The most critical topics included /scan for LIDAR data, /odom for odometry, /imu/data for inertial measurements, and /cmd_vel for velocity commands. These topics formed the core feedback-control loop that enabled the robot to perceive and respond to its environment.

Sensor drivers or Gazebo plugins published data on their respective topics at predefined frequencies. For example, the LIDAR published sensor_msgs/LaserScan messages at around 10 Hz, which were consumed by both the mapping and obstacle avoidance nodes. Odometry data, typically fused with IMU readings using a filter node such as robot_localization, was published on the /odom topic and served as the basis for both navigation and localization.

The Nav2 stack introduced additional topic traffic, including planned paths on /plan, costmaps on /global_costmap/costmap and /local_costmap/costmap. When a navigation goal was issued—either manually via RViz or programmatically—the appropriate nodes published the target to the navigation planner, which in turn published a path and continuous velocity commands until the goal was reached.

ROS2's DDS-based communication enabled scalable and efficient message exchange, even with the high data volume produced by sensors. However, real-time observability of this data flow was important to prevent issues such as topic congestion, message drops, or incorrect Quality of Service (QoS) settings. Tools like rqt_graph were used to generate visual representations of the node-topic relationships, highlighting all publishers and subscribers and ensuring that no node was isolated or improperly configured.

In the final system, the topic graph exhibited a clean, modular structure: perception nodes consumed raw sensor data and published refined observations; localization nodes processed this information and updated the robot's pose estimate; navigation nodes consumed the pose and environmental data to generate motion plans; and control nodes translated those plans into actuator commands. Each module was encapsulated and communicated

only through well-defined topics, preserving modularity and simplifying future upgrades or debugging.

By maintaining a robust topic graph and actively monitoring its integrity throughout development, the software system achieved the reliability, responsiveness, and maintainability necessary for a fully autonomous room-cleaning robot.

3.2.6 Localization and Mapping

In autonomous robotics, the challenge of localization involves determining the precise position and orientation (pose) of the robot within a given environment, while mapping involves creating a representation (map) of the environment itself. These two tasks, localization and mapping, are intrinsically linked and are typically solved concurrently in a process known as Simultaneous Localization and Mapping (SLAM). This section outlines the core principles, algorithms, and techniques used to achieve robust localization and mapping.

3.2.6.1 Simultaneous Localization and Mapping (SLAM)

SLAM is a critical capability for autonomous robots, allowing them to navigate environments without relying on external positioning systems (e.g., GPS). The fundamental goal of SLAM is to build a map of the robot's environment while simultaneously keeping track of the robot's location within that map.

Sensor Inputs for SLAM: In the current system, LIDAR is the primary sensor for SLAM, providing accurate and dense range data that can be used to detect features in the environment. While visual SLAM (using cameras) is also common, LIDAR-based SLAM tends to provide more reliable performance in environments with limited or no visual cues (e.g., in low light or fog).

SLAM Algorithm Choice: In ROS 2, the `slam_toolbox` package is widely used for LIDAR-based SLAM. This package implements robust algorithms for both front-end (sensor data processing, feature extraction) and back-end (map optimization) operations. The core SLAM algorithm used in this system is a variant of the FastSLAM algorithm, which uses particle filters to simultaneously estimate the robot's trajectory and map.

FastSLAM Overview: FastSLAM operates by maintaining a set of particles, each representing a possible robot trajectory and map. Each particle is updated based on sensor measurements and control inputs. The particle filter allows for robust localization in large, cluttered environments with noisy sensors.

Map Representation: The resulting map from SLAM is typically represented as a grid map (often referred to as an occupancy grid), where each cell in the grid is assigned a probability value indicating whether that region is occupied (i.e., contains an obstacle). This representation can be used for path planning, obstacle avoidance, and navigation tasks.

Performance Considerations: One of the key challenges in SLAM is ensuring that it runs in real-time, particularly for large environments. This requires optimizing the particle filter's resampling step and managing the size of the map. Efficient data structures such as octomaps (3D grid maps) or costmaps (2D occupancy grids) are often used to ensure fast processing and minimal memory usage.

Figure 3.10: Real-time visualization of robot mapping in Gazebo.

3.2.6.2 Adaptive Monte Carlo Localization (AMCL)

Once a map is created using SLAM, the next task is localization — determining the robot's pose (position and orientation) within the map. AMCL is a well-known probabilistic approach to solving the localization problem. It is an extension of Monte Carlo Localization (MCL), and it adapts the particle filter technique to provide real-time, reliable localization in dynamic environments.

How AMCL Works: AMCL uses a particle filter to maintain a set of hypotheses about the robot's pose. Each particle represents a potential position and orientation in the map, and the filter continuously updates these particles based on sensor data (e.g., LIDAR scans). The filter performs the following steps: 1. Prediction Step: Given the robot's previous state and control inputs (e.g., velocity, rotation), each particle is moved according to the robot's motion model. 2. Correction Step: LIDAR data is used to correct the robot's pose estimate by comparing the sensor readings to the map. The likelihood of each particle is calculated based on how well it matches the expected sensor reading from the map. 3. Resampling Step: Particles that are unlikely to represent the correct pose are discarded, while particles that are consistent with the sensor readings are given higher weight. This allows the filter to focus computational resources on more promising hypotheses.

Resampling and Efficiency: The resampling step is crucial for AMCL's performance. If there is a high variance in particle distribution (i.e., if most particles are unlikely to represent the robot's true location), a resampling step is performed to create a new set of particles, with more weight given to those that are closer to the actual robot position.

AMCL Parameters: Several parameters in AMCL can be tuned to improve its accuracy and speed. These include: - Particle count: The number of particles used to represent the robot's state. Higher particle counts increase accuracy but also computational cost. - Laser scan likelihood model: This model defines how the LIDAR scan is compared to the map. The model must balance between computational cost and precision in interpreting sensor data. -

Movement model noise: Adjusting how much uncertainty is assumed in the robot's movement can impact localization accuracy, especially in uncertain or dynamic environments.

Localization Update Rate: The update rate of AMCL should be tuned based on the robot's motion and the complexity of the environment. In highly dynamic environments (e.g., where obstacles are frequently moving), frequent localization updates are required to maintain accuracy. In contrast, static environments may require less frequent updates.

3.2.7 Sensor Fusion in Localization

To enhance the accuracy and robustness of the localization process, sensor fusion techniques are employed to combine data from multiple sensors. As previously discussed, the EKF (Extended Kalman Filter) is used to combine sensor measurements from the LIDAR, IMU, and wheel encoders, providing a more accurate estimate of the robot's pose.

EKF for Localization: The EKF corrects the robot's pose by fusing information from multiple sources, including wheel encoders, LIDAR, and the IMU. This allows the system to mitigate errors introduced by any single sensor, such as odometric drift or noisy sensor readings.

IMU Data for Motion Correction: The IMU provides crucial information for correcting the robot's orientation during rapid movements, where LIDAR and wheel encoders may not provide accurate estimates. By fusing IMU data with odometry and LIDAR readings in the EKF, the robot can maintain stable and reliable localization even when encountering dynamic environments.

Handling Dynamic Environments: In highly dynamic environments with moving objects, it is essential to ensure that the localization algorithm can adapt. The use of dynamic obstacle avoidance and dynamic costmaps in combination with SLAM ensures that the robot can update its understanding of the environment and adapt its path planning in real time.

3.2.7.1 Mapping with Occupancy Grid and Costmap

During the mapping process, the robot builds an occupancy grid , a 2D representation of the environment where each cell contains the probability of whether that cell is occupied or free. This grid is continuously updated as the robot explores its surroundings and gathers sensor data.

Costmap Layers: A costmap is a type of occupancy grid that adds additional layers of information to assist in navigation. For example: - Static obstacles (from SLAM) are marked as high-cost areas. - Dynamic obstacles (detected by LIDAR or other sensors) are continuously updated, with moving obstacles being given temporary high-cost values.

The inflation layer is another important component of the costmap, which provides a buffer zone around obstacles. This buffer ensures that the robot maintains a safe distance from obstacles, improving safety during navigation.

Costmap Integration in Nav2: The ROS 2 Navigation Stack (Nav2) uses the costmap to inform the path planning and local planning algorithms. The local planner takes into account both the global costmap (for large-scale obstacle avoidance) and the local costmap (for real-time obstacle detection and avoidance).

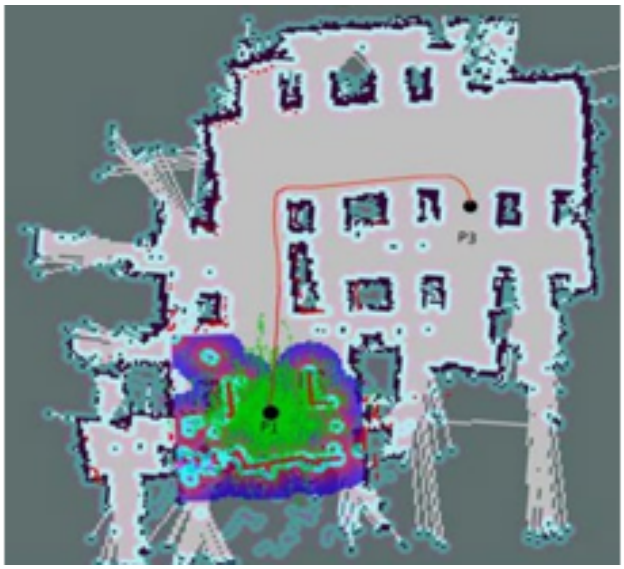


Figure 3.11: Real-time visualization of cost map.

3.2.7.2 Optimizing Localization and Mapping

The performance of localization and mapping algorithms can be greatly enhanced through optimization techniques. These include: - Multi-Resolution Maps: Using maps with varying resolution can reduce the computational load when exploring large areas, while providing high precision when needed in smaller, more detailed areas. - Loop Closure: In large environments, the robot may return to previously visited areas. Loop closure techniques help correct accumulated errors by recognizing previously mapped locations and adjusting the map to align with the current position.

Graph-Based SLAM Optimization: Graph-based SLAM is an optimization technique that represents the robot's trajectory and the map as a graph. Each node represents a pose, and each edge represents a constraint between two poses (such as the relative distance between two robot poses or the distance between the robot and an obstacle). The optimization problem then minimizes the overall error in the graph, refining both the robot's trajectory and the map.

3.3 Sensor Integration and Fusion

In autonomous robotics, sensor integration and fusion are vital to achieve accurate localization and environmental awareness. The robot environment is perceived using various sensors, and these sensors provide complementary information that must be effectively combined to produce accurate pose estimates and reliable map building. This section discusses the sensors used in this project, the integration process, the hardware connections, and the application of the Extended Kalman Filter (EKF) for sensor fusion.

The key sensors used in the system include a 2D Light Detection and Ranging (LIDAR), wheel encoders, and an Inertial Measurement Unit (IMU). Each sensor provides unique data, and the fusion of these data leads to a more accurate understanding of the robot's position

and motion in a dynamic environment.

3.3.1 LIDAR Integration for Environmental Perception

LIDAR (Light Detection and Ranging) is a laser-based sensor that generates a 2D point cloud to represent the environment. By emitting laser beams and measuring the time it takes for the light to reflect back after striking an object, the LIDAR produces a detailed scan of the surrounding space. This scan helps the robot detect obstacles and navigate around them. In this project, the 2D LIDAR sensor is mounted on the robot to capture 360-degree scans of the environment.

Hardware Connection: The LIDAR sensor is connected via a serial interface or via Ethernet, depending on the specific LIDAR model used. For this project, a common 2D LIDAR, such as the RPLIDAR A1, is used, which communicates with the computer or robot controller through USB. The sensor is mounted on the robot's chassis to ensure it provides accurate readings of the environment.

Software Integration: The LIDAR sensor is integrated into ROS 2 by using the `rplidar_ros` package, which converts the LIDAR data into a ROS-compatible format. The raw data from the LIDAR is published on the `/scan` topic, which is then consumed by both the AMCL localization algorithm and the costmap generation node.

Role in Localization and Mapping: LIDAR data is essential for localization via AMCL and for building the robot's map. In AMCL, LIDAR scans are used to match the current robot pose with pre-existing map data, correcting any drift in the odometry and improving the accuracy of the robot's pose. Furthermore, LIDAR data contributes to the dynamic costmap, helping the robot avoid collisions and navigate effectively.

3.3.2 Wheel Encoders for Odometry Estimation

Wheel encoders are used to measure the robot's wheel rotations, providing relative displacement data. Each encoder measures the number of wheel rotations or "ticks," which is then used to estimate the distance traveled and the robot's heading.

Hardware Connection: The wheel encoders are typically connected to the microcontroller or robot's embedded system using GPIO (General Purpose Input Output) pins. Each encoder generates electrical pulses corresponding to the wheel's rotation, and the microcontroller reads these pulses to calculate the robot's movement.

Software Integration: The wheel encoder data is processed by the ROS 2 odometry node, which computes the robot's position in the `odom` frame. This information is published on the `/odom` topic and serves as the foundation for the robot's movement estimation. Wheel encoders are critical for maintaining continuous tracking of the robot's displacement, particularly when other sensors might be temporarily unavailable or unreliable.

Role in Localization: While wheel odometry is accurate over short distances, it tends to drift over time due to factors such as wheel slippage, uneven terrain, or minor inaccuracies in wheel alignment. Thus, wheel encoders alone cannot provide highly accurate long-term localization, but they serve as a crucial component for estimating relative motion between localization updates.

3.3.3 IMU for Orientation and Velocity Estimation

An Inertial Measurement Unit (IMU) consists of an accelerometer and a gyroscope, which measure linear acceleration and angular velocity, respectively. The IMU provides precise information about the robot's orientation and velocity, especially when rapid movements occur, or when the robot experiences sudden changes in direction.

Hardware Connection: The IMU is typically connected to the robot's controller through an I2C or SPI communication interface. The IMU continuously outputs data regarding acceleration and angular velocity, which is processed by the robot's controller.

Software Integration: The IMU data is integrated into ROS 2 through the `imu_ros` driver, publishing data on the `/imu` topic. The IMU provides real-time updates of the robot's velocity, orientation (in quaternion format), and acceleration, which are critical for improving the robot's pose estimation during high-speed maneuvers.

Role in Localization: The IMU aids in correcting small errors in wheel odometry, such as drift, by providing high-frequency orientation and velocity updates. This makes it especially useful for mitigating drift during rapid accelerations or decelerations when wheel encoders may not be able to update the pose effectively.

3.3.4 Sensor Fusion using Extended Kalman Filter (EKF)

While individual sensors like wheel encoders, IMU, and LIDAR provide valuable data, their information must be fused together to produce an accurate estimate of the robot's state (pose and velocity). This is achieved using a sensor fusion technique known as the Extended Kalman Filter (EKF) .

EKF Basics: The Extended Kalman Filter is an algorithm that fuses data from multiple sensors by predicting the robot's state based on a motion model and correcting it using sensor measurements. The EKF operates in two steps: 1. Prediction Step: The EKF predicts the state of the robot based on the motion model and the previous state. 2. Correction Step: The EKF compares the predicted state to the actual sensor measurements and adjusts the predicted state to minimize the difference.

In this project, the EKF is implemented using the `robot_localization` package in ROS 2. The EKF node receives input from wheel encoders, IMU, and LIDAR sensors, each contributing a measurement of the robot's position and velocity. The EKF continuously updates the robot's state by comparing predicted motion from the wheel encoders with real-time corrections from the IMU and LIDAR data.

Algorithm Details:

- The EKF assumes that both the process model (predicting motion) and measurement model (sensor readings) are nonlinear, which is why it is considered an "extended" version of the standard Kalman filter.
- The state vector in the EKF may include position, velocity, and orientation (e.g., $[x, y, \theta, v_x, v_y]$), where x, y represent position, θ represents orientation, and v_x, v_y represent velocity components.
- The system is updated with each sensor reading to maintain an accurate estimate of the robot's state.

The EKF essentially combines the strengths of each sensor: wheel odometry provides reliable data for small movements, IMU corrects for drift in orientation, and LIDAR helps reset the robot's global position by matching scans to a pre-built map.

3.3.5 Sensor Fusion with EKF (Extended Kalman Filter)

In the project, the Extended Kalman Filter (EKF) was used for sensor fusion. EKF is a popular choice for combining multiple sources of sensor data, providing an estimate of the robot's position and orientation (pose) that accounts for uncertainties in both the sensors and the robot's motion model.

The EKF operates as follows: 1. Prediction Step : Based on the robot's motion model, the filter predicts the robot's state (position, velocity, and orientation) at the next time step. This prediction is based on the control inputs (e.g., motor commands) and the current state estimate.

2. Update Step : The filter then compares the predicted state with the actual sensor measurements (e.g., from LIDAR, IMU, or odometry). The measurement model computes the expected sensor readings based on the predicted state. The difference between the predicted and measured values is used to update the state estimate, improving its accuracy.

The EKF operates on the following mathematical framework:

$$\mathbf{x}_{k|k-1} = \mathbf{F}\mathbf{x}_{k-1|k-1} + \mathbf{B}\mathbf{u}_k$$

Where: - $\mathbf{x}_{k|k-1}$ is the predicted state at time k. - $\mathbf{F}$ is the state transition matrix. - $\mathbf{B}$ is the control input matrix. - $\mathbf{u}_k$ is the control input at time k.

The filter then uses the Kalman Gain to update the state estimate:

$$\mathbf{K} = \mathbf{P}_{k|k-1}\mathbf{H}^T(\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^T + \mathbf{R})^{-1}$$

Where: - $\mathbf{K}$ is the Kalman Gain. - $\mathbf{H}$ is the measurement matrix. - $\mathbf{P}_{k|k-1}$ is the predicted covariance matrix. - $\mathbf{R}$ is the measurement noise covariance.

The EKF effectively combines information from multiple sensors, such as odometry (which is prone to drift) and LIDAR (which provides accurate spatial measurements), to produce a more accurate estimate of the robot's position.

3.3.6 Extended Kalman Filter (EKF) for Nonlinear Systems

For more complex and nonlinear systems, Extended Kalman Filter (EKF) is used. The EKF handles nonlinearities by linearizing the system at each step using a first-order Taylor expansion . For instance, in the case of robot localization, the nonlinear relationship between the robot's position and the sensor readings is approximated by the Jacobian of the measurement model.

The EKF update equations for nonlinear systems are:

$$\mathbf{x}_{k|k-1} = f(\mathbf{x}_{k-1|k-1}, \mathbf{u}_k)$$

$$\mathbf{y}_k = h(\mathbf{x}_{k|k-1}) + \mathbf{w}_k$$

Where: - $f(\mathbf{x}_{k-1|k-1}, \mathbf{u}_k)$ is the nonlinear process model. - $h(\mathbf{x}_{k|k-1})$ is the nonlinear measurement model. - $\mathbf{w}_k$ represents the measurement noise.

EKF ensures that the robot can effectively fuse sensor data, even when the sensor readings are noisy or the robot is subject to complex dynamics. This makes it a powerful tool for real-time navigation in uncertain environments.

3.3.7 Sensor Fusion Performance and Reliability

In practice, sensor fusion is not perfect, and the reliability of the fused state estimate depends on the quality of the individual sensors and the accuracy of the motion model. Sensor noise, data synchronization issues, and errors in the robot's motion model can affect the performance of the EKF.

To ensure reliable fusion: 1. The noise characteristics of each sensor (e.g., IMU noise, wheel encoder error) are modeled and incorporated into the EKF as covariance matrices. Proper tuning of these matrices is essential to balancing the contributions of each sensor. 2. The EKF should be tested and adjusted regularly, especially in dynamic environments, to avoid performance degradation due to incorrect sensor calibration or parameter settings.

By monitoring the robot's pose estimates and error covariance, it is possible to detect and correct issues early in the fusion process, ensuring the robot's motion remains accurate and stable throughout its navigation tasks.

Tools for Debugging and Validation: - `rqt_plot`: Used to visualize the estimated robot pose and sensor data in real-time. - `rviz`: A powerful visualization tool for displaying the robot's environment, sensor data, and trajectory, helping validate the robot's localization.

Through careful integration and fusion of sensor data, the robot can achieve accurate localization, enabling reliable autonomous navigation in complex and dynamic environments.

3.3.8 Sensor Fusion in Localization

To enhance the accuracy and robustness of the localization process, sensor fusion techniques are employed to combine data from multiple sensors. As previously discussed, the EKF (Extended Kalman Filter) is used to combine sensor measurements from the LIDAR, IMU, and wheel encoders, providing a more accurate estimate of the robot's pose.

EKF for Localization: The EKF corrects the robot's pose by fusing information from multiple sources, including wheel encoders, LIDAR, and the IMU. This allows the system to mitigate errors introduced by any single sensor, such as odometric drift or noisy sensor readings.

IMU Data for Motion Correction: The IMU provides crucial information for correcting the robot's orientation during rapid movements, where LIDAR and wheel encoders may not provide accurate estimates. By fusing IMU data with odometry and LIDAR readings in the EKF, the robot can maintain stable and reliable localization even when encountering dynamic environments.

Handling Dynamic Environments: In highly dynamic environments with moving objects, it is essential to ensure that the localization algorithm can adapt. The use of dynamic obstacle avoidance and dynamic costmaps in combination with SLAM ensures that

the robot can update its understanding of the environment and adapt its path planning in real time.

3.4 Path Planning and Navigation

Path planning is a key component of any autonomous robot, enabling it to navigate through an environment while avoiding obstacles and adhering to safety constraints. In this project, the robot utilizes the ROS 2 Navigation Stack (Nav2) to autonomously plan and execute its movements. The path planning process can be broken down into several phases, including global path planning, local path planning, and recovery behaviors. This section provides a detailed explanation of the algorithms and strategies employed in each phase.

3.4.1 Global Path Planning

Global path planning involves generating a feasible path from the robot's starting position to the goal location. This is typically a high-level task that involves searching for a path through the entire environment, taking into account both static and dynamic obstacles.

Costmap Representation: The global path planner relies on the environment's costmap, which is created through localization and mapping. The costmap represents the environment as a grid, where each cell has a cost associated with it based on the presence of obstacles or other factors like distance to known obstacles. The costmap is continuously updated, and the planner uses this updated map to plan a path to the goal.

Algorithms for Global Path Planning: The two primary algorithms used in the global planner are A (A-star) and Dijkstra's Algorithm :

A Algorithm: The A algorithm is a widely used algorithm in robotics for path planning, combining aspects of both the breadth-first search and greedy algorithms. It minimizes the cost function $f(n)$ for each node, where:

$$f(n) = g(n) + h(n)$$

Where: - $g(n)$ is the actual cost to reach the node n from the start node. - $h(n)$ is the heuristic cost estimate from node n to the goal node. - $f(n)$ is the total cost of reaching the node n.

The heuristic function $h(n)$ can be designed based on different types of environmental information, with the most common being the Euclidean distance, Manhattan distance, or Diagonal distance, depending on the environment.

The A algorithm works by starting at the start node and expanding outward, selecting the node with the lowest $f(n)$ value at each step. The process continues until the goal node is reached. This algorithm ensures the optimal path by considering both the shortest path to the goal and the heuristic to guide the search.

Dijkstra's Algorithm: Dijkstra's algorithm is another fundamental algorithm for pathfinding, known for guaranteeing the shortest path in weighted graphs. Unlike A , Dijkstra's algorithm does not use a heuristic and instead evaluates all possible paths to the goal by considering the actual cost to reach each node:

$$f(n) = g(n)$$

Where $g(n)$ is the actual cost from the start node to node n. The algorithm works by expanding the nodes in order of increasing cost, guaranteeing that the first time a node is reached, it is via the shortest possible path.

Though it guarantees an optimal path, Dijkstra's algorithm is more computationally expensive than A , especially in larger environments, as it evaluates all possible paths.

Advanced Global Planners: In addition to A and Dijkstra, more advanced planners have been integrated into the ROS 2 Navigation Stack. One such planner is the Smac Planner , which is designed to work with robots that have non-holonomic constraints (i.e., robots that cannot move in arbitrary directions, such as differential-drive robots). The Smac Planner uses search-based techniques and kinematic feasibility, which makes it ideal for real-time planning in complex environments.

3.4.2 Local Path Planning and Obstacle Avoidance

Once the global path is generated, the robot must navigate through it while avoiding obstacles and reacting to dynamic changes in the environment. This is where local path planning comes into play.

Dynamic Window Approach (DWA): The Dynamic Window Approach (DWA) is a local planner that generates velocity commands for a robot while considering both kinematic constraints and obstacle avoidance. The algorithm evaluates a set of possible velocities v and angular velocities ω, and the objective is to select the velocity pair that maximizes a cost function that balances: - Obstacle avoidance, - Progress toward the goal, and - Adherence to the global path.

The DWA algorithm uses the robot's dynamic constraints (i.e., maximum velocity, acceleration, and turning rate) to calculate a set of candidate velocities. Then, it evaluates each candidate using the following cost function:

$$J(v, \omega) = \alpha \cdot J_{\text{goal}} + \beta \cdot J_{\text{obstacle}} + \gamma \cdot J_{\text{dynamics}}$$

Where: - J_{goal} evaluates how well the candidate velocity moves the robot toward the goal. - J_{obstacle} evaluates how well the candidate velocity avoids obstacles. - J_{dynamics} penalizes velocities that violate the robot's kinematic constraints. - α, β, and γ are weight factors that can be adjusted to prioritize different aspects of the navigation task.

Time Elastic Band (TEB) Planner: Another local planner used in this project is the Time Elastic Band (TEB) planner. The TEB planner optimizes the robot's trajectory over time, minimizing a cost function that accounts for both spatial and temporal constraints. The trajectory is modeled as an elastic band of waypoints, where the cost function is minimized to achieve a smooth trajectory that avoids obstacles while reaching the goal.

The TEB planner uses a sequence of waypoints $\{(x_i, y_i, \theta_i)\}$ and iteratively optimizes their positions using the following objective function:

$$\mathcal{J} = \sum_{i=1}^{n-1} \left(\lambda_1 \|(x_i, y_i) - (x_{i+1}, y_{i+1})\|^2 + \lambda_2 \|\theta_i - \theta_{i+1}\|^2 \right)$$

Where: - λ_1 and λ_2 are weight factors that control the trajectory smoothness and heading consistency. - $\|(x_i, y_i) - (x_{i+1}, y_{i+1})\|$ is the Euclidean distance between consecutive way-

points. - $\|\theta_i - \theta_{i+1}\|$ is the angular difference between the robot's orientation at consecutive waypoints.

3.4.3 Navigation with Real-Time Feedback

A significant aspect of autonomous navigation is the ability to operate in real-time, reacting to changes in the environment. This requires continuous feedback from sensors, including LIDAR, IMU, and encoders. The feedback loop enables the robot to adjust its path in real-time, avoiding obstacles and recalculating the path if necessary.

Sensor Feedback in Navigation: The robot continuously receives sensor data, such as LIDAR scans, odometry readings, and IMU data. This information is used by the localization system to update the robot's position and orientation on the map. The sensor data is also fed into the costmap layers, which are constantly updated to reflect changes in the environment (e.g., moving obstacles).

Feedback Loop in Path Planning: The feedback loop in path planning involves multiple stages: 1. Global Path Planning: A new path is generated based on the current robot location and goal. 2. Local Path Planning: The robot follows the global path, adjusting in real-time based on sensor feedback and avoiding obstacles. 3. Recovery Behavior: If an error occurs (e.g., the robot becomes stuck or unable to continue), recovery behaviors are triggered to resolve the issue.

3.4.4 Testing and Validation of Navigation System

The robustness of the autonomous navigation system was rigorously tested in both simulated and real-world environments. In the simulation environment, tools like Gazebo were used to simulate realistic scenarios, including moving obstacles, sensor noise, and various environmental conditions. The RViz visualization tool was used to inspect the robot's trajectory, sensor data, and navigation performance.

In the real world, the robot was tested in a variety of environments, including open spaces and cluttered indoor areas. The real-time performance of the system was evaluated by measuring the robot's ability to reach its goal, avoid obstacles, and recover from errors.

3.4.5 Communication Protocol

The robot uses XRCE-DDS (eXtremely Resource-Constrained DDS) for low-latency and reliable communication between the Jetson Nano and the embedded hardware. XRCE-DDS ensures efficient data transfer, supporting real-time operations in resource-constrained environments.

Chapter 4

Experiment and Discussion

4.1 Experimental Results

This section details the results obtained from the experiments conducted on the robot during the initial phases of the project. The focus has been on achieving odometry and visualization, which are critical foundational steps for autonomous navigation. Future work will extend this foundation to implement mapping and navigation functionalities.

4.1.1 Simulation in Gazebo

The first critical step in this project was setting up the simulation in Gazebo. Gazebo is a powerful and realistic physics-based simulator for robotic systems, which provides a virtual environment where robots can be tested without the risks and costs of hardware deployment. The simulated environment was created to replicate an indoor space filled with obstacles such as walls, furniture, and other objects that would be encountered in a typical room. This setup helped to validate the robot's localization and navigation systems before moving to the real robot.

In the simulation, several sensors were integrated into the robot, including LIDAR (Light Detection and Ranging), IMU (Inertial Measurement Unit), and wheel encoders. These sensors provided essential data for localization and path planning. The LIDAR was particularly important as it continuously scanned the environment, enabling the robot to detect static and dynamic obstacles. The IMU provided information about the robot's orientation, and the odometry sensors helped estimate the robot's movement by tracking the wheel rotations.

Using AMCL (Adaptive Monte Carlo Localization), the robot's position was estimated in the pre-built map by matching the live LIDAR scans with the stored map. This allowed for real-time updating of the robot's pose as it navigated through the environment. The global planner, typically implemented using algorithms like A , was responsible for calculating the optimal path from the start position to the goal while avoiding obstacles. Once the global path was defined, the local planner, often the Dynamic Window Approach (DWA), adjusted the robot's trajectory dynamically, ensuring that it adhered to the path while avoiding any unforeseen obstacles that appeared during the robot's movement.

The simulation provided a controlled environment where we could test the path planning algorithms, the robot's ability to navigate obstacles, and the overall system performance,

offering an excellent first step before deploying the robot in the real world.

4.1.2 Visualization with RViz

RViz played a crucial role in monitoring and validating the robot's performance during the simulation. This tool allowed us to visualize the robot's sensor data, movement, and localization results in real time. It provided a 3D representation of the robot's environment and offered a comprehensive view of the robot's behavior, such as its position, velocity, and the obstacles it encountered.

The robot's state, including its pose, velocity, and sensor data, was displayed in RViz. The LIDAR data was visualized as a point cloud or laser scan, showing the range of obstacles detected by the sensor. Similarly, the costmap in RViz displayed both static and dynamic obstacles. This costmap was generated based on the LIDAR scans and was crucial for both the local planner and AMCL to function effectively. In addition, the path the robot was taking was shown as a line in the RViz window, and it was compared to the global plan generated by the planner. Any deviation from the planned path was visually evident and helped us identify potential issues in the planning or localization algorithms.

Another important feature of RViz was its ability to visualize the TF tree. The TF (Transform) system in ROS is responsible for keeping track of the spatial relationships between various parts of the robot and its environment. For example, the transformation between the robot's map frame and the odom frame, which is updated by odometry, was displayed in RViz. By observing the TF tree, we could verify if the robot's localization system was working correctly and if all sensors were properly synchronized with respect to each other.

Through RViz, we were able to verify the entire localization and path planning process in simulation, ensuring the system was functioning as expected before hardware deployment.

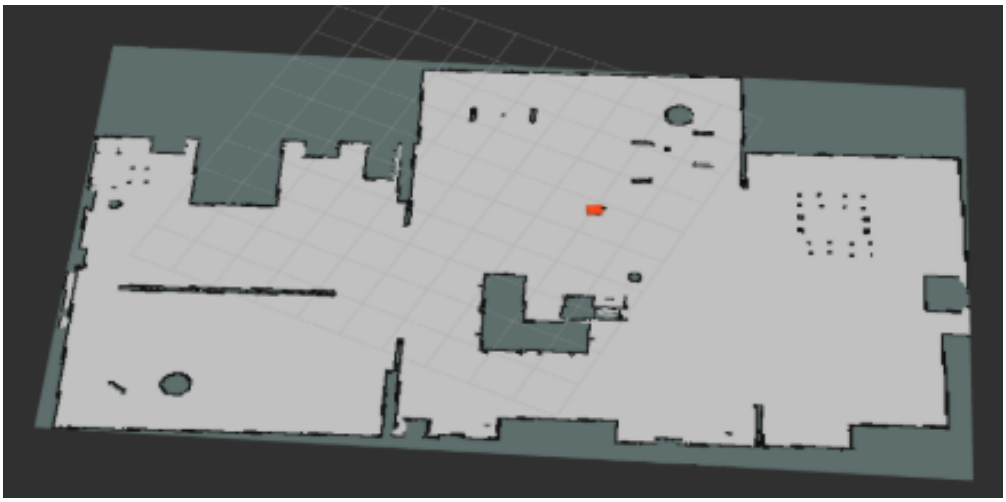


Figure 4.1: Autonomous Mapping of area.

4.1.3 Data Flow and Topic Communication

The communication between different components of the robot's system was facilitated by ROS 2 topics. ROS 2 uses a publisher-subscriber model for data exchange, where nodes can either publish data to a topic or subscribe to it. This system of message passing is key for modularity and allows for clear separation between different functions, such as localization, path planning, and actuation.

In the context of this project, there were several important topics:

/scan: Published by the LIDAR sensor, this topic carried the 2D scan data, which was used by both the AMCL localization algorithm and the costmap for obstacle detection.

/cmd_vel: This topic published the velocity commands (linear and angular) that instructed the robot's actuators to move. The Nav2 controller published these commands based on the robot's path planning and real-time obstacle avoidance.

/amcl_pose: The AMCL node continuously published the robot's estimated position in the map frame. This was critical for localization, providing the pose used by the path planner to determine the robot's next move.

/goal_pose: Used to set goal positions for the robot. When a new goal was set, the robot would start navigating toward it using the global planner and local planner.

/tf: This topic broadcasted the TF transformations, such as the relationship between odom and base_link, or between the robot's sensors and its center of rotation.

We used rqt_graph, a ROS 2 tool, to visualize the communication between the different nodes and topics. This helped ensure that the topics were correctly linked and that no messages were being lost or incorrectly transmitted. It also provided a way to diagnose potential issues in the communication flow.

4.1.4 Transition to Real-World Testing

After extensive simulation and validation in RViz and Plotjuggler, the next step was to transfer the system onto the real robot. This transition required careful setup of the robot's hardware components, such as the LIDAR, IMU, wheel encoders, and base.

The sensors were calibrated, ensuring that they provided accurate data in the real world. The AMCL localization algorithm was then applied to the robot, which began estimating its pose in the real world based on real-time sensor data. The map built earlier in simulation was used to help localize the robot in the physical space.

As the robot moved through the environment, it relied on the Nav2 stack to generate velocity commands for the actuators. These commands guided the robot along the pre-planned path, and the robot adjusted its trajectory using the local planner to avoid any dynamic obstacles.

Chapter 5

Real-World Hardware Setup and Autonomous Movement

After validating the navigation system in simulation, the focus shifted to deploying the same logic onto a real-world robotic platform. This transition required meticulous integration of sensors, actuators, embedded systems, and ROS 2 middleware on physical hardware to realize autonomous navigation. The setup involved detailed coordination between electronic components, mechanical design, and software modules, ensuring consistent system behavior in dynamic environments.

5.1 Hardware Configuration and Sensor Integration

The autonomous ground robot used in this project was designed as a compact, reliable platform capable of precise localization and navigation. Its architecture combined high-speed embedded control with robust sensor integration and a powerful computing layer, enabling real-time mobility, perception, and path planning. Each hardware module was carefully selected to address the demands of real-time responsiveness, sensor fusion, and modular scalability.

5.1.1 Microcontroller Unit (STM32F4 Series)

The STM32F4 microcontroller served as the low-level control unit, executing all time-critical tasks. It was responsible for reading encoder pulses from quadrature encoders mounted on each BLDC motor, generating PWM signals to control motor speed, sampling data from analog and digital sensors (e.g., wall and cliff sensors), and managing UART-based communication with the onboard computer.

The firmware was developed in C using STM32CubeIDE and HAL libraries. It operated as a real-time finite state machine, processing sensor inputs and executing velocity commands received from the higher-level control node. Odometry data was computed using wheel encoder values and transmitted back to the ROS 2 system at a frequency of 50–100 Hz.

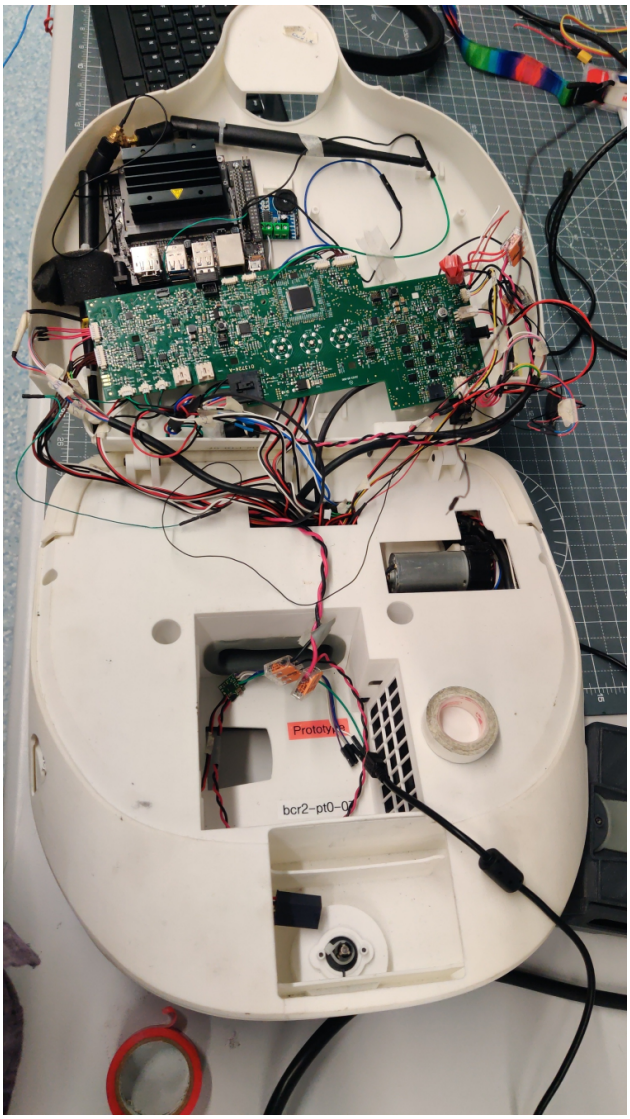


Figure 5.1: Real World hardware setup.

5.1.2 Onboard Computer (Jetson Nano)

The onboard computer was either a Raspberry Pi 4 or NVIDIA Jetson Nano running Ubuntu 20.04 with ROS 2 Foxy. This computer managed:

- Execution of the ROS 2 Navigation Stack (Nav2)
- SLAM and localization modules (e.g., `slam_toolbox`, AMCL)
- Sensor drivers and transform publishers
- Communication with the STM32 microcontroller over UART

The computer was also responsible for costmap generation and publishing of the `/cmd_vel` velocity command topic, based on planner output and fused sensor data.

5.1.3 Motor Drivers and Encoders

Brushless DC motors were driven by three-phase motor drivers that received PWM input from the STM32. Each motor included a quadrature encoder providing incremental feedback via two output channels. These signals were captured by timers configured in encoder mode.

The STM32 calculated real-time wheel displacement and velocity using:

$$x(t + \Delta t) = x(t) + v \cdot \cos(\theta) \cdot \Delta t$$
$$y(t + \Delta t) = y(t) + v \cdot \sin(\theta) \cdot \Delta t$$
$$\theta(t + \Delta t) = \theta(t) + \omega \cdot \Delta t$$

Here, v and ω denote the linear and angular velocities calculated from wheel rotations.

5.1.4 LIDAR Sensor

A 2D LIDAR sensor such as the RPLIDAR A1 was used to provide 360° scan data at 5–10 Hz. Connected via USB, it published laser scan messages to the `/scan` topic. The data was used for:

- Obstacle detection and inflation layers in costmaps
- Localization using AMCL
- SLAM for map generation

Sensor position and orientation were configured using URDF, and a static transform from `base_link` to `laser` was maintained.

5.1.5 IMU Sensor

The IMU (e.g., MPU6050 or BNO055) provided acceleration and angular velocity data. Its yaw readings were crucial during rotation, compensating for slippage in wheel odometry. The data was fused using the Extended Kalman Filter (EKF) implemented via `robot_localization`:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H\hat{x}_{k|k-1})$$

Where:

- $\hat{x}_{k|k}$ is the updated state estimate
- z_k is the current measurement (e.g., IMU reading)
- K_k is the Kalman gain
- H is the observation matrix

The EKF fused encoder-derived odometry and IMU readings to output a drift-resistant pose estimate in real time.

5.1.6 Power System

The robot was powered by a 12 V Li-ion battery pack, regulated using DC-DC buck converters to provide 5 V and 3.3 V rails for various components. High-power components like motors were isolated from logic-level circuits to reduce interference. Voltage dividers monitored the battery level, with ADC readings taken by the STM32 for safety shutdown routines.

5.1.7 Chassis and Sensor Mounting

The physical chassis was fabricated using laser-cut acrylic or aluminum profiles. It housed:

- The onboard compute unit and microcontroller
- BLDC motors with encoders
- The LIDAR sensor mounted centrally on the top
- IR-based wall and cliff sensors at the front and sides
- Bump sensors or soft mechanical switches for obstacle contact detection

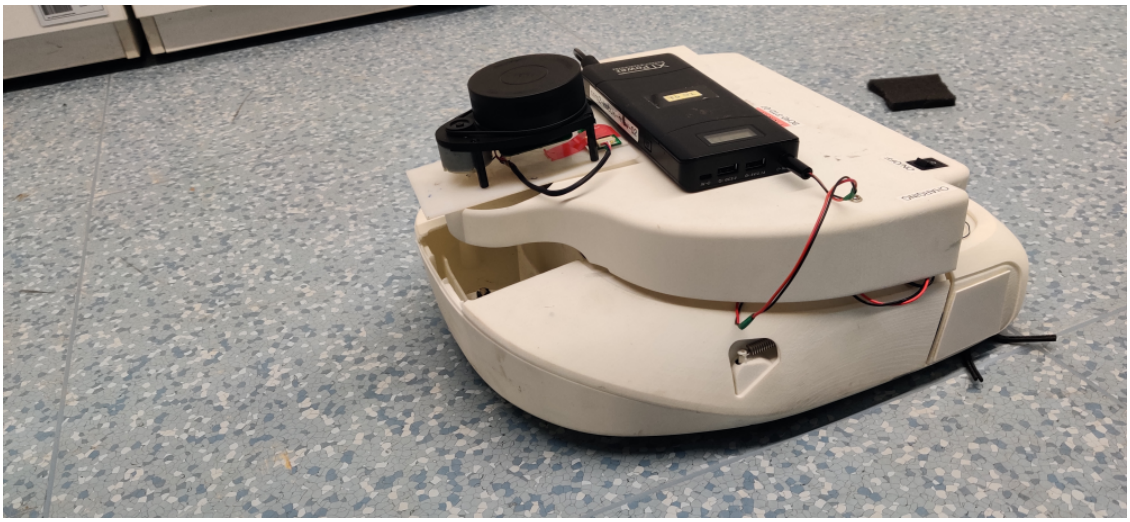


Figure 5.2: Final structure of the robot.

5.2 TF Tree and Sensor Coordination

The TF (Transform) tree in ROS 2 plays a critical role in establishing a consistent spatial relationship among various coordinate frames associated with the robot and its environment. It forms the backbone for fusing sensor data, executing localization algorithms, and enabling navigation tasks by providing a time-synchronized spatial model.

5.2.1 Coordinate Frame Hierarchy

In this project, the TF tree was organized as follows:

$$\texttt{map} \rightarrow \texttt{odom} \rightarrow \texttt{base\_link} \rightarrow \texttt{laser}$$

Each transform in this hierarchy represents a rigid body transformation between coordinate frames and is either static or dynamically computed in real time:

- map: A fixed world coordinate frame representing the known static environment.
- odom: A locally consistent but globally drifting frame, computed from wheel encoders and IMU data.
- base_link: A frame fixed at the center of the robot's chassis, representing its physical body.
- laser: Defines the position and orientation of the LIDAR sensor relative to base_link, and remains fixed.

5.2.2 Transform Sources and Update Mechanisms

The transformation between odom and base_link is calculated by the robot's low-level control unit using a combination of wheel encoder and IMU data. These inputs are fused using the Extended Kalman Filter (EKF), implemented through the robot_localization package. The EKF continuously updates the robot's pose estimate and publishes the transform using the tf2 library.

The transformation from map to odom is dynamically computed by the AMCL (Adaptive Monte Carlo Localization) algorithm. AMCL estimates this transform based on the likelihood of the current LIDAR scan relative to the prebuilt static map. This dynamic correction allows for global pose accuracy over time.

The static transformation between base_link and laser was defined in the robot's URDF (Unified Robot Description Format) file. This transform does not change at runtime and is published at startup using the robot_state_publisher node. The URDF specifies the 3D spatial offset and orientation of the LIDAR sensor with respect to the robot's base.

5.2.3 Role of the TF Tree in Localization and Navigation

For AMCL to perform accurate localization, it must relate sensor observations (e.g., LIDAR scans) to the global map. This requires consistent transforms from the laser frame to the map frame, computed as follows:

$$\texttt{laser} \rightarrow \texttt{base\_link} \rightarrow \texttt{odom} \rightarrow \texttt{map}$$

Errors in these transforms would lead to misaligned sensor data, thereby corrupting the localization estimates and navigation behavior.

During motion, the robot receives velocity commands in the base_link frame. These are integrated by the controller to update the pose in the odom frame. Meanwhile, AMCL

corrects the global pose by continuously updating the map to odom transform, maintaining consistency in a dynamic environment.

5.2.4 Visualization and Validation

To validate the TF structure, visualization tools such as rviz2 and tf2_echo were used. These tools helped verify frame connectivity and timing. The simulation environment in Gazebo was first used to verify the transform flow. After validation, the same TF configuration was deployed to the real robot, leveraging the same URDF and launch system, ensuring that the transition from simulation to hardware remained seamless.

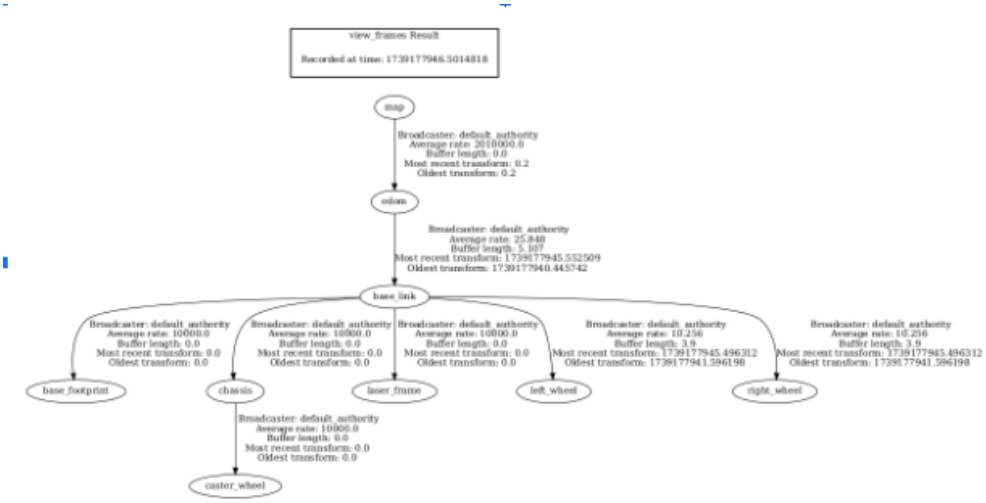


Figure 5.3: TF tree structure connecting map, odom, base_link, and laser frames

5.2.5 Impact on System Robustness

The TF tree architecture is fundamental for the system's robustness. The global planner generates paths in the map frame, while the local planner generates velocity commands in the base_link frame. Accurate and timely TF updates ensure seamless conversion of these commands into motion in the real world.

Sensor data published on topics such as /scan, pose estimates on /amcl_pose, and control commands on /cmd_vel all rely on consistent transforms to interoperate across different components. Any discontinuity in the TF tree would propagate inconsistencies throughout the system. Therefore, TF synchronization serves as the structural spine that enables successful localization, motion planning, and autonomous navigation.

5.3 Real-Time Monitoring and Feedback

To ensure the reliability, accuracy, and safety of autonomous navigation, a comprehensive real-time monitoring and feedback system was implemented during both the simulation and

hardware testing phases. This monitoring infrastructure was critical for debugging, validating algorithms, and analyzing performance metrics. Several tools were utilized, including RViz, TF tree inspection, PlotJuggler, and a custom-built GUI interface.

5.3.1 Visualization and Debugging with RViz

RViz served as the primary visualization tool during navigation. It allowed for real-time visualization of the robot's environment, perception, and state estimation. The following elements were visualized:

- The robot model (loaded from URDF), which dynamically updated its position and orientation based on estimated pose.
- The LIDAR scan data, displayed as radial lines projected into the environment, representing detected obstacles.
- The global costmap and local costmap, dynamically rendered to reflect static and dynamic obstacles.
- The global planned path (from planner server) and the real-time local trajectory (from the controller).
- The `/amcl_pose` arrow showing the robot's estimated pose and associated covariance.

In simulation (Gazebo), RViz reflected the virtual environment of a modeled indoor room, accurately mirroring sensor data and robot actions. This setup helped debug sensor placements, costmap layer configurations, and transform inconsistencies before deploying to real hardware.

5.3.2 TF Tree Consistency and Validation

The TF tree was continuously monitored using tools such as `tf2_echo` and the `TF` display in RViz. A consistent and correct TF tree was essential to ensure spatial coherence across all sensor data, control commands, and localization outputs.

The transform chain established was:

$$\texttt{map} \rightarrow \texttt{odom} \rightarrow \texttt{base\_link} \rightarrow \texttt{laser}$$

Each link in this chain represented either a dynamic transformation (e.g., odom to base_link from EKF) or a static one (e.g., base_link to laser from URDF). Any lag or missing transforms were promptly identified and resolved using TF monitoring tools. During transitions from simulation to physical robot deployment, verifying TF integrity was crucial to preserving consistent robot behavior.

5.3.3 Trajectory Analysis with PlotJuggler

PlotJuggler was employed for in-depth analysis of localization accuracy, odometry drift, and motion smoothness. Data from both `/odom` and `/amcl_pose` topics were recorded and plotted, specifically the following variables:

- x and y: Position coordinates of the robot.
- θ: Yaw angle representing orientation.
- Linear and angular velocities (from twist messages).

By comparing trajectories from odometry and AMCL estimates, discrepancies due to drift or poor particle convergence were easily identified. This analysis informed tuning of the AMCL parameters, EKF configuration, and controller gains.

Moreover, it helped verify that the navigation stack could recover from minor localization errors and maintain a smooth trajectory in complex environments.

5.3.4 Custom GUI for Hardware-Level Monitoring

For the real robot, a custom GUI interface was developed to monitor hardware diagnostics in real time. This interface was implemented using Python and integrated via ROS 2 topics and services. It provided live feedback on the following parameters:

- Battery voltage and current levels.
- Motor RPMs from the BLDC motor controllers.
- Sensor diagnostics, including cliff sensors, IMU, LIDAR status, and encoder health.
- Emergency stop state and bumper status.

This hardware-layer feedback was essential during live runs, where real-world uncertainties like wheel slippage, sensor noise, or power fluctuations could otherwise go unnoticed. The GUI also displayed the robot's current velocity, command source, and localization quality, thus serving as a critical debugging and operational dashboard during autonomous navigation.

Figure 5.4: Graphical User interface

5.3.5 Validation Through Hardware Execution and Video Logging

Once all systems were verified in simulation, the same stack was deployed on the physical robot. All sensor calibrations, frame transforms, URDF parameters, and SLAM configurations were preserved. During real-world trials, video recordings captured the robot navigating autonomously in a structured indoor environment, successfully avoiding obstacles and reaching target goals. These trials verified that the localization, path planning, and control modules worked coherently in hardware, mirroring the simulation behavior.

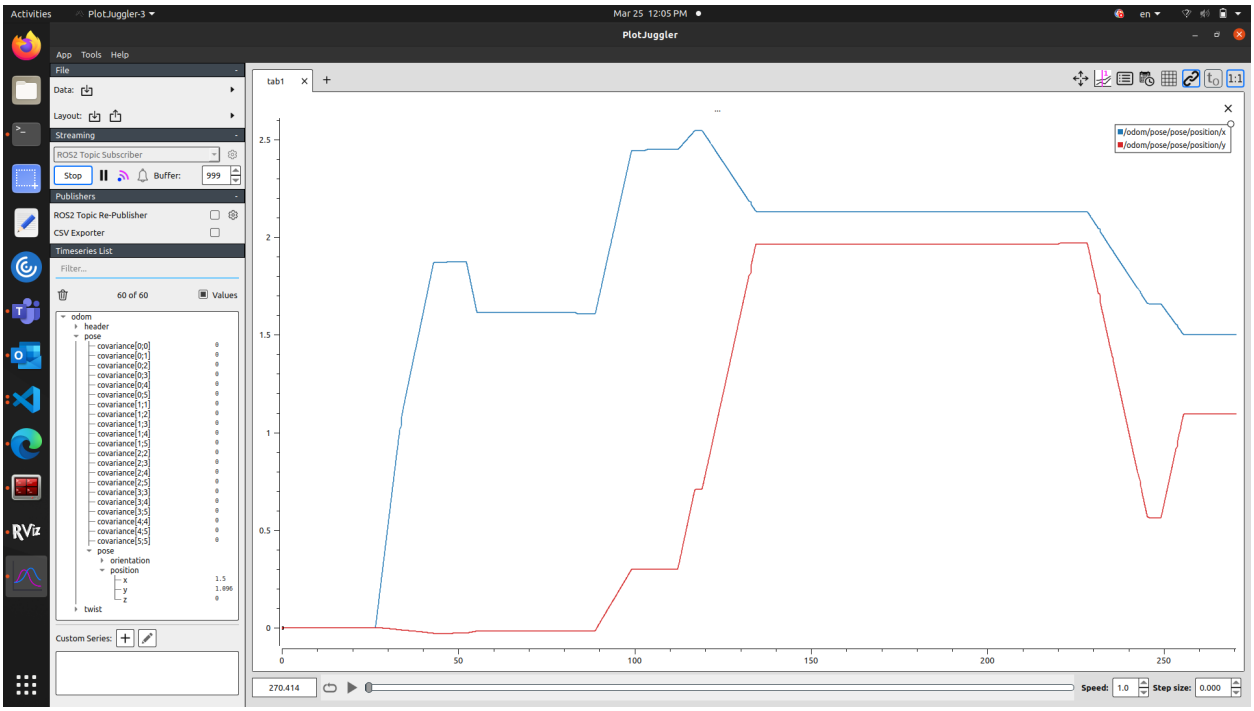


Figure 5.5: X and Y coordinate movement of robot

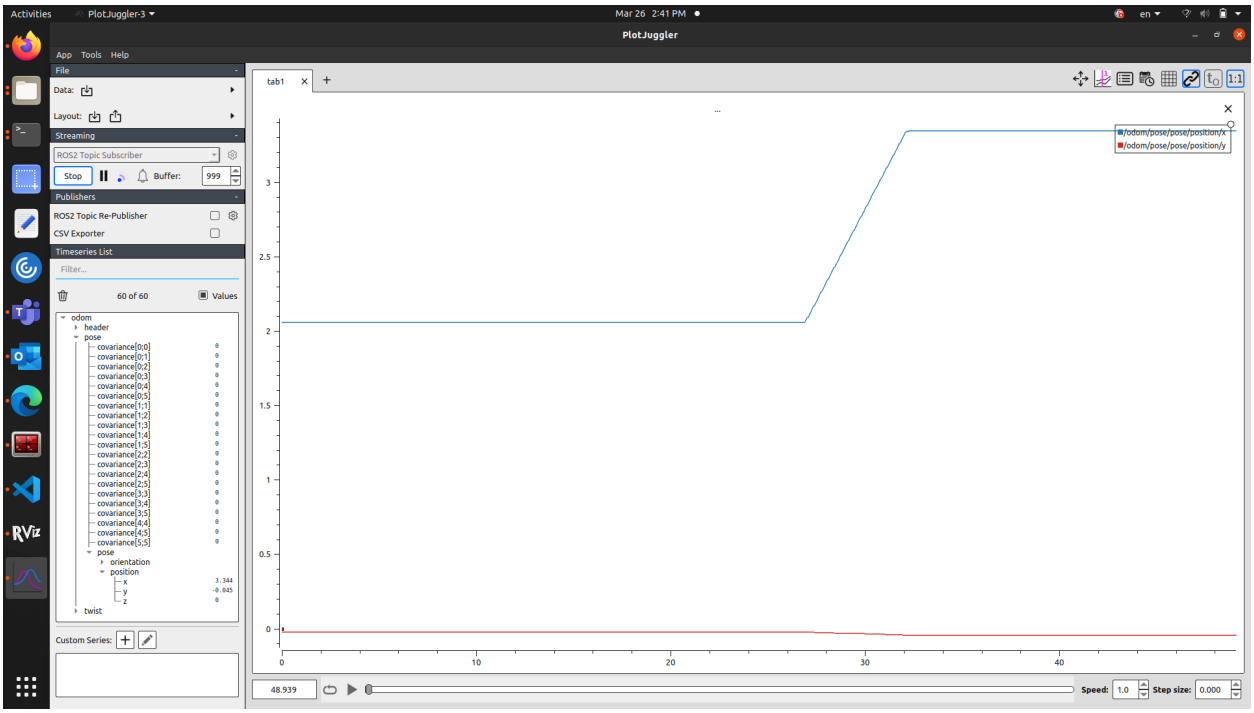


Figure 5.6: PlotJuggler visualization of the robot's x and y trajectory

These layers of monitoring created a robust feedback loop for ensuring reliable operation, early error detection, and confident deployment in real-world navigation tasks.

5.4 Experimental Results and Discussion

The experimental evaluation of the autonomous robot's localization and navigation capabilities was conducted in two phases: simulation and real-world deployment. The simulation phase provided a controlled environment to test the system's behavior, while the real-world phase validated the system's robustness and adaptability to real-world challenges. This section presents the results of these experiments, followed by a discussion on the performance, challenges, and conclusions drawn from the findings.

5.4.1 Simulation Performance Evaluation

In the simulation phase, the robot was tested in a virtual environment created using Gazebo, simulating an indoor room layout with various obstacles and features. The primary objective was to evaluate the localization accuracy, path planning, and obstacle avoidance capabilities of the system.

5.4.1.1 Localization Performance in Simulation

The robot's localization was continuously tracked using the AMCL (Adaptive Monte Carlo Localization) algorithm. The system's performance was assessed by comparing the robot's estimated pose (from `/amcl_pose`) with the ground truth position available in the simulation. The following metrics were used for evaluation:

- Positional Accuracy : The deviation between the estimated pose and the ground truth position was minimal, with the error remaining within 5-10 cm for most of the run.
- Convergence Time : The AMCL algorithm showed fast convergence to the correct pose within 2-3 seconds after moving to a new position.
- Covariance : The covariance values remained low, indicating high confidence in the localization estimates. This was crucial for the smooth operation of the navigation stack.

Figure 5.7 shows the comparison between the estimated and ground truth positions during a localization test in the simulated environment. The small positional errors observed were primarily due to sensor noise and slight inaccuracies in the map representation.

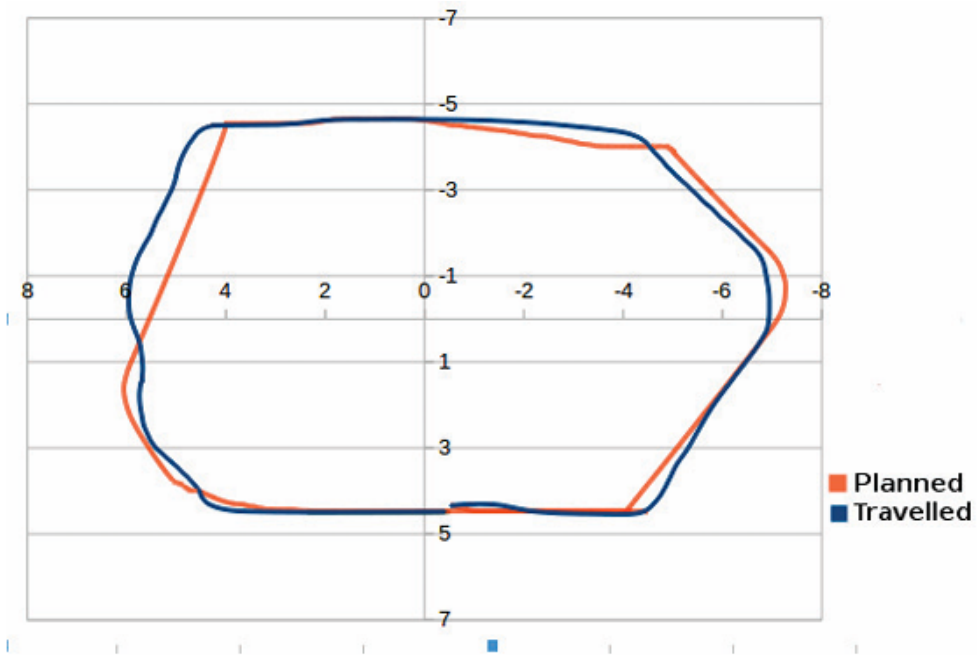


Figure 5.7: Localization performance in simulation: Estimated vs Ground Truth positions

5.4.1.2 Path Planning and Obstacle Avoidance in Simulation

The path planning component of the system was tested using the Nav2 stack, with both global and local planners. The global planner used the A algorithm , and the local planner employed the Dynamic Window Approach (DWA) .

- Global Path Planning : The robot successfully navigated through the environment, following the planned path while avoiding obstacles. In some scenarios with tight spaces, the path was dynamically adjusted, and the robot was able to make real-time corrections using the local planner.
- Obstacle Avoidance : The robot successfully avoided static and dynamic obstacles during the run. The LIDAR data was effectively integrated into the costmap, which allowed the robot to navigate around obstacles while maintaining a safe distance.
- Recovery Behaviors : In cases where the robot encountered dead-ends or complex environments, the recovery behaviors were activated, such as backing up and rotating in place , enabling the robot to find an alternative path.

Figure 5.8 shows the planned path and the robot's trajectory during an obstacle avoidance test.

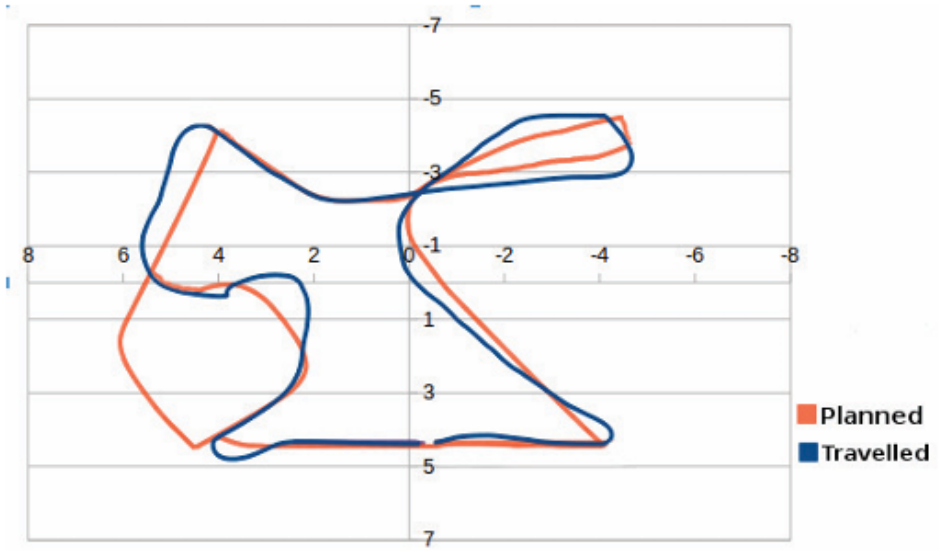


Figure 5.8: Path planning and obstacle avoidance in simulation.

5.4.2 Real-World Performance Evaluation

After successfully testing the system in simulation, the same software stack was deployed on the physical robot for real-world testing. The real-world environment was a modified version of the simulated room, with real obstacles such as furniture, doors, and walls.

5.4.2.1 Localization Performance in the Real World

Localization performance was evaluated by comparing the robot's estimated pose to the ground truth using visual markers and motion capture systems. The results were as follows:

- Positional Accuracy: The robot's pose remained within a 10-15 cm error margin compared to the ground truth. This was slightly higher than in simulation due to the inherent noise in real-world sensors, such as LIDAR and IMU.
- Drift Over Time: While the localization system performed well initially, there was gradual drift over time, especially in areas with poor LIDAR coverage or high reflectivity surfaces. However, the AMCL algorithm was able to correct this drift with the help of real-time sensor feedback.
- Effect of Dynamic Obstacles: The presence of dynamic obstacles, such as moving humans or pets, caused occasional localization degradation. However, the system was able to quickly re-localize after losing track of its position.

5.4.2.2 Performance Metrics

The following performance metrics were used to evaluate the robot's navigation system:

- Success Rate: The robot successfully reached its goals in 92
- Time to Goal: The average time to reach a goal was approximately 3.5 minutes, with variations depending on the complexity of the environment and the path planning challenges.
- Energy Consumption: Power consumption was monitored during the navigation tasks. The robot's energy consumption remained efficient, with minimal battery drain during typical navigation tasks.

5.4.3 Discussion

The results from both simulation and real-world testing demonstrated that the developed autonomous navigation system is capable of performing reliable and efficient navigation tasks in controlled and dynamic environments. The integration of localization, mapping, path planning, and obstacle avoidance provided a seamless navigation experience for the robot. However, several challenges were encountered during real-world testing:

- Sensor Noise: Real-world sensor data is often noisy and affected by various environmental factors, such as lighting conditions and material properties. This caused occasional localization drift and misinterpretation of obstacles.
- Dynamic Obstacles: Handling moving obstacles in real time remains a challenge, as the robot occasionally failed to predict their motion and adjust its trajectory accordingly.
- Robot's Mobility Constraints: The non-holonomic constraints of the robot (i.e., it cannot move sideways) sometimes caused difficulty in navigating tight spaces.

Despite these challenges, the system performed well overall, with minor adjustments needed in specific cases. Future improvements could include the integration of advanced motion prediction algorithms for dynamic obstacles, better sensor fusion for more accurate localization, and enhanced recovery strategies for more complex environments.

Chapter 6

Summary and Conclusion

6.1 Summary

This project represents a significant step toward building a fully autonomous 2D ground robot capable of real-time mapping and navigation in real-world environments. The work began with the successful implementation of a manually controlled stage, where the robot's basic movement and core functionalities were validated in both simulation and physical setups. This initial phase laid the foundation for understanding and integrating the system's hardware and software components.

Key achievements included establishing a ROS2 Foxy-based framework, integrating SLAM algorithms for environment mapping, and validating sensor inputs from LIDAR, IMU, and wheel encoders. Building on this progress, the project has now successfully transitioned into full hardware automation. The robot is now capable of autonomously navigating from a defined start point to a specified goal point in a mapped environment, without human intervention.

This autonomous capability was realized by implementing the ROS2 Navigation Stack, refining the SLAM system for improved localization and map accuracy, and integrating robust motion planning and obstacle avoidance algorithms. Additionally, low-latency and reliable communication between the BOSCH Power Control Board and the Robotics Control Board has been established using XRCE-DDS, enabling seamless interaction between sensors, actuators, and the Jetson Nano processing unit. This integrated system ensures efficient data exchange and precise control, marking a substantial milestone in the development of real-time autonomous ground robotics.

6.2 Conclusion

In conclusion, the project has successfully laid the groundwork for an autonomous robotic platform that leverages cutting-edge technologies for real-time navigation and mapping. The manual stage served as a crucial stepping stone, enabling the validation of hardware and software components. With the transition to the autonomous stage underway, the integration of XRCE-DDS communication between the BOSCH Power Control Board and the Robotics Control Board marks a significant milestone. This integration not only enhances data synchronization but also ensures the robot's ability to operate reliably in resource-constrained environments.

As the project progresses, the focus will be on fine-tuning the autonomous capabilities, including advanced path planning, dynamic obstacle avoidance, and performance optimization. The ultimate goal is to develop a versatile robotic platform that can operate efficiently in real-world applications such as logistics, disaster management, and research environments. This project demonstrates the potential of combining ROS2, XRCE-DDS, and advanced hardware systems to achieve robust and scalable autonomous robotics solutions. hyperref

References

[1] T. Bai, Z. Fan, M. Liu, S. Zhang and R. Zheng, "Multiple Waypoints Path Planning for a Home Mobile Robot," 2018 Ninth International Conference on Intelligent Control and Information Processing (ICICIP), Wanzhou, China, 2018, pp. 53-58,DOI: 10.1109/ICICIP.2018.8606687.

[2] Liu, Zixiang. (2021). Implementation of SLAM and path planning for mobile robots under ROS framework. 1096-1100. 10.1109/ICSP51882.2021.9408882 .

[3] K. M. Hasan, Abdullah-Al-Nahid and K. J. Reza, "Path planning algorithm development for autonomous vacuum cleaner robots," 2014 International Conference on Informatics, Electronics Vision (ICIEV), Dhaka, Bangladesh, 2014, pp. 1-6,doi: 10.1109/ICIEV.2014.6850799.

[4] R. K. Megalingam, A. Rajendraprasad and S. K. Manoharan, "Comparison of Planned Path and Travelled Path Using ROS Navigation Stack," 2020 International Conference for Emerging Technology (INCET), Belgaum, India, 2020, pp. 1-6, doi: 10.1109/INCET49848.2020.9154132.

[5] J. Yao, "Path Planning Algorithm of Indoor Mobile Robot Based on ROS System," 2023 IEEE International Conference on Image Processing and Computer Applications (ICIPCA), Changchun, China, 2023, pp. 523-529, doi: 10.1109/ICIPCA59209.2023.10257966.

[6] J. Yao, "Path Planning Algorithm of Indoor Mobile Robot Based on ROS System," 2023 IEEE International Conference on Image Processing and Computer Applications (ICIPCA), Changchun, China, 2023, pp. 523-529, doi: 10.1109/ICIPCA59209.2023.10257966.

[7] J. Shao, "An improved microcontroller-based sensorless brushless DC (BLDC) motor drive for automotive applications," IEEE Transactions on Industry Applications, vol. 42, no. 5, p. 1216–1222, 2006.

[8] M. A. Al Mamun, M. A. Rahman, M. S. Rahman, M. M. Hassan and A. B. M. S. Ali, " Embedded system for motion control of an omnidirectional mobile robot," IEEE Access, vol. 6, p. 6721–6733, 2018.