

**THE UNIVERSITY OF DANANG
UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ADVANCED SCIENCE AND TECHNOLOGY**



CAPSTONE PROJECT

**AUTONOMOUS ROBOT INTEGRATING
ARUCO MARKER BASED VISION FOR
RELOCALIZATION USING ROS**

NGUYEN DOAN THAO THU

Supervisor: Dr. NGUYEN LE HOA

Co-supervisors: Eng. LE VIET TRIEU TIEN

Submitted to the Faculty of Advanced Science and Technology, Advanced Undergraduate Program in Industrial Informatics in Partial Fulfillment of the Requirements for the Degree of Engineer

Da Nang, June 2025

**THE UNIVERSITY OF DANANG
UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ADVANCED SCIENCE AND TECHNOLOGY**

CAPSTONE PROJECT

**AUTONOMOUS ROBOT INTEGRATING
ARUCO MARKER BASED VISION FOR
RELOCALIZATION USING ROS**

NGUYEN DOAN THAO THU

Supervisor: **Dr. NGUYEN LE HOA**
Co-supervisors: **Eng. LE VIET TRIEU TIEN**

Da Nang, June 2025

[illegible]

[illegible]

ABSTRACT

In recent years, the world continues to develop, technology has also advanced rapidly. In particular, the field of autonomous robots has garnered increasing attention from scientists due to their wide range of applications across various industries. This graduation thesis presents **Autonomous Robot integrating Aruco Marker-Based vision for Re-localization using ROS** at THACO Industries.

The positioning system is designed by combining absolute and relative positioning methods. It is developed on the Robot Operating System (ROS) and utilizes sensors such as cameras, LiDAR (Light Detection and Ranging), and ultrasonic sensors to perceive the environment. In this context, absolute positioning refers to determining the robot's location based on fixed references in the environment (e.g., map frame or known landmarks like Aruco markers), while relative positioning relies on incremental measurements (e.g., wheel encoders or odometry) to estimate movement from a known point.

The project report includes 6 chapters:

Chapter 1: OVERVIEW OF RESEARCH TOPIC

Chapter 2: DESIGN OF AUTONOMOUS ROBOT

Chapter 3: RESEACH OF SOFTWARE

Chapter 4: SIMULATION OF AUTONOUMOUS ROBOT USING ROS

Chapter 5: RESULTS AND ANALYSIS IN THE SIMULATED ENVIRONMENT

Chapter 6: EXPERIMENTAL RESULTS AND EVALUATION

ACKNOWLEDGEMENT

I would like to express my sincere gratitude and appreciation to those who gave us the opportunity to complete this project.

First of all, I would like to sincerely thank my internship supervisors, Mr. Le Viet Trieu Tien and Dr. Nguyen Le Hoa for their invaluable advice and guidance throughout my internship. Their expertise and support have played a crucial role in shaping my professional growth and development throughout this time. I am truly grateful for their constant guidance, encouragement, and willingness to share their knowledge and expertise.

Second, I would also like to thank the entire team at THACO Industries for creating a conducive learning environment. The collaborative and supportive culture within the organization has fostered a sense of belonging and allowed me to grow in my role. The collective knowledge and experience shared by my colleagues have been instrumental in my professional development. I am grateful for their willingness to answer my questions, offer guidance, and provide valuable insights throughout my internship.

Finally, I would like to thank myself for not giving up.

Da Nang, 19th May 2025

Author

(Signature and full name)

Nguyen Doan Thao Thu

DECLARATION

Dear Faculty of Advanced Science and Technology,

The University of Science and Technology - The University of Danang.

I am Nguyen Doan Thao Thu – 20PFIEV2, hereby declare that this thesis complies with the regulations on academic integrity. I affirm that this work and the results presented in this thesis are truthful and do not violate any research ethics. The data and figures in this thesis have been analyzed, interpreted, and evaluated based on various sources, all of which have been properly cited in the references section.

Furthermore, any comments, evaluations, and data borrowed from other authors or organizations have been duly acknowledged and clearly cited.

I take full responsibility for the authenticity and integrity of this thesis. In the event of any detected misconduct, I commit to addressing it transparently and responsibly.

Da Nang, 19th June 2025

Author

(Signature and full name)

Nguyen Doan Thao Thu

TABLE OF CONTENTS

ABSTRACT	i
ACKNOWLEDGEMENT	ii
DECLARATION	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES	vii
LIST OF TABLES	xi
ABBREVIATIONS	xii
Chapter 1: OVERVIEW OF RESEARCH TOPIC	1
1.1 Introduction to the topic	1
1.1.1 Introduction Truong Hai Group	1
1.1.2 Project overview	2
1.2 General investigation	3
1.2.1 The urgency of the topic	3
1.2.2 Overview of existing technical solution	5
1.3 Preliminary solution	7
1.3.1 Object and scope of the research	7
1.3.2 Design process	7
1.4 Expectation	9
1.5 Evaluation method	10
1.6 Assignment of tasks	11
Chapter 2: DESIGN OF AUTONOMOUS ROBOT	12
2.1 Hardware	12
2.1.1 Block diagrams	12
2.1.2 Materials	13

2.1.3 The fabrication process of the experimental model	19
Chapter 3: RESEACH OF SOFTWARE.....	21
3.1 Software	21
3.2 Environment and Programming Software.....	22
3.2.1 Concepts of ROS.....	23
3.2.2 Gazebo	26
3.3 Introduction and selection of SLAM for Mapping.....	29
3.3.1 Selection of SLAM based on the type of sensors used.....	30
3.3.2 Analysis of LiDAR SLAM solutions for Mapping	34
3.3.3 Principle of LIDAR SLAM Gmapping Methods.....	37
3.4 Concept of Aruco Marker	40
Chapter 4: SIMULATION OF AUTONOUOUS ROBOT USING ROS.....	42
4.1 Relocalization Program Design.....	42
4.1.1 Dependent Packages in the Program.....	42
4.1.2 Overall system algorithm flowchart.....	43
4.1.3 Develop the mathematical model	44
4.2 Programming models.....	45
4.2.1 Model programming	45
4.2.2 Environment programming.....	48
4.2.3 Realistic Parameters for Robot.....	51
4.2.4 Developing image processing	52
Chapter 5: RESULTS AND ANALYSIS IN THE SIMULATED ENVIRONMENT	57
5.1 Map scanning and saving in Gazebo simulation.....	57
5.2 Robot localization.....	59
5.3 Obstacle avoidance	61

5.4 Aruco Marker detection and Pose estimation.....	62
5.5 Robot Relocalization Process.....	63
5.6 Experimental measurement and error analysis	65
5.7 Comparison with Expected Performance.....	68
Chapter 6: EXPERIMENTAL RESULTS AND EVALUATION	70
6.1 Mapping and saving in real environment.....	70
6.3 Determining and recording the position of the Aruco marker tag.....	71
6.4 Detect Aruco markers to localize the robot within the map	75
6.5 Robot Relocalization Process.....	75
6.6 Practical Evaluation of Robot Relocalization.....	77
CONCLUSION	81
REFERENCES.....	83
APPENDIX 1: SYSTEM DIAGRAM – SINGLE LINE DIAGRAM.....	85
APPENDIX 2: SYSTEM DIAGRAM – POWER SUPPLY DRAWING	86
APPENDIX 3: SYSTEM DIAGRAM – DETAILED POWER DISTRIBUTION DRAWING.....	87
APPENDIX 4: ROS.....	88
APPENDIX 5: MOMENT OF INERTIA	91
APPENDIX 6: COMPENENTS RVIZ ENVIRONMENT	92
APPENDIX 7: MOVE_BASE PACKAGE	95
APPENDIX 8: NAVIGATION STACK.....	99
APPENDIX 9: DIFFERENTIAL DRIVE	103

LIST OF FIGURES

Figure 1.1: Structure of Truong Hai Group (Source: www.thacogroup.vn).....	1
Figure 1.2: Production activities in Automotive chassis & body parts company (Source: Automotive Chassis & Body Parts Company).....	2
Figure 1.3: R&D Center (Source: R&D Center).....	2
Figure 1.4: Materials Research & Testing Center (Source: R&D Center)	2
Figure 1.5: The “Car Delivery autonomous delivery robot” was showcased at the Viettel Post booth featuring economic and defense products (Source: www.dantri.com.vn)....	5
Figure 1.6: AMR at Phenikaa Electronics Factory (Source: AMR Autonomous Mobile Robot).....	5
Figure 1.7: The Bionic Quadruped Robot was launched by Unitree Robotics in China in 2023. (Source: Robot Dog Go2_Quadruped_Robot Dog Company Robot Unitree) ..	6
Figure 1.8: Service robots by Pudu in Belgium.....	6
Figure 1.9: Engineering design process (Source: TeachEngineering.org)	7
Figure 2.1: Hardware Block Diagram.....	12
Figure 2.2: Intel Real Sense Depth Camera D435i device (Source: Intel® RealSense™ Camera 400 Series Product Family Datasheet).....	13
Figure 2.3: RPLiDAR A1M8 device (Source: LD108_SLAMTEC_rplidar_datasheet_A1M8_v3.0_en.pdf).....	14
Figure 2.4: DC Motor	14
Figure 2.5: NVIDIA Jetson Nano Dev Kit device (Source: NVIDIA Jetson NANO Developer Kit).....	15
Figure 2.6: Arduino Mega 2560 (Source: https://www.alldatasheet.vn/datasheet-pdf/view/1425035/ETC/MEGA-2560.html).....	16
Figure 2.7: DC-DC converter device (Source: https://www.alldatasheet.com/datasheet-pdf/view/1106403/MORNSUN/URB2403YMD-20WR3.html)	17
Figure 2.8: DC Motor Driver device (Source: 7A-160W motor control.pdf).....	18
Figure 2.9: Performed using Autodesk Fusion 360 design the cover	19
Figure 2.10: Prototyping in 3D at the Materials Research & Testing Center	19
Figure 2.11: The robot model after fabrication	20

Figure 3.1: Illustrates the software architecture of the system	21
Figure 3.2: Relational diagram of ROS Filesystem Level [3]	23
Figure 3.3: Packages Structure in ROS [3].....	24
Figure 3.4: Stack Structure in ROS [3]	25
Figure 3.5: Serve-Client model in ROS [3]	26
Figure 3.6: The Gazebo virtual environment for robot testing, this is the available package (the robot is in the red frame).....	27
Figure 3.7: Tags and subtags for building models (Source: Simulate a Mobile Robot in ROS).....	28
Figure 3.8: All objects are assigned positions and can change their locations to generate diverse and dynamic scenarios.....	28
Figure 3.9: SLAM algorithm model	29
Figure 3.10: The result of SLAM (Source: Ego-Motion and Tracking for Continuous Object Learning: A Brief Survey).....	31
Figure 3.11: Robot navigation by Laser scanning from LiDAR in RViz environment (Source: Dynamic objects in Gazebo does not produce laser scan - Gazebo Answers archive).....	31
Figure 3.12: The results of multi-sensor fusion SLAM on the quadraped robot (Source: The results of multi-sensor fusion SLAM on the quadraped robot. Download Scientific Diagram).....	32
Figure 3.13: Types of SLAM Mapping Algorithms [5].....	34
Figure 3.14: Principle of Gmapping Methods [8].....	37
Figure 3.15: Scan matching: problem definition. (a) Scansbeforealignment (b) Scansafteralignment (Source: (PDF) Scan Matching by Cross-Correlation and Differential Evolution).....	38
Figure 3.16: The robot is using a Particle Filter to determine position and direction represented (Source: SLAM using Grid-based FastSLAM - fjp.github.io).....	39
Figure 3.17: a) ARToolKit, b) ARTag, c) AprilTag, d) Aruco Marker [9].....	40
Figure 4.1: Overall system algorithm flowchart.....	43
Figure 4.2: My robot in RViz.....	45

Figure 4.3: The red frame creates wheel's name; The green frame was creates a joint; The blue frame designed LiDAR details.	47
Figure 4.4: Testing environment for this Capstone Project.....	48
Figure 4.5: My configuration SDF file is named 'big_apartment.world'	49
Figure 4.6: a) The <light> tag is used to configure lighting settings; b) The Aruco-2 marker is defined within the <model> tag, with a scale factor of 60% applied along the x, y, and z axes.....	50
Figure 4.7: An example declaration for the robot's body.....	51
Figure 4.8: The yellow, pink, and red frames respectively represent the formulas for calculating the moment of inertia of a cylinder, a box and a sphere	51
Figure 4.9: Aruco Marker image processing flowchart.....	52
Figure 4.10: Detection Aruco Marker tag flowchart.....	53
Figure 4.11: Relocalization flowchart.....	55
Figure 4.12: Automatic relocalization flowchart.....	56
Figure 5.1: The Gazebo environment appears on the screen after executing launch...	57
Figure 5.2: I am commanding scan map on Gazebo	57
Figure 5.3: The robot is currently scanning the surrounding environment with Particle Filter, represented by small yellow arrows	58
Figure 5.4: The robot's movement in Gazebo is immediately reflected in RViz; the robot's position in both environments is relatively consistent.....	58
Figure 5.5: The map consists of two types of data: an image file and a metadata description file	59
Figure 5.6: Configure the RViz environment with GlobalMap and LocalMap.....	59
Figure 5.7: The robot's position relative to the map in Gazebo is (x, y, z) = (5.019100, -3.006682, 0.00000) with a yaw of -3.097307 radians	60
Figure 5.8 The robot's position relative to the map in RViz is (x, y, z) = (5.006, -3.018, 0.000) with a converted yaw angle from radians to degrees as -176.319 degrees	60
Figure 5.9: I am requesting the robot to navigate following the red directional arrow in RViz.....	61
Figure 5.10: I place a black cube along the path of the robot to test obstacle avoidance algorithms	61

Figure 5.11: A new Global Planner has been generated by the robot.....	62
Figure 5.12: A comparison of the two environments is illustrated in this image	62
Figure 5.13: The topic /image has been converted to /aruco_simple/result for observing the IDs	63
Figure 5.14: The robot has captured frame2 and frame0.....	63
Figure 5.15: I used the Pose Estimate tool in RViz to manually displace the robot in Gazebo for testing the localization algorithm	64
Figure 5.16: The node goal_relocalize has outputted the line "Received pose from /pose2" and published it to /initialpose	64
Figure 5.17: Relocalization successful at /pose2.....	65
Figure 6.1: Marker tags are attached in the robot testing area.....	70
Figure 6.2: RViz was launched to visualize and monitor the mapping process	70
Figure 6.3: The robot is being directed to explore specific features within the map ...	70
Figure 6.4: The figure shows matching features between the two maps	71
Figure 6.5: The yaw orientation is measured to be 0.031 radians.....	72
Figure 6.6: The Publish Point tool was used to click precisely at the origin of the frame1 coordinate frame	72
Figure 6.7: The x value is 5.035 meters and the y value is -0.453 meters	73
Figure 6.8: Enlarged image of the terminal	73
Figure 6.9: Command “roslaunch tf_echo map base_link” to get yaw orientation: -1.575	74
Figure 6.10: We have coordinate value with x = -0.556, y = -1.861	74
Figure 6.11: The /pose1 topic was successfully received	75
Figure 6.12: The terminal display was scaled up.....	75
Figure 6.13: The robot started navigating.....	76
Figure 6.14: Relocalization of the robot was completed successfully.....	76

LIST OF TABLES

Table 1.1: Expected result	9
Table 3.1: Comparison of Sensor-based SLAM Methods.....	33
Table 3.2: Comparison of 4 SLAM mapping algorithms.....	35
Table 3.3: Comparison table of fiducial marker systems.....	41
Table 4.1: Details of the tags that build the robot.....	45
Table 4.2: Details of the tags that build the simulation environment	49
Table 5.1: Relocalization Result for Marker ID 0.....	66
Table 5.2: Relocalization Result for Marker ID 1.....	66
Table 5.3: Relocalization Result for Marker ID 2.....	66
Table 5.4: Relocalization Result for Marker ID 3.....	67
Table 5.5: Average Relocalization Error of the 4 Aruco Markers	67
Table 6.1: Relocalization Result for Marker ID 0 at the Testing Center.....	77
Table 6.2: Relocalization Result for Marker ID 0 at the Testing Center.....	77
Table 6.3: Average Relocalization Error at two positions.....	77

ABBREVIATIONS

AMR	Autonomous Mobile Robot
OSHA	Occupational Safety and Health Administration
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
RBPF	Rao-Blackwell zed Particle Filter
2D OGM	2D occupancy grid map
ESS	Effective Sample Size
MCL	Monte Carlo Localization

Chapter 1: OVERVIEW OF RESEARCH TOPIC

1.1 Introduction to the topic

1.1.1 Introduction Truong Hai Group

Truong Hai Group Corporation (TRUONG HAI GROUP), formerly known as Truong Hai Auto Corporation (THACO), was founded on April 29, 1997, in Dong Nai province, southern Vietnam. The founder is Mr. Tran Ba Duong, who currently serves as the THACO Chairman.

After 27 years, from a company which mainly traded and repaired second-hand vehicles, THACO has evolved into a multi-industry holding corporation as shown *Figure 1.1* below which has 6 sub-holdings, i.e. THACO AUTO (Automobiles), THACO INDUSTRIES (Mechanics & Supporting Industries), THACO AGRI (Agriculture), THADICO (Investment – Construction), THISO (Commerce – Service) and THILOGI (Logistics) under a highly integrated and complementary model.



Figure 1.1: Structure of Truong Hai Group
(Source: www.thacogroup.vn)

THACO INDUSTRIES is a subsidiary of THACO, operating in mechanics and supporting industries. With a scale of 320 hectares, a total investment of 1 billion USD, and 8,500 employees, THACO INDUSTRIES owns 19 factories, an R&D center, and a testing center in Chu Lai. Built on a strong mechanical production foundation and in response to the global supply chain integration trend, the company provides products

and services for both domestic and international markets. Below is *Figure 1.2* that of a small part in Automotive Chassis & Body Parts.



Figure 1.2: Production activities in Automotive chassis & body parts company
(Source: Automotive Chassis & Body Parts Company)

R&D Center:

At THACO Industries, R&D focuses on product design, prototyping, and technology development. The 370+ engineers and experts from Korea, Japan, and France work on vehicle design, material selection, and production technology. A key achievement is the Vietnamese brand THACO Bus, with 60% localization and patented design. THACO continues to expand R&D, enhance simulation/testing, and train engineers to develop innovative products. Working space at R&D Center, Materials Research & Testing Center is shown in *Figure 1.3* and *Figure 1.4*.



Figure 1.3: R&D Center
(Source: R&D Center)



Figure 1.4: Materials Research & Testing Center
(Source: R&D Center)

1.1.2 Project overview

This project develops a part of the control system for an autonomous robot prototype, serving as a foundation for future Autonomous Mobile Robots (AMRs) used in

warehouses and factories to perform most warehouse tasks such as transporting, picking, delivering and stacking goods, even in hazardous or toxic environments. Built on ROS (Robot Operating System), the robot integrates camera, LiDAR and ultrasonic sensors for environment perception. One key function is autonomous navigation, utilizing the ROS Navigation Stack, which includes localization, mapping, path planning, and obstacle avoidance to enable safe and efficient movement.

Indoor localization presents challenges such as lighting variations, moving objects, and reflective surfaces, which can affect sensor accuracy. Odometry drift and mapping errors also impact positioning. To address this, THACO Industries proposes an Aruco Marker-based vision positioning system on ROS. The robot detects Aruco Markers, calculates its pose (position and orientation), and integrates this with LiDAR-based localization for improved accuracy.

This approach enhances navigation reliability, reduces dependence on manual intervention, and provides a scalable solution for industrial automation. The system is designed for flexibility and adaptability, supporting future AMR developments at THACO Industries.

1.2 General investigation

1.2.1 The urgency of the topic

According to the Severe Injury Reports (SIRs) from the U.S. Occupational Safety and Health Administration (OSHA), 77 robot-related accidents were identified between 2015 and 2022. Among these, autonomous robots caused 23 accidents, resulting in 27 injuries, primarily leg and foot fractures [1]. Additionally, 54 accidents involved stationary robots, leading to 66 injuries, mainly finger amputations and fractures to the head and torso. Some of these accidents were caused by robot positioning errors, such as odometry inaccuracies, incorrect mapping, cumulative error or losing localization,...

Overall, robots today still face significant challenges in maintaining the accuracy of their localization systems, especially in dynamically changing environments. Factors such as moving objects, people walking around, and environmental changes over time can greatly impact a robot's positioning capability. Over-reliance on LiDAR, cameras, and

GPS can also become an issue, as these technologies are susceptible to interference, occlusion, and limited operating range. In some cases, if an autonomous robot operates in an extensively large space, LiDAR errors are almost unavoidable.

To reduce these positioning errors, the R&D Center of THACO Industries has assigned me the task of researching and developing the "Autonomous Robot Integrated with Aruco Marker-Based Localization Technology on ROS". This project aims to enable the robot to automatically re-localize itself and stay on its designated path inside and outside the factory.

The decision to implement this project is based on several compelling reasons:

- **Ensuring Safety for People and Property:** Preventing unwanted workplace accidents for employees and robots is a top priority for the company. Since a robot's ability to operate efficiently depends on accurate localization, but positioning errors may accumulate over time, re-localization becomes essential.
- **Reducing Labor Costs:** For a multi-industry corporation, optimizing production processes and reducing reliance on manual intervention are crucial factors. This not only streamlines operations but also enables managers to minimize labor costs, thereby increasing profitability. Specifically, it is necessary to transport raw materials, semi-finished products and finished products between intermediate stages or even longer distances between factories.
- **Foundation for AMR Development:** This project serves as a foundation for the development of Autonomous Mobile Robots (AMRs) by the R&D Center. Additionally, it can be integrated with existing robots in THACO Industries' factories, enhancing stability in challenging conditions such as low-light environments and the presence of moving objects. Moreover, the system can be applied in THISO's supermarkets and convention centers to improve the stability of waiter robots currently in use.
- **Commercialization Potential:** THACO Industries aims to commercialize this product for both domestic and international markets.

In summary, the decision to implement the project "Autonomous Robot integrating Aruco Marker-Based vision for Re-localization using ROS" not only enhances THACO

Industries' technological capabilities but also contributes to advancing smart automation solutions for the future. The project focuses on risk prevention rather than damage control, helping to improve the company's safety standards.

1.2.2 Overview of existing technical solution

Other autonomous robot solutions researched and developed in Vietnam include:

- Viettel autonomous delivery robot as shown in *Figure 1.5* is about to be launched in 2025, using LiDAR sensors for localization.



*Figure 1.5: The “Car Delivery autonomous delivery robot” was showcased at the Viettel Post booth featuring economic and defense products
(Source: www.dantri.com.vn)*

- The AMR by Phenikaa-X in *Figure 1.6* is equipped with two 2D LiDAR sensors capable of 360-degree scanning for localization, along with SICK collision sensors.,...



*Figure 1.6: AMR at Phenikaa Electronics Factory
(Source: AMR Autonomous Mobile Robot)*

Around the world, some well-known autonomous robot solutions include:

- Unitree Go2 Robot as *Figure 1.7*, equipped with a 3D LiDAR localization sensor and other advanced technologies developed by Unitree.

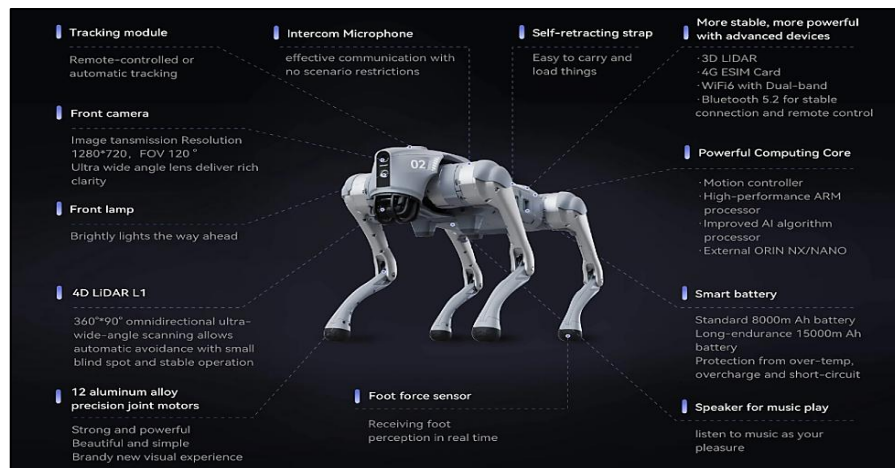


Figure 1.7: The Bionic Quadruped Robot was launched by Unitree Robotics in China in 2023.

(Source: Robot Dog Go2_Quadruped_Robot Dog Company | Robot Unitree)

- Another technology company from China as shown in *Figure 1.8* below has developed multiple intelligent robot models, utilizing SLAM solutions integrated with LiDAR, Camera, IMU, Encoder and various other sensors.



Figure 1.8: Service robots by Pudu in Belgium

(Source: Service robots become "stars" of restaurant in Belgium-Xinhua)

In summary, most of the popular solutions today rely on LiDAR-based localization, integrated with multiple sensors for repositioning. In this project, I propose an alternative localization and repositioning solution by applying Aruco Marker image processing technology. Aruco Marker is a fixed system that can relocalize positions that

filters cannot process, offers high integration capability, and can be applied to various scenarios.

1.3 Preliminary solution

1.3.1 Object and scope of the research

Topic includes the following main subjects:

- Programming simulation Robot
- Detect and re-localization using Aruco Marker
- 2D/3D Visualization for ROS
- Obstacle Avoidance
- Mapping

Research scope:

- Design and build models
- Run experiments and evaluate

1.3.2 Design process

As demonstrated in *Figure 1.9*, the project consists of the following phases:

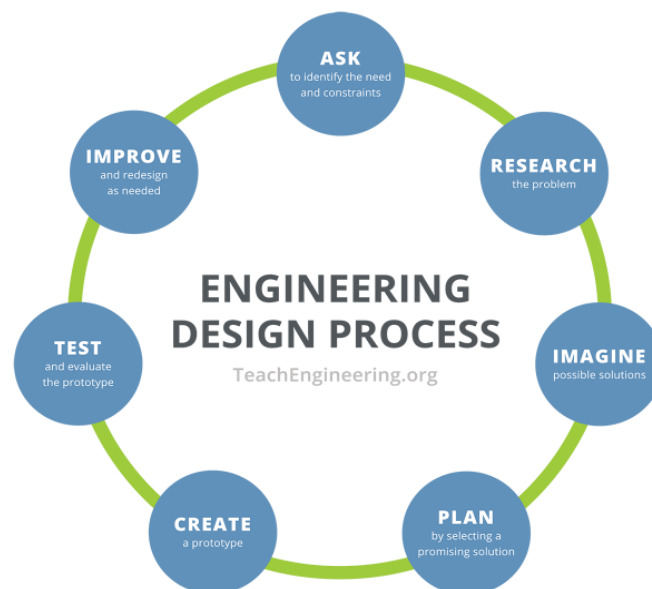


Figure 1.9: Engineering design process
(Source: TeachEngineering.org)

Ask: Identify the Need & Constraints

Goal: Develop an autonomous robot using Aruco Marker-based localization in a ROS environment.

Constraints: Hardware compatibility, real-time processing speed, and marker detection accuracy.

Research the Problem

Analyze existing localization methods: Aruco Markers, SLAM and LiDAR.

Explore industrial solutions and academic research to determine the most effective approach.

Imagine: Develop Possible Solutions

Evaluate options for camera setup (monocular vs. stereo), processing units (Jetson Nano vs. industrial PC), ROS architecture, and robot mobility system (differential vs. omnidirectional wheels).

Plan: Select a Promising Solution

Formulate a detailed development plan including:

- Hardware selection: Cameras, sensors, motors, and computing units.
- Software architecture: Cameras, sensors, motors, and computing units.
- Testing procedures: Criteria to evaluate localization accuracy, efficiency and reliability.

Create: Build a Prototype

Integrate hardware and software components into a working prototype.

Implement Aruco Marker-based localization in ROS and verify key functionalities:

- Marker detection and identification.
- Position estimation using the camera.
- Real-time feedback to the system.

Test and Evaluate Prototype

Conduct comprehensive testing to evaluate:

- Localization accuracy tests: Compare estimated vs. ground-truth positions.

- System responsiveness: Ensure real-time performance.
- Reliability: Identify weaknesses for improvement.

Improve and Redesign as Needed

Refine system performance by:

- Optimizing marker detection under varied lighting.
- Enhancing ROS node efficiency and reducing computation load.

This phase is iterative, meaning multiple refinements may be needed until the system meets all performance requirements.

1.4 Expectation

Following the implementation of the technical solution and design process, the project objectives are categorized into input and output requirements, as shown in *Table 1.1*.

Table 1.1: Expected result

Input <i>Desired Function</i>	Output <i>Expected Result</i>
Flexible ROS Integration	The system is easily extensible, supporting upgrades to localization, mapping and intelligent navigation.
Obstacle Avoidance	Accurate map building to avoid collisions and plan safe navigation paths.
Simulated Robot Testing of Algorithms	Algorithm verification and testing in simulation environments for localization and relocalization.
Aruco Marker Detection	Real-time localization and orientation of the robot.
Relocalization using Aruco Marker	The robot accurately recovers its position after signal loss or localization failure.
Robot Design, Modeling and Fabrication	Build and test a physical robot platform that supports sensor integration and ROS operation, ensuring stable motion during algorithm deployment.
Application in Industrial Environmentsof THACO	Enhanced localization efficiency, reduced quality loss, and ensured worker safety.

Detailed description:

The system is developed on the ROS platform to allow modular and scalable integration. Starting from flexible ROS integration, key components like localization, mapping, and navigation are connected into a unified system.

Using LiDAR and the ROS Navigation Stack, the robot detects obstacles and creates real-time maps. This enables it to avoid collisions and plan safe paths dynamically.

Localization is improved through camera-based Aruco Marker detection, allowing the robot to determine its position and orientation accurately. When localization fails, the robot uses these markers again to relocalize.

Before real-world deployment, all algorithms are verified in simulation to ensure performance and stability. A physical robot is then designed, modeled, and fabricated to support sensors and ROS operation, allowing real testing of the algorithms.

The final system is intended for use in THACO's industrial environment, improving transport accuracy, reducing product loss, and ensuring safety and efficiency on the factory floor.

1.5 Evaluation method

When developing an autonomous robot localization system using Aruco Marker on the ROS platform, evaluation and testing are crucial to ensuring accuracy, stability, and real-world applicability. I rely on the standards set by the company to come up with an evaluation method.

- Good application of existing technologies and equipment, detect Aruco Markers at distances less than 1.5 meters.
- Perform relocalization with a positional error of less than 10 cm and an angular deviation below 5 degrees.
- Evaluate the robot's mapping capability and obstacle avoidance performance.
- Verify the robot's stability and calibrate its localization accuracy.
- Prepare a detailed report based on the results and submit feedback to the company.

1.6 Assignment of tasks

Preparation of Reports and Presentations: Nguyen Doan Thao Thu

Implementation plan:

Phase	Task Description	Start	End
1. Research on Application Solutions	Study ROS and application solutions	05/03/2025	15/03/2025
	Research the positioning model in space	16/03/2025	22/03/2025
	Proposing a practical application solution	23/03/2025	27/03/2025
2. Design of the Experimental Robot Model	Design the conceptual model for testing	28/03/2025	31/03/2025
	Arrange components and wiring for the model	31/03/2025	12/04/2025
	3D print and assemble the experimental model	12/04/2025	14/04/2025
3. Algorithm Programming	Develop the control program on the microcontroller	14/04/2025	19/04/2025
	Implement algorithm for reading Aruco Marker	19/04/2025	23/04/2025
4. Robot and environment simulation	Simulation in a 3D environment on ROS	23/04/2025	01/05/2025
	Conduct experimental testing of the algorithms in a simulation environment.	01/05/2025	09/05/2025
5. Algorithm Testing and Evaluation Report	Conduct experimental testing of the algorithms in a factory environment	09/05/2025	21/05/2025
	Report and evaluate the algorithms and results	21/05/2025	31/05/2025

Chapter 2: DESIGN OF AUTONOMOUS ROBOT

2.1 Hardware

The model features a compact design, weighing less than 2kg, with a protective shell suitable for movement across various environments. Its dimensions are 25x20x19cm (LxWxH).

The materials used for the model ensure durability and reliability in both factory conditions and test environments.

The project requires a two-wheel differential drive system accompanied by two supporting caster wheels.

The block diagram of the system is illustrated as follows *Figure 2.1*. (The system diagrams are also illustrated in Appendices 1, 2 and 3)

2.1.1 Block diagrams

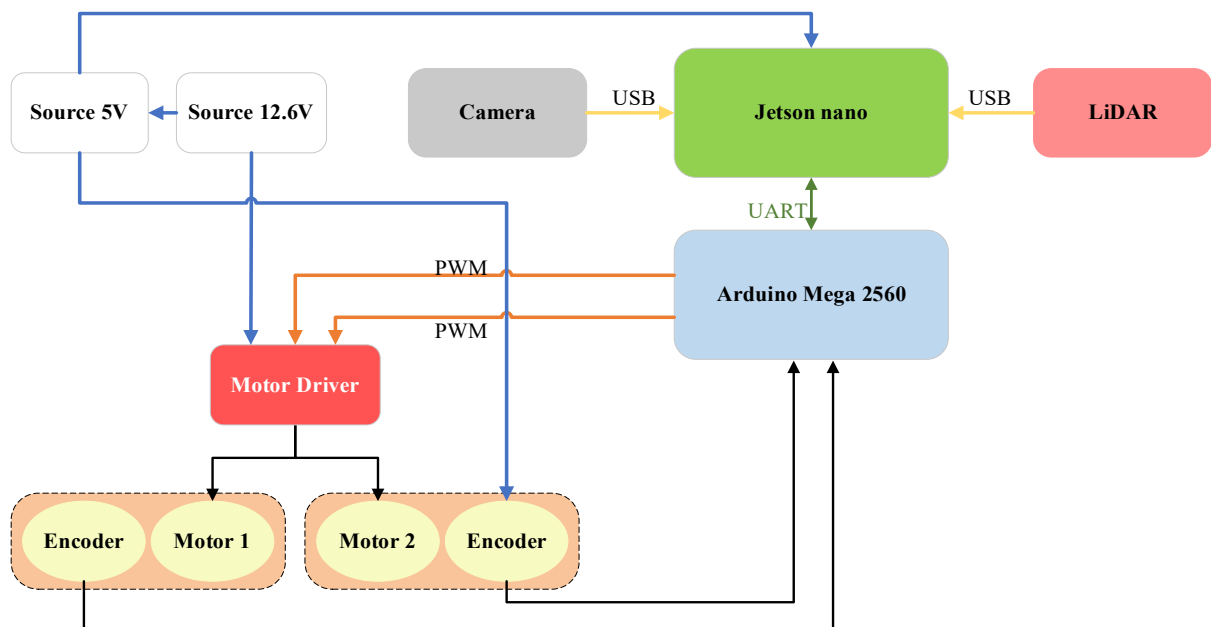


Figure 2.1: Hardware Block Diagram

Working principle of the device:

- The system is powered by two separate sources: a 5V supply for the low-power electronics (e.g., Jetson Nano, Arduino Mega 2560, camera) and a 12V source dedicated to the motor driver and actuators.

- The Jetson Nano serves as the main processing unit, responsible for high-level tasks such as SLAM, image processing, and navigation planning. It receives sensor inputs from a USB camera and a 2D LiDAR, which provide visual and range data, respectively. These inputs are essential for perception, obstacle avoidance, and map generation.
- The Arduino Mega 2560 functions as the low-level motion controller. It communicates with the Jetson Nano via UART and receives motion commands. It generates PWM signals to control a motor driver, which powers two DC motors. The motor feedback is captured by quadrature encoders attached to each motor, and the encoder signals are processed by the Arduino to estimate wheel velocities and enable closed-loop motor control.

2.1.2 Materials

Intel Real Sense Depth Camera D435i: Captures depth and image data for object detection and environment mapping.



*Figure 2.2: Intel Real Sense Depth Camera D435i device
(Source: Intel® RealSense™ Camera 400 Series Product Family Datasheet)*

- Processor: Intel® RealSense™ Vision Processor D4
- RGB Resolution: Up to 1920 x 1080 (Full HD)
- Depth Resolution: Up to 1280 x 720
- Frame Rate: Up to 90 FPS
- Depth Range: Minimum: 0.2 m, Maximum: >10 m (depending on lighting conditions)
- Infrared Projector (IR): VCSEL (narrow-beam laser), Class 1 laser (completely safe for eyes and skin)
- Interface: USB 3.0
- Power Supply: 5V DC, maximum current consumption ~700 mA (via USB)
- Supported Operating Systems: Ubuntu 16.04+
- Operating Temperature: 0°C to 40°C

RPLiDAR A1M8: Measures distances and create a 2D map for navigation and obstacle avoidance.



Figure 2.3: RPLiDAR A1M8 device
(Source: LD108_SLAMTEC_rplidar_datasheet_A1M8_v3.0_en.pdf)

- Measurement range: 0.15–12 m
- Scan angle: 360 degrees
- Scan rate: 1–10 Hz
- Sampling rate: ≥ 8000 points/second
- Distance resolution: < 0.5 mm (within 1.5 m) or $< 1\%$ of the measured distance
- Angular resolution: $\leq 1^\circ$ (at 5.5 Hz)
- Power supply: 5V (4.9–5.5V)
- Laser power: < 5 mW, Class I
- Operating temperature: 0 – 40 °C
- Communication interface: UART TTL 3.3V

Motor integrated Encoder 12V 111RPM: Measures number of rotations of the motors for precise movement control.

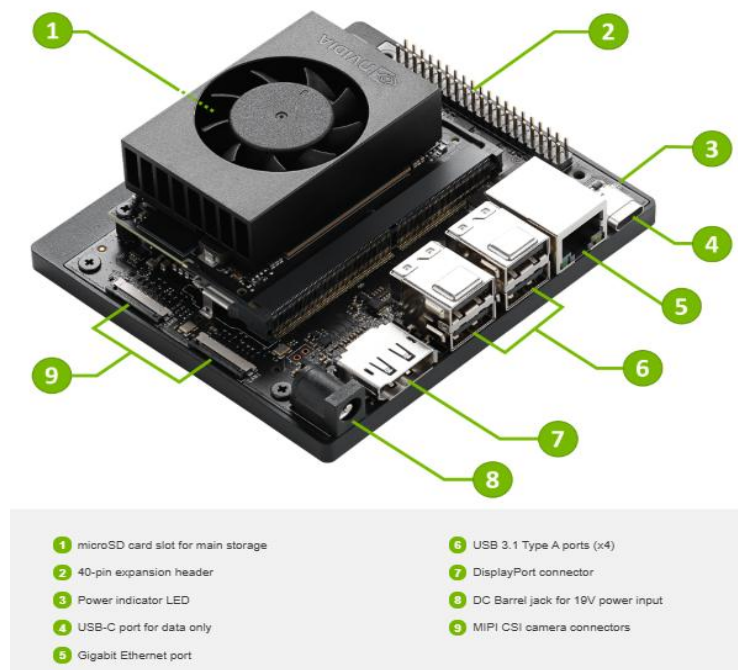


Figure 2.4: DC Motor

- Transmission ratio: 10
- No-load current MA: ≤ 120
- No-load speed: 1000
- Rated torque kg.cm: 1
- Rated speed rpm: 770

- Rated current A: ≤ 1
- Maximum torque kg.cm: 1.6
- Stop current A: 2.3
- Length: 19
- Number of encoder pulses: 110

NVIDIA Jetson Nano Dev Kit: Main processing unit for autonomous navigation and decision-making.



*Figure 2.5: NVIDIA Jetson Nano Dev Kit device
(Source: NVIDIA Jetson NANO Developer Kit)*

Power Supply

- Power Input:
 - Micro-USB (J28): 5V – 2A
 - DC Jack (J25): 5V – 4A (*recommended*)
- Power Output:
 - Header J41 (Pins 2 and 4): 5V – up to 2.5A per pin
 - USB 3.0 Type A ports: 5V – total output ~2A for all 4 ports

Connectivity & Interfaces

- USB: 4 × USB 3.0 Type A ports, 1 × Micro-USB port

- Camera: MIPI CSI-2 interface (2 lanes)
- Display: HDMI 2.0
- Networking: Gigabit Ethernet (RJ45), PoE support
- Expansion:
 - GPIO, UART, I2C, SPI, I2S (via Header J41)
 - M.2 Key E slot (supports WiFi/Bluetooth modules)

Software

- JetPack SDK:
 - Operating System: Ubuntu Linux
 - Includes: CUDA, cuDNN, TensorRT, OpenCV, VisionWorks, VPI
 - Tools and Libraries: AI development tools and libraries
- Installation Options: via microSD card or NVIDIA SDK Manager

Arduino Mega: Acts as a microcontroller to interface with sensors and control motors acts as a bridge between the Jetson Nano and the motors by receiving control signals, converting them into PWM signals and processing encoders data.



Figure 2.6: Arduino Mega 2560

(Source: <https://www.alldatasheet.vn/datasheet-pdf/view/1425035/ETC/MEGA-2560.html>)

- Power Supply:
 - Via USB or external DC jack
 - Includes pins: +5V, +3.3V, GND, VIN
- Operating Temperature: -40 °C to +85 °C
- Operating Frequency: 16 MHz
- Working Registers: 32 x 8-bit registers
- PWM: 15 PWM output pins
- Analog Input: 16 pins
- Digital I/O: 54 pins (including UART, I2C, SPI)

12-24V Lithium Ion Battery:

The 3S battery pack I used consists of three Li-ion cells connected in series, each with a nominal voltage of 3.7V, resulting in a total voltage of 11.1V and reaching 12.6V when fully charged. Since the cells are connected in series, the overall capacity is equal to the capacity of a single cell, which is 7.5Ah, and each cell must support a maximum discharge current of 35A. To ensure safe operation during both charging and discharging, the pack is equipped with a 3S Battery Management System (BMS), which provides over-voltage and under-voltage protection, current limiting, and cell balancing, thereby enhancing the durability and safety of the battery pack.

5-12V Power: Converts and regulates voltage levels to power different components.



Figure 2.7: DC-DC converter device

(Source: <https://www.alldatasheet.com/datasheet-pdf/view/1106403/MORNSUN/URB2403YMD-20WR3.html>)

Key Features

- Power: 20W
- Input-to-output isolation voltage: 1500VDC
- High efficiency: up to 91%
- Ultra-wide input voltage range:
 - 24V models: 9–36VDC
 - 48V models: 18–75VDC
- Operating temperature range: -40°C to +105°C
- Built-in protections:
 - Reverse polarity protection (available in chassis or DIN-rail mounting types)
 - Input under-voltage, output over-voltage, over-current, and short circuit protection
- Certifications: EN62368, UL62368, IEC62368 (UL/CE/CB)

DC Motor Driver XY-160D: Controls the direction and speed of DC motors based on signals from the microcontroller.

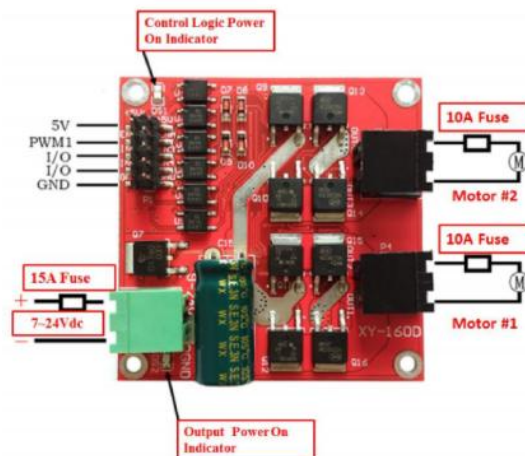


Figure 2.8: DC Motor Driver device
(Source: 7A-160W motor control.pdf)

- Supply Voltage: 7 ~ 24 VDC. (Limit: 6.5 ~ 27VDC).
- Control signal Level (Compatible 3.3V/5V)
 - Logic High (H): DC 3.0 ~ 6.5V
 - Logic Low (L): DC 0 ~ 0.8V
- Output Channel: 2.
- Control signal current: 3 ~ 11 mA (Each route).
- Maximum continuous operating current: 7A.
- Peak current: 50A.
- PWM Speed Control: 0~10KHz.
- Minimum valid Pulse Width: 5us.
- Working Temperature: -25 ~ 85 °C
- Mounting Hole: M3.
- Dimension (LxWxH): 55 x 55 x 13(mm).
- Weight: 32g.

Robot's cover:

- Fusion 360: A cloud-based 3D CAD, CAM and CAE tool used for designing, engineering and manufacturing.



Right View



Front View



Top View

Figure 2.9: Performed using Autodesk Fusion 360 design the cover

- FlashPrint: A slicing software that converts 3D models into printable layers for FlashForge 3D printers.

2.1.3 The fabrication process of the experimental model

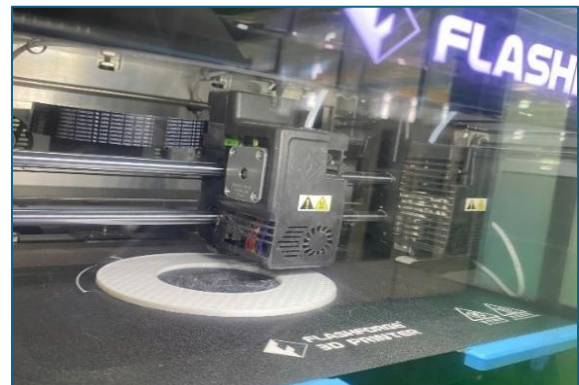
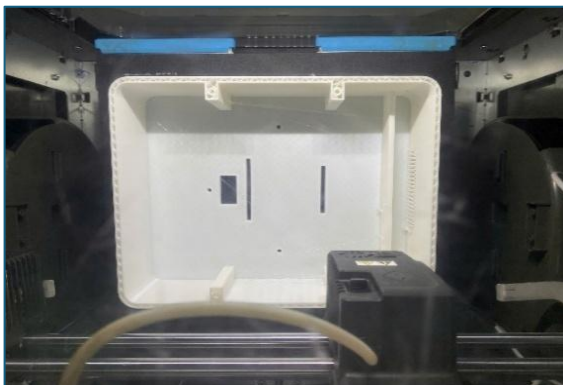
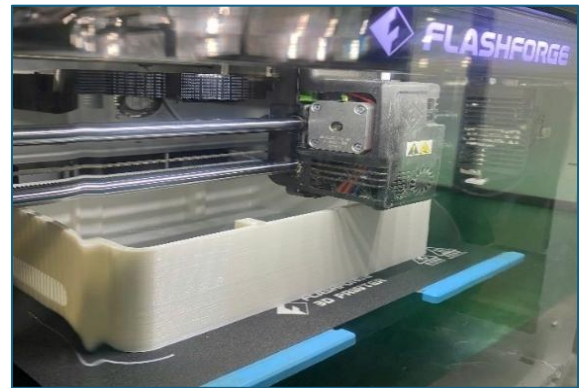
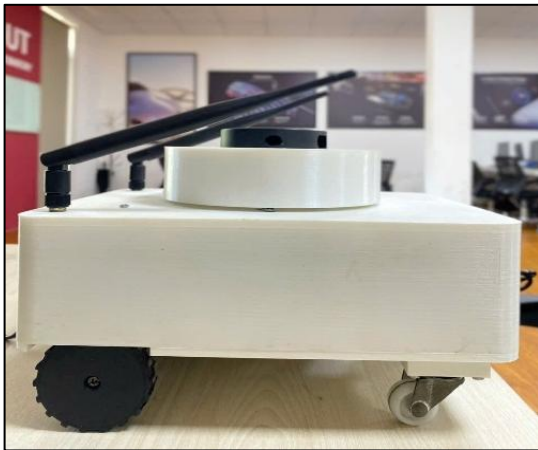


Figure 2.10: Prototyping in 3D at the Materials Research & Testing Center

Image of completed test model



Front View



Left View



Back View

Figure 2.11: The robot model after fabrication

Chapter 3: RESEACH OF SOFTWARE

3.1 Software

To better visualize my project, I have presented it in *Figure 3.1* as an illustration of the software architecture of the system. Each block of the diagram will be explained in detail in the following sections.

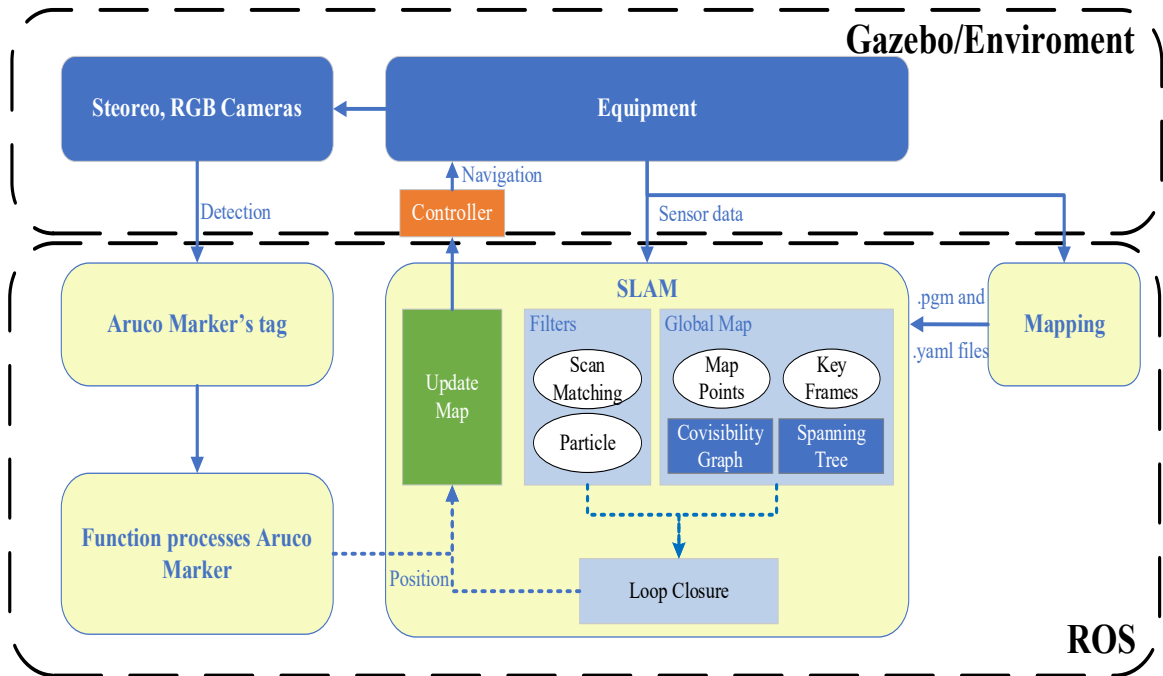


Figure 3.1: Illustrates the software architecture of the system

- The simulation environment consists of two parts: Stereo-RGB cameras and Component-Simulations. The sensor data gathered from the environment serves as the input for map building and SLAM.
- After processing the LiDAR data, the Map Building module uses the GMapping algorithm to generate two map files a *.pgm* occupancy grid and a corresponding *.yaml* file. These files also serve as inputs to the SLAM pipeline for scan alignment and relative pose estimation of the robot. Within SLAM, Filters apply Scan Matching and a Particle Filter technique to align the new data with the existing map and estimate the robot's pose in the environment. SLAM also maintains a Global Map composed of KeyFrames and MapPoints. A KeyFrame is an image frame capturing a unique view of the environment, while a MapPoint

stores the 3D position, feature descriptors, and references to every KeyFrame that observes it. Each KeyFrame links to multiple MapPoints, and each MapPoint may be observed by several KeyFrames. Together, all active KeyFrames and their MapPoints form the current Global Map. The outputs of the Filters and the Global Map feed into the Loop Closure module, which compares the robot's current pose against the Global Map to correct accumulated drift. After loop closure, SLAM updates the map using the newly refined pose. Once SLAM has produced an accurate pose estimate, the Controller takes both the robot's position and the static map to perform path planning, collision checking, and motion-decision logic. It issues velocity and steering commands to the robot's actuators and, by integrating with the ROS Navigation Stack (using *move_base*, a *global_planner* and a *local_planner*; as detailed in *Appendix 7* and *Appendix 8*), enables the robot to autonomously navigate to its goal within the SLAM-constructed environment.

- During the map-building process or while operating according to the designed plan, if the system loses its positioning, it will activate Stereo and RGB Cameras. The cameras will then detect the Aruco Markers placed in the environment. These markers are processed by the Function that handles Aruco Markers, providing the ground truth marker pose to help the system re-localize the robot. The result obtained is also the Position and the Update Map loop continues to provide an accurate position.

In the course of conducting this research and completing the project, a wide range of software tools were employed to support tasks such as computation, simulation, and system design. The details of these tools are presented in the following section.

3.2 Environment and Programming Software

Linux is an open source operating system. First developed in 1991, it has since evolved into various distributions such as Ubuntu, Mint, and Debian. Linux is widely adopted due to its free availability and ability to run efficiently on low-specification hardware. In this project, Linux is used with the Ubuntu 18.04 distribution.

3.2.1 Concepts of ROS

“Robot Operating System (ROS) is a flexible framework that provides various tools and libraries for writing robotic software. It offers several powerful features to help developers in tasks such as message passing, distributed computing, code reusing, and implementing state-of-the-art algorithms for robotic applications” [2]. ROS provides a flexible framework for developing, simulating and deploying robotic applications.

- ROS is not a programming language. In fact, most ROS programs are written in C++. It also supports other languages like Python, Java, Lisp, and more through client libraries. [2]
- ROS is not just a library. Besides providing libraries, ROS includes a central server, command-line tools, graphical tools, and a build system. It offers many libraries for various tasks such as AI, computer vision, natural language processing, and navigation. [2]
- ROS is not an integrated development environment (IDE). ROS doesn't require any specific IDE and works well with most common ones. Many developers prefer using ROS with just a text editor and terminal instead of a full IDE. [2]

ROS is completely free and can be used for commercial purposes, making it a popular choice for testing, development, and cost-effective product deployment. Users simply need to follow the official ROS installation guide to get started. *(A clearer illustration of ROS is provided in Appendix 4)*

Filesystem Level

Figure 3.2 is a relational diagram of the concepts grouped in the Filesystem Level. Let's explain them a bit: [3]

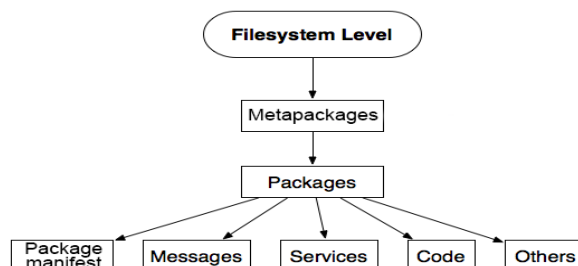


Figure 3.2: Relational diagram of ROS Filesystem Level [3]

- **Packages:** Main software units in ROS. Each handles one function (e.g., camera, localization). Contains code, nodes, config, and a package.xml file.
- **Metapackages:** Groups of related packages. Contains only a package.xml.
- **Package Manifest:** The package.xml file. Stores package info (name, version, dependencies).
- **Message Types (.msg):** Define data formats for node communication (e.g., int, float, string).
- **Service Types (.srv):** Define request/response interactions between nodes.

Packages

In the ROS framework, packages serve as the core building blocks for organizing software. Each package may include nodes, ROS-specific libraries, configuration files, and other resources that enable its functionality. The main goal is to encapsulate minimal and reusable components. This modular structure is illustrated in *Figure 3.3*, which shows how a typical ROS package may consist of nodes, message definitions and service interfaces highlighting the internal organization of a package used in ROS-based development environments [3].

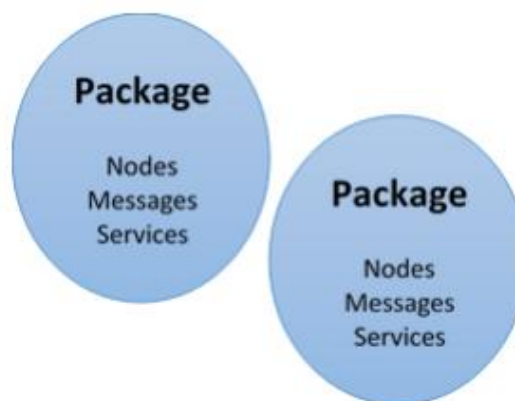


Figure 3.3: Packages Structure in ROS [3]

ROS enforces a naming rule for packages that permits only lowercase letters, digits, and underscores to ensure consistency across systems.

Metapackages

Metapackages are a special type of ROS package used to represent collections of related packages. Rather than containing code or data directly, they provide a logical grouping

and are particularly helpful when transitioning between systems or maintaining backward compatibility, such as with older rosbuilt stacks.

Package Manifest

A package manifest is an XML file named `package.xml` located at the root of each package or metapackage. This file holds essential metadata such as the package name, version number, maintainer details, description, and dependencies. It acts as a declaration of the package's structure and requirements.

Stacks

When we combine packages together with a few functions, it's called a stack. In ROS, there are many stacks with different uses. Similar to Packages as shown in *Figure 3.4*, where information about the stack is called the stack manifest.

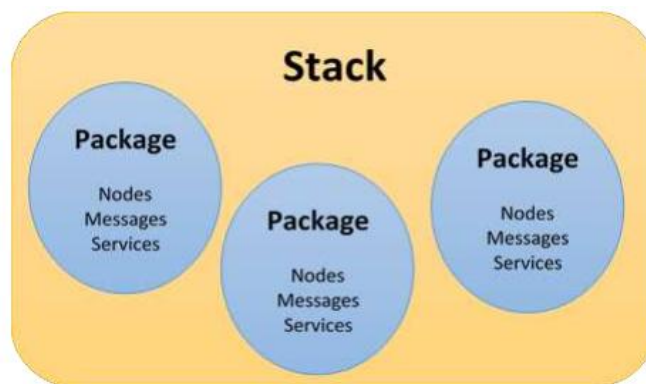


Figure 3.4: Stack Structure in ROS [3]

The goal of using a stack is to simplify the process of sharing application programs.

Service types

Describes a service, define the structure of request and response data exchanged between nodes in ROS. These are stored in `.srv` files, typically located at `my_package/srv/MyServiceType.srv`. As shown in *Figure 3.5*, ROS implements a client-server model, where one node acts as the server providing the service, and another node as the client invoking it using the package and service name [3].

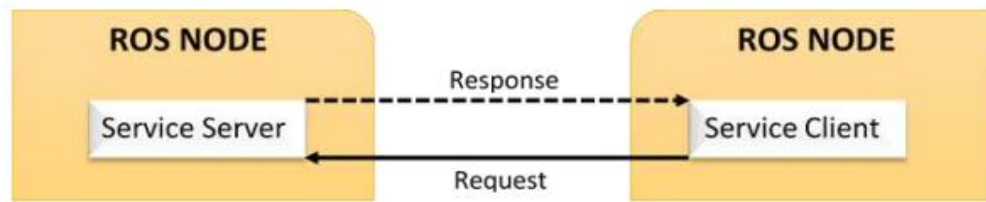


Figure 3.5: Serve-Client model in ROS [3]

In my Capstone Project "Autonomous Robot integrating Aruco Marker-Based vision for Re-localization using ROS", ROS plays a central role in the development and organization of the entire autonomous robot system. Specifically, ROS provides a flexible platform that enables the connection and coordination of various components such as sensors (camera), image processing algorithms, localization systems, and robot control modules. Through its communication mechanisms topics, services, and actions ROS ensures that data from the camera is transmitted to the Aruco Marker recognition algorithm in a continuous and stable manner. Another key feature of ROS is the *tf* tool, which manages coordinate frames (e.g., from marker to robot), allowing for accurate localization of the robot within its environment. Furthermore, ROS provides motion control packages that enable the robot to autonomously navigate to predefined locations based on the localization data. Finally, with the support of simulation environments such as Gazebo, I am able to test the entire system virtually before real-world deployment, thereby improving reliability and reducing risks.

In summary, ROS is not only a development platform but also a crucial framework that enables the integration, operation, and optimization of the entire Aruco Marker based robot localization system.

Now that we've gained a foundational understanding of ROS and how it enables communication between various components in a robotic system, we can shift our focus to one of its most essential problem to test or evaluate navigation and localization algorithms before real world deployment: Gazebo environment.

3.2.2 Gazebo

Gazebo is a 3D simulation program that lets you emulate robots and their surrounding environments with real-world physical properties such as gravity, velocity, acceleration, etc. When you don't have a real robot on hand or you want to test your robot in a virtual

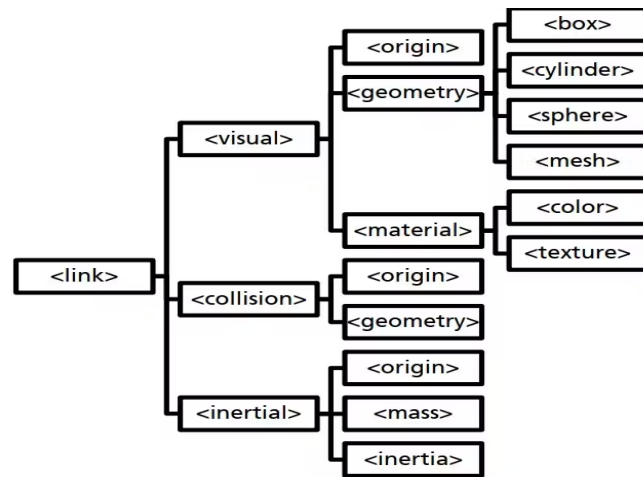
environment before running it in the real world you should use Gazebo. An example of Gazebo virtual environment is depicted in *Figure 3.6*.



Figure 3.6: The Gazebo virtual environment for robot testing, this is the available package (the robot is in the red frame)

During the development and testing of autonomous robotic systems, building a virtual simulation environment is an essential condition that allows for algorithm verification without the need to deploy on actual hardware. The world file used in this project serves as a 3D environment simulation model in the Gazebo software, acting as a virtual world where the robot can interact, move and perform tasks such as localization or navigation. The file is written in a standard XML format, enabling detailed definitions of physical parameters, lighting, ground surfaces, simulated objects and the robot used in the testing space.

The simulation environment is carefully designed to mimic real-world conditions. For instance, sunlight is configured as a directional light source to create realistic shadows and reflections, while the ground is modeled as a large neutral-colored plane where all activities take place. The setup of this environment relies on a structured XML format using specific tags and subtags. These elements, used to define lights, ground planes, and object properties, are illustrated in *Figure 3.7*, which shows the key components for building models in the simulation.



*Figure 3.7: Tags and subtags for building models
(Source: Simulate a Mobile Robot in ROS)*

During simulation, the robot moves within the environment as demonstrated in *Figure-3.8*, while objects can also change positions, creating unexpected and complex scenarios similar to those in real-world situations. This is particularly useful for evaluating the robot's localization capability-that is, its ability to accurately determine its position and orientation in space.

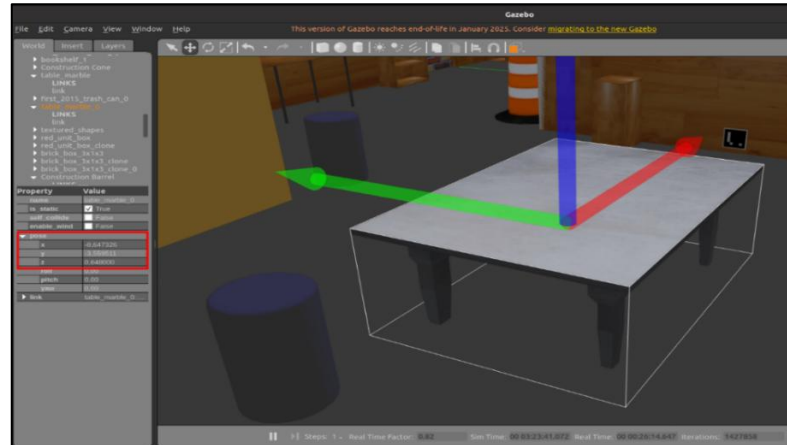


Figure 3.8: All objects are assigned positions and can change their locations to generate diverse and dynamic scenarios

From a physics perspective, the simulation environment uses the ODE (Open Dynamics Engine) a powerful engine for handling object dynamics. Parameters such as gravity, friction coefficients, and simulation update rates are set to relatively high values to ensure smoothness and accuracy of the results. Notably, this virtual world also employs the WGS84 geodetic coordinate system, allowing simulations to be based on global parameters such as latitude and longitude.

Simulate in Gazebo is a way that helps me develop, test and adjust navigation solutions while minimizing risks and costs when transitioning to real-world deployment. Gazebo has created a comprehensive testing environment that effectively supports the development of autonomous robots in both research projects and practical applications. Gazebo can be said to be similar to the real world.

Section 3.2.2 of Chapter 3 will specify how to create a virtual simulation environment.

3.3 Introduction and selection of SLAM for Mapping

SLAM stands for Simultaneous Localization and Mapping, a technology that enables robots to autonomously build a map of their surroundings (mapping) while simultaneously determining their position (localization) within that map. It involves two key components: localization and mapping. However, the definitions and specific implementations of these components may vary depending on the particular application in which SLAM is used.

To better visualize about the SLAM, I have presented it in *Figure 3.9* as an illustration of the SLAM algorithm model:

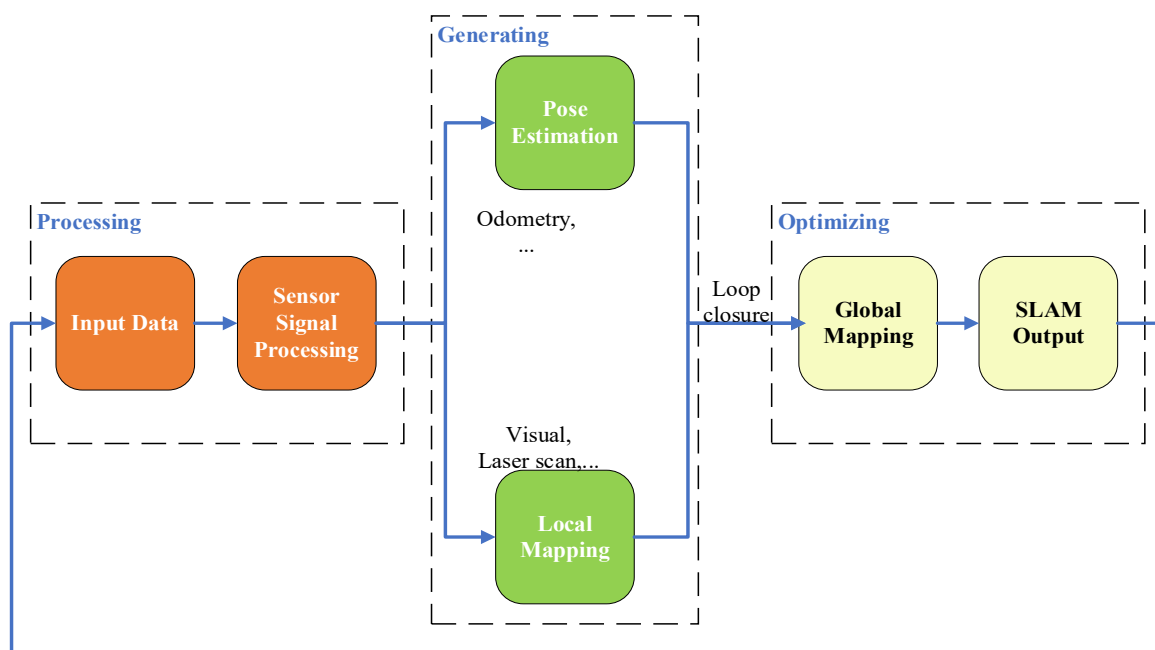


Figure 3.9: SLAM algorithm model

Processing block: This block receives and processes signals from sensors to prepare input data for the subsequent steps.

- **Sensor Input:** Collects raw data directly from sensors such as images from cameras, scan points from LiDAR, encoder ticks, GPS, etc.
- **Data Analyzing:** Analyzes and processes raw data, such as transforming data from the sensor's coordinate system to the robot's reference frame, filtering out noise and normalizing sensor data.

Generating block: This is the stage where a local map is estimated.

- **Pose Estimation:** Uses data such as odometry, GPS or image processing methods to estimate the robot's position in space and calculates its relative position from the starting point or a known map.
- **Local Mapping:** Builds a local map based on data from laser scans, visual images, or detected landmarks.

Optimization Block: This stage focuses on fusing data from multiple sources to create an accurate global map and optimize the robot's position.

- **Global Mapping:** Uses loop closure algorithms to merge local maps and build a complete global map of the environment, eliminating accumulated errors from raw data to enhance mapping accuracy.
- **SLAM Output:** Applies graph optimization algorithms such as Bundle Adjustment or Graph Optimization to update the robot's position and its surrounding environment.

3.3.1 Selection of SLAM based on the type of sensors used

Visual SLAM (V-SLAM)

Visual SLAM uses cameras (monocular, stereo, or RGB-D cameras) to capture images from the surrounding environment. Through image processing and computer vision algorithms, the system can detect features, track motion, and build a map. An example of such a mapping result is shown in *Figure 3.10*, which visualizes the output of a SLAM process in a real environment. [4]



Figure 3.10: The result of SLAM

(Source: Ego-Motion and Tracking for Continuous Object Learning: A Brief Survey)

- Advantages: Low cost and V-SLAM can operate in both indoor and outdoor environments.
- Limitations: Accuracy heavily depends on lighting conditions and the ability to detect image features.

LiDAR SLAM

LiDAR SLAM uses LiDAR sensors (Light Detection and Ranging) to measure the distance to surrounding objects. From this data, the system generates a high-precision point cloud map and uses SLAM algorithms for simultaneous localization and mapping. *Figure 3.11* illustrates this process, showing robot navigation using LiDAR-based laser scanning in the RViz environment. [4]

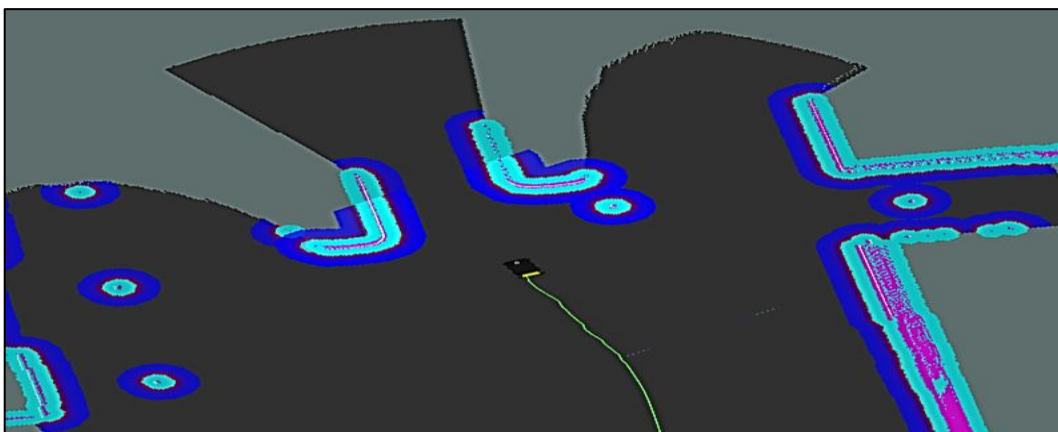


Figure 3.11: Robot navigation by Laser scanning from LiDAR in RViz environment

(Source: Dynamic objects in Gazebo does not produce laser scan - Gazebo Answers archive)

- Advantages: High accuracy performs well in low-light conditions or at night.

- Limitations: Higher cost compared to cameras, heavy data load and require powerful processing.

Fusion SLAM

Fusion SLAM integrates multiple types of sensors such as cameras, LiDAR, IMU (Inertial Measurement Unit), GPS, etc..., to leverage the strengths of each sensor type, enhancing the reliability and accuracy of localization and mapping. *Figure 3.12* demonstrates the results of this approach, showing how multi-sensor fusion SLAM is applied on a quadruped robot to achieve robust localization and mapping performance. [4]

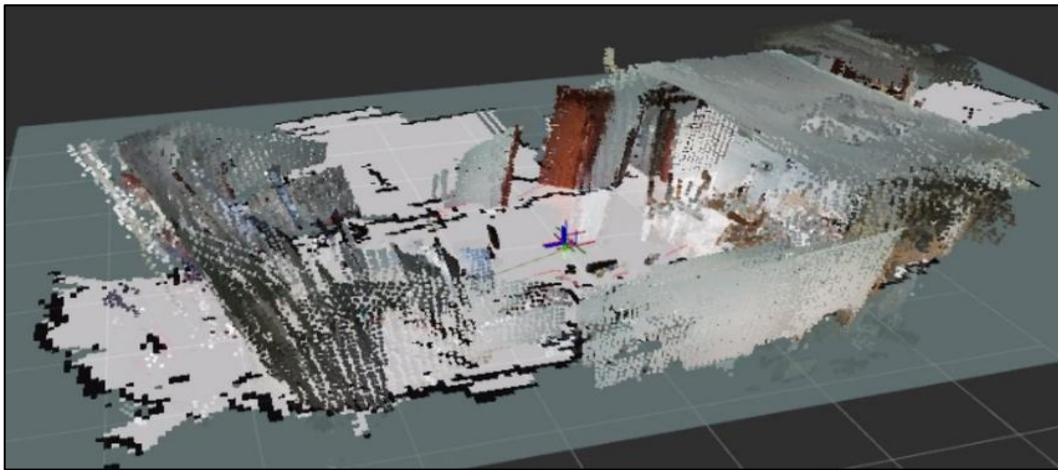


Figure 3.12: The results of multi-sensor fusion SLAM on the quadruped robot
(Source: The results of multi-sensor fusion SLAM on the quadruped robot. | Download Scientific Diagram)

- Advantages: High stability and capability of operating in various environmental conditions.
- Disadvantages: Complex system architecture requires data synchronization and more sophisticated processing.

Based on the content presented above, I have created a comparison *Table 3.1* to select the most suitable implementation method.

Table 3.1: Comparison of Sensor-based SLAM Methods

	Visual SLAM	LiDAR SLAM	Fusion SLAM
Definitiaon	SLAM algorithms use cameras, image sensors and IMU sensors.	SLAM algorithms mainly use LiDAR sensors and Camera.	SLAM algorithms combine data from multiple sensors such as cameras, LiDAR, IMU... to improve accuracy and reliability.
Classification	• Monocular vSLAM • Stereo vSLAM • RGB-vSLAM	• SLAM with 2D LiDAR • SLAM with 3D LiDAR	• Visual + LiDAR SLAM • LiDAR + IMU SLAM • Visual + IMU SLAM • Visual + LiDAR + IMU SLAM
Advantages	- Cost-effective: Cameras are cheaper than LiDAR. - Wide field of view: Cameras offer a wide view, suitable for object detection in crowded environments.	- High accuracy: excellent in building precise maps in large environments. - High reliability and adaptability: unaffected by low lighting or dust. - 3D mapping capability: high-resolution 3D mapping.	- Optimizes strengths of each sensor: combines LiDAR's accuracy with camera's image details. - Performs well in complex environments: low light, many obstacles. - Minimizes individual sensor weaknesses.
Disadvantages	- Relatively low accuracy: not ideal for applications requiring high precision over distance. - Sensitive to lighting: accuracy is affected in low or overly bright environments.	- High cost: LiDAR sensors are significantly more expensive than cameras. - Narrow field of view: limited range, difficult to detect objects in blind spots or above the robot; can be affected by strong light or mirrors, smokes. - High processing demand: large data from sensors requires heavy computation.	- High deployment cost: requires multiple sensors and complex data synchronization. - Demands advanced processing software and algorithms to integrate multi-sensor data. - High computational cost.

Conclusion:

- The 2D LiDAR-based SLAM solution proves to be an effective and cost-efficient choice for this project. Its real-time processing capability and robustness under various conditions make it suitable for autonomous navigation. By combining LiDAR and camera sensors, the system leverages the strengths of both precise

distance measurement and visual recognition to enhance environmental perception. This fusion improves localization accuracy, reduces drift and strengthens loop closure, contributing to a more reliable SLAM framework.

3.3.2 Analysis of LiDAR SLAM solutions for Mapping

In the field of SLAM, several algorithms are widely used for constructing maps of unknown environments. Each has its own strengths and is suitable for different types of robot platforms and sensor setups [5]. These commonly used SLAM mapping approaches are categorized and illustrated in *Figure 2.22*.

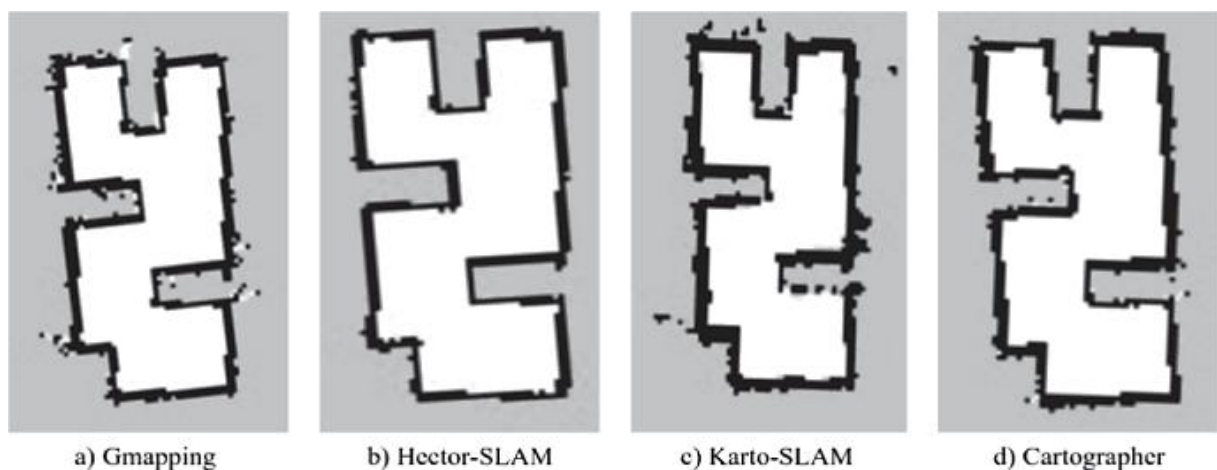


Figure 3.13: Types of SLAM Mapping Algorithms [5]

Cartographer: Developed by Google, it supports both 2D and 3D SLAM. Cartographers employ methods such as pose-graph optimization and loop-closure detection to enhance map accuracy. It uses sensors to collect data, then optimizes the robot's pose and the map by constructing a *pose graph*, where each node represents a potential robot pose and the edges represent the relationships between them.

KARTO SLAM: The algorithm operates based on a graph. The graph nodes represent either robot poses or sensor measurements. The graph edges encode spatial relationships or constraints between poses, derived from motion data or sensor observation. [6]

Hector SLAM: Primarily uses 2D LiDAR sensors to build maps and localize within the environment. Hector SLAM is specifically designed for fast-moving robots and does not require kinematic measurements (such as odometry). It employs a *scan matching*

approach to compare current LiDAR scans with previous ones and update the map accordingly. [6]

GMapping: It uses odometry data, LiDAR and probabilistic filtering to build maps and estimate the robot's position in real time. It is based on the Rao Blackwell zed Particle Filter (RBPF) to create a 2D occupancy grid map (2D-OGM). Gmapping uses a *Particle Filter* to maintain a set off “particles”, each representing a potential pose and map off the robot. [6]

Based on the content presented above, I have created Table 3.2 to compare and select one of the four SLAM mapping algorithms for implementation.

Table 3.2: Comparison of 4 SLAM mapping algorithms

	GMapping	Hector-SLAM	Karto-SLAM	Cartographer
How does it work?	GMapping uses the Particle Filter to maintain a group of "particles", each representing a potential position and map of the robot. Every time the robot moves, GMapping updates the robot's position based on sensor data (encoder, camera), thereby updating the surrounding environment map. These particles resample focus on areas with higher probability of accuracy.	Hector SLAM uses a scan matching technique to compare current LiDAR scans with previous ones and update the map. A key feature of Hector SLAM is that it does not require information about the robot's velocity or kinematics to operate only LiDAR measurements are needed. Hector SLAM employs a special type of filter to improve accuracy during scan matching operations.	1. Collect LiDAR, pose, and odometry data to calculate the initial pose. 2. Compare pose changes to determine if a map update is needed 3. Scan new data to update the current map (Coarse + FineSearch) 4. Add vertex and edge information to the graph. 5. Loop closure detection and global optimization using Sparse Pose Adjustment to adjust the graph and reduce constraint errors. 6. Update the map and repeat.	It uses sensors to collect data. Then, it optimizes the robot's position and the map by constructing a pose graph, where each node represents a potential robot pose and the relationships between them. The ability to detect loop closures and correct accumulated errors during movement helps improve the accuracy of the map.

Advantages	Real-time mapping. High accuracy: Accurate when building small-scale maps Low computation: Requires less computation to build small-scale maps.	No need for odometry such as encoders or IMUs. Suitable for fast-moving robots: due to minimal processing of kinematic data. High accuracy: Due to the use of <i>scan matching</i> techniques.	Accurate mapping capability in large-scale environments. High convergence speed and accuracy in solving nonlinear problems. Fast and resource-efficient 2D mapping capability.	3D SLAM: constructs complex 3D maps High accuracy: <i>pose graph</i> optimization algorithms enhance precision. Loop closure detection identifies and corrects accumulated errors.
Disadvantages	Not suitable for large-scale maps: the required number of particles increases, making loop closure infeasible. No loop closure detection: as a result, the map may be misaligned when loops are closed. Increased computation and memory usage: increasing the number of particles may help complete the map but leads to higher computational and memory demands.	Inaccurate with large-scale maps due to lack of loop closure capability and poor LiDAR quality. Dependent on LiDAR scan quality: if the sensor is poor, map quality will also be affected. Limitations in complex environments: struggles when dealing with complex surroundings.	The algorithm is not suitable for 3D mapping. Large maps require more processing time when high accuracy is needed. Limited development in open-source support.	No longer maintained or supported by Google. Requires powerful computational resources: especially strong hardware for processing 3D data. More complex to implement: harder to deploy, requiring more detailed configuration.

Conclusion:

- Cartographers no longer receive updates or new development from Google, only minimal maintenance [7] and it requires high-performance hardware, making it difficult to deploy.
- GMapping and Karto-SLAM provide high accuracy, while Hector SLAM offers lower accuracy and relatively poor mapping performance due to insufficient update frequency and precision.

- In large-scale factory environments using 2D LiDAR, Karto-SLAM is more effective than GMapping. However, for my Capstone Project, Gmapping using a Particle Filter and not requiring complex data structures like Graph Optimization in Karto-SLAM reduces hardware load and is more resource-efficient.

3.3.3 Principle of LIDAR SLAM Gmapping Methods

The algorithm receives two types of input data as *Figure 3.14*: 2D laser scan measurements δ and odometry data u . The processing pipeline consists of five main stages: [8]

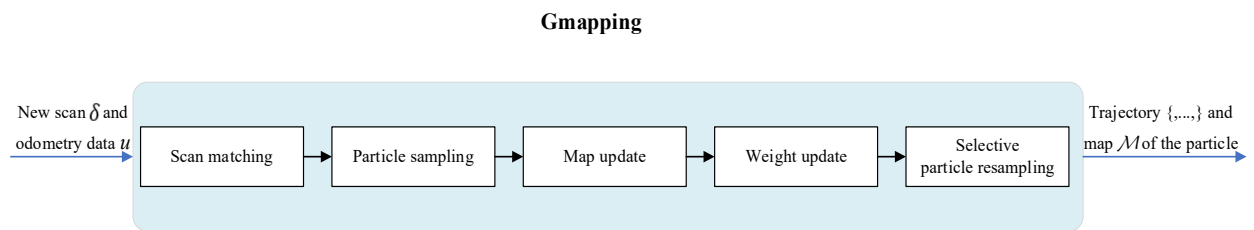


Figure 3.14: Principle of Gmapping Methods [8]

- **Scan Matching**, which utilizes laser data to refine the robot's estimated pose by compensating for errors in odometry;
- **Particle Sampling**, where new state hypotheses are generated for each particle based on motion and current observations;
- **Weight Update**, in which each particle's weight is adjusted according to the consistency between the observation data and the corresponding map;
- **Map Update**, where the occupancy grid map associated with each particle is updated based on its new state and the laser scan;
- **Selective Particle Resampling**, which is performed when necessary to preserve particle diversity and mitigate the issue of particle depletion.

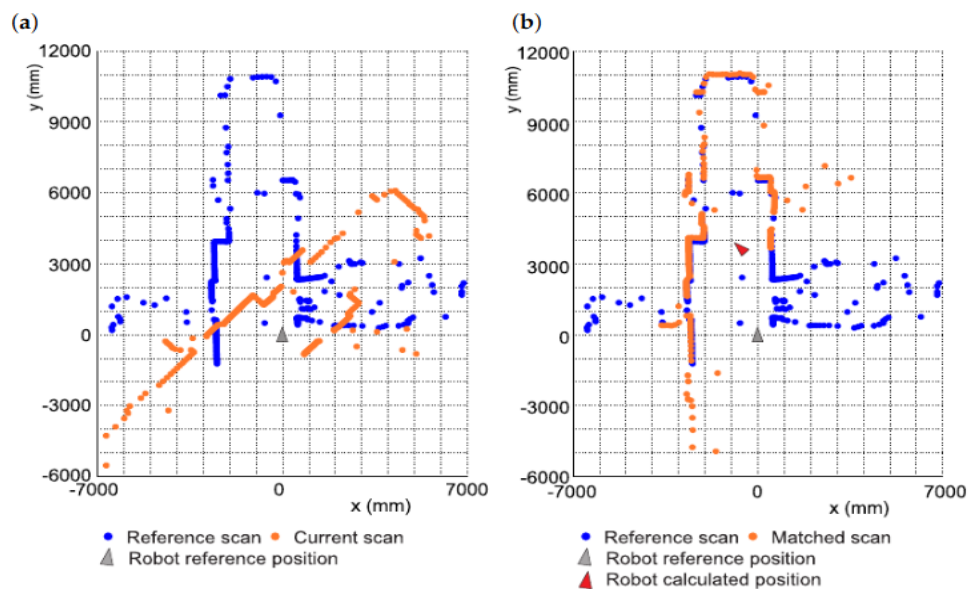
The output of the algorithm includes the environmental map $\mathcal{M}$ and the robot trajectory $\{x_t\}$, corresponding to the particle with the highest weight. The use of Effective Sample Size (ESS) as a criterion for resampling enhances both the robustness and efficiency of the algorithm. Thanks to the probabilistic integration of sensor measurements and motion information, GMapping can simultaneously construct maps and localize the robot with high accuracy in real time.

- **Scan Matching**

Definition: Scan Matching uses LiDAR or camera scans and aligns them with each other to estimate the robot's position as demonstrated in *Figure 3.15*.

Operating Principle:

- Receive two or more sets of sensor scan data: new scan data and reference data (previous scans or local map).
- Preprocessing: filter out noise (remove outliers and distortions) and down sample data to speed up processing.
- Search for alignment by combining translation and rotation to match new scan data with reference using algorithms such as ICP (Iterative Closest Point), GICP (Generalized ICP) or NDT (Normal Distribution Transform).
- Use optimization methods like Gradient Descent and RANSAC (Random Sample Consensus) to align the two data sets and eliminate incorrect matches.
- Output the estimated robot position and the aligned scan data.



*Figure 3.15: Scan matching: problem definition. (a) Scans before alignment
(b) Scans after alignment*

(Source: (PDF) Scan Matching by Cross-Correlation and Differential Evolution)

- **Particle Filter**

Definition: The Particle Filter, also known as Monte Carlo Localization (MCL) is an algorithm that distributes a number of random samples (particles)

throughout the environment that process is illustrated in *Figure 3.16*, assuming them to be the actual robot positions and gradually eliminates them over time to estimate the true position of the robot.

- Initialize particles and randomly distribute them throughout the environment with equal weights.
- Update particle positions based on the robot's odometry, adding random noise to simulate movement uncertainty.
- Compare sensor data with the position of each particle and evaluate their weights based on how closely they match the environment.
- Resample: Generate a new set of particles from the old set. Particles with higher weights attract more particles, while particles with low weights are gradually eliminated.
- This process repeats continuously as the robot moves, allowing the system to provide the most accurate estimation of the robot's position.

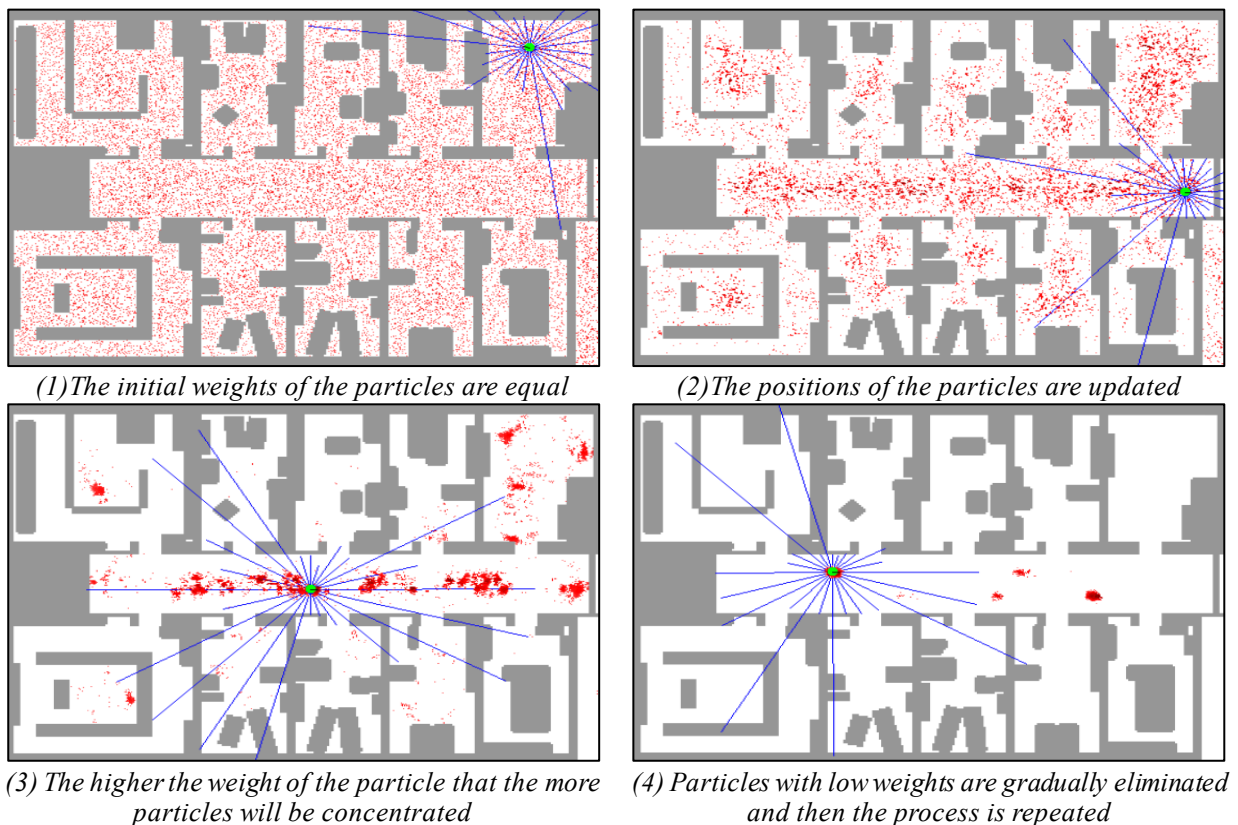


Figure 3.16: The robot is using a Particle Filter to determine position and direction represented

(Source: SLAM using Grid-based FastSLAM - fjp.github.io)

3.4 Concept of Aruco Marker

Aruco Markers are square fiducial tags designed to support computer-vision applications, particularly camera pose estimation. With a thick black border and an inner binary matrix that encodes a unique identifier, they can be robustly detected and recognized even under varying illumination or partial occlusion.

Each marker consists of a black border surrounding an inner binary grid (e.g 4×4 or 6×6). Detection proceeds by adaptive thresholding to locate candidate squares, a perspective transforms to normalize them, and bit-pattern extraction for comparison against a predefined dictionary to recover the marker ID. These dictionaries are constructed to maximize inter-marker Hamming distance, improving error-correction capabilities.

Before the introduction of Aruco Markers in 2014, prominent fiducial systems included ARToolKit (circa 1999), ARTag (around 2004) and AprilTag as demonstrated in *Figure-3.17*, whose first paper was published by a University of Michigan research group in 2011. All of these marker systems have been widely used in computer-vision tasks such as augmented reality and robot localization in both 2D and 3D spaces.

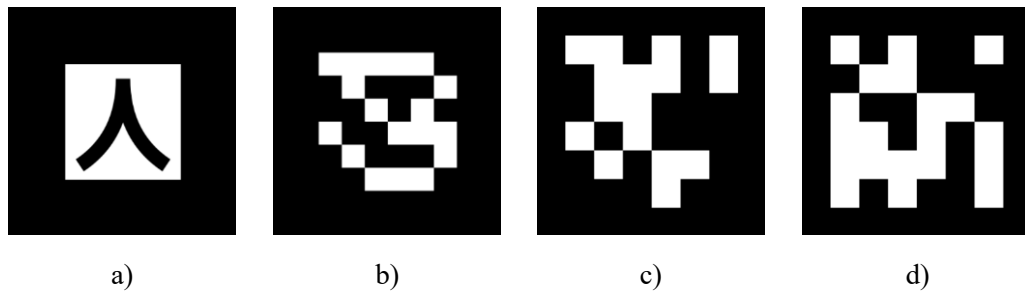


Figure 3.17: a) ARToolKit, b) ARTag, c) AprilTag, d) Aruco Marker [9]

To ensure the tags used in the simulation environment align with the project requirements, a comparison *Table 3.3* was developed to evaluate and select the most suitable options.

Table 3.3: Comparison table of fiducial marker systems

Criterion	ARToolKit	ARTag	AprilTag	Aruco Marker
Automatic code generation	No	No	Yes, it can generate codes of varying lengths with strong error-correction	Yes (with optimized Hamming-distance dictionaries)
Error correction on detection	Not clearly specified	Limited	Very good (uses Gray-code error correction; robust multi-marker detection)	Yes (auto-detects and corrects bit errors)
Support for detection when rotated	Yes, but unstable	Yes	Very stable at all angles	Very stable at all angles
Processing speed (real time)	Slower when many markers	Relatively fast	Good high performance thanks to optimized algorithms	High optimized for real-time use
Recognition when partially occluded	Poor	Medium	Very good—detects even with partial occlusion	Good, especially when using marker boards
Encoding scheme/type	Fixed image-pattern codes	Fixed binary codes	Binary, many tag families (e.g. tag36h11, tag25h9...) with high reliability	Binary; customizable grid sizes (4×4, 5×5, etc.)
Open-source library	Yes, but no longer officially maintained	Yes, but less widely used	Yes (AprilTag2/3 in C, C++, Python; well-integrated with ROS, SLAM & robotics)	Yes (Aruco in C++, Python; integrated in OpenCV and ROS)
ROS integration capability	Difficult to integrate / poor compatibility	Limited	Very strong widely used in ROS 1-2; packages available (apriltag_ros, apriltag)	Very good aruco_ros package available, easy to use

Conclusion:

- Although AprilTag offers many advantages, it is not suitable for my Capstone Project. AprilTag requires installing a separate library such as AprilTag C or wrappers like *apriltag_ros*. Configuring and integrating it with ROS is more complex especially when you need to customize or process marker outputs. As a result, using AprilTag in this project would degrade my hardware performance. The most appropriate choice, therefore, is the Aruco Marker.

Chapter 4: SIMULATION OF AUTONOMOUS ROBOT USING ROS

4.1 Relocalization Program Design

4.1.1 *Dependent Packages in the Program*

To implement the Aruco Marker algorithm effectively within the ROS environment, it is necessary to integrate the following key dependent packages: Camera (RealSense D435i), Odometry and Position. These packages will supply image data, kinematic localization and the robot's spatial coordinates. When combined with the Aruco Marker algorithm, the accuracy and localization capability of the autonomous robot will be significantly enhanced.

Image processing package for the Intel RealSense D435i camera:

ROS package name: `realsense2_camera`

Function: This package enables retrieval of data from the Intel RealSense D435i camera, including:

- Color image (RGB Image) (`/camera/color/image_raw`)
- Depth image (Depth Image) (`/camera/depth/image_rect_raw`)
- Camera information (Camera Info) (`/camera/color/camera_info`)

To launch the camera: `roslaunch realsense2_camera rs_camera.launch`

Odometry Processing Package:

ROS package name: `nav_msgs/Odometry`

Function: Provides the robot's position and velocity information based on sensors such as:

- Wheel encoders
- LiDAR
- Camera SLAM

Message:

- /odom: Robot localization data (position, orientation, velocity)
- /tf: Transforms between coordinate frames (map $\rightarrow$ odom $\rightarrow$ base_link)

Position Processing Package:

ROS package name: tf2_ros

Function: Determines the position of the robot or objects in the global coordinate frame:

- Supports retrieving position information from Aruco Markers.
- Works with *tf2* to transform between coordinate frames (world, camera, base_link).

4.1.2 Overall system algorithm flowchart

In order to give a comprehensive overview of the system's operation, a flowchart has been constructed and presented in *Figure 4.1*.

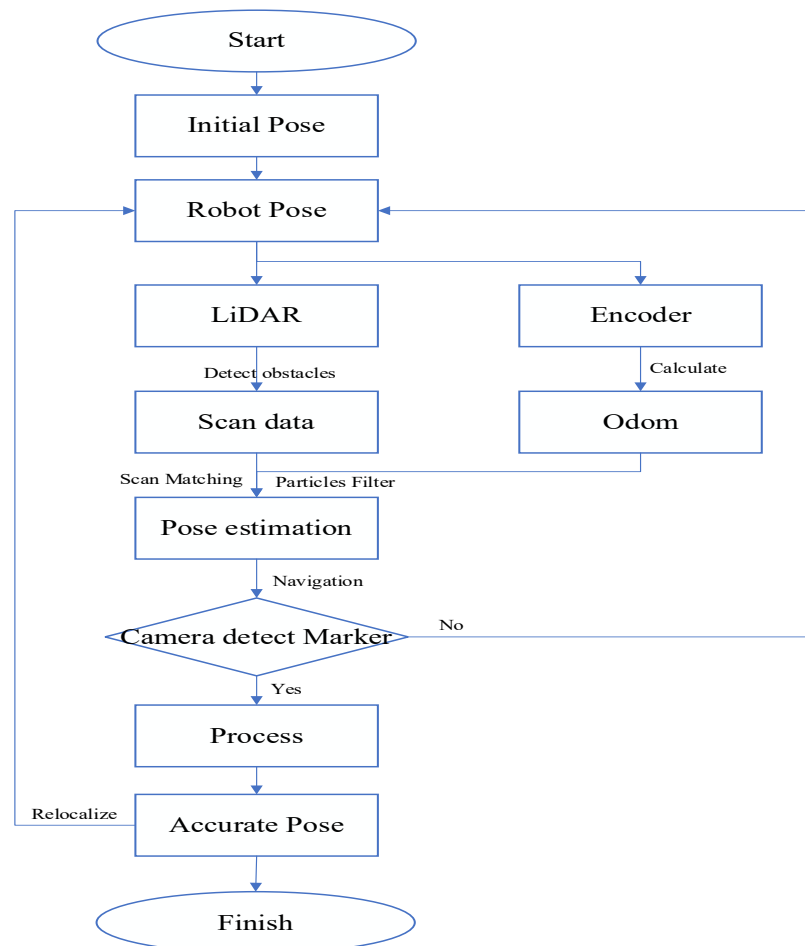


Figure 4.1: Overall system algorithm flowchart

4.1.3 Develop the mathematical model

After obtaining the coordinates of the Aruco Marker, the distance from the camera to the marker can be calculated. Additionally, since we also know the distance from the center of the robot to the camera, the distance from the robot to the marker can be inferred. With $\mathbf{H}$ as a transformation matrix that includes both rotations, translations or both to determine its position on the map.

$$\mathbf{H}\text{-Transformation matrix} = \text{Rotation} * \text{Translated}$$

To perform the rotation or translation of the Robot around a specific coordinate axis in a 3D environment, it can be defined using the following matrix:

The rotation around the Z-axis by an angle θ is represented by the following matrix: [10] [11]

$$Rot(z, \theta^\circ) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In there:

- θ is the angle of rotation.
- (x, y) is the translation vector.

The translation from the current position of the robot with coordinates x , y and z is represented by the following matrix: [10] [11]

$$Trans(x, y, z) = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The method for calculating the position of the robot based on Aruco:

- The camera detects the marker and calculates the relative coordinates of the marker with respect to the camera:

$$P_{cam} = [x_c, y_c, z_c, 1]$$

- Perform a transformation with the robot's position to determine the new position of the robot by:

$$P_{new} = Trans(x_c, y_c, 0) * Rot(z, \theta) * P_{cam} \quad (\forall \theta = \arctan \frac{x_c}{y_c})$$

4.2 Programming models

4.2.1 Model programming

The URDF model definition file uses Xacro tags, allowing code reuse through macros and robot descriptions: mechanical structure, linkage, joints, sensors, material, basic physical model. I named this file is “test.urdf.xacro”

- ❖ The robot is named 'model' and consists of the following components:
 - A chassis (base) that serves as the main structural frame of the robot.
 - Two drive wheels (left and right), powered by motors.
 - Two caster wheels (free-rotating), located at the front and rear to maintain balance.
 - A Lidar sensor for 2D distance measurement and environment mapping.
 - A depth camera for capturing RGB-D images, supporting 3D perception and navigation.

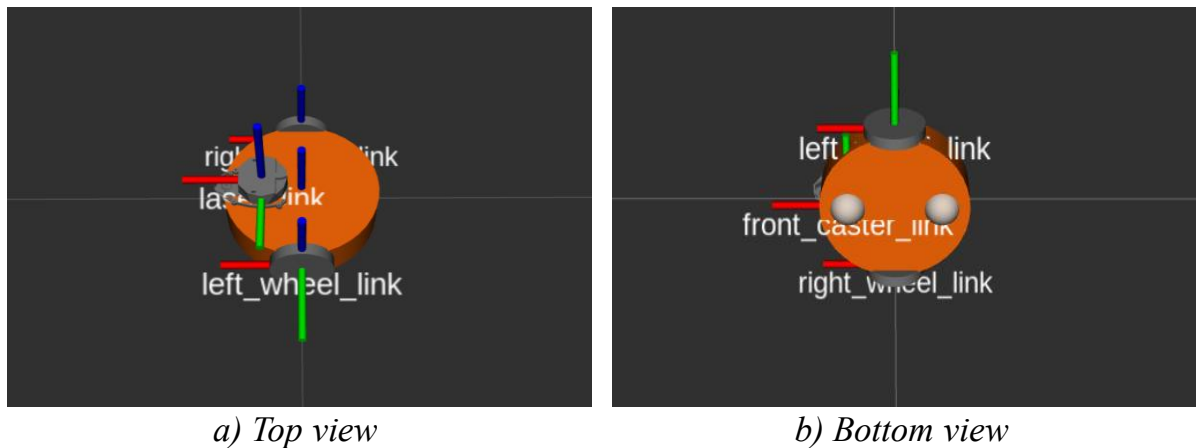


Figure 4.2: My robot in RViz

- ❖ Details of each component are summarized in Table 4.1:

Table 4.1: Details of the tags that build the robot

Tag	Function & Detailed Description
<robots>	The root tag encompasses the entire robot model. name="model" is the name of the robot. Declare the xmlns:xacro namespace to use the Xacro (XML macro) syntax. All components such as links, joints, and macros are located here.
<xacro:include>	Used to embed other Xacro files. Example: • model.gazebo.xacro: contains the Gazebo parameters or plugins.

	• <code>_d435i.urdf.xacro</code>: contains macros for attaching the RealSense D435i sensor.
<code><material></code>	Define colors that can be reused multiple times in the visual section. Each material has a name (blue, grey, etc.) and a color attribute <code>rgba="R G B A"</code> .
<code><link></code>	Represents a specific physical component of the robot (e.g., base_link, laser_link, wheels, wheels, wheels). A link can have the following sections: • inertial: mass & moment of inertia • Visual: Display • Collision: Simulate collision
<code><joint></code>	Link between two links. There are many types of joints such as: • Fixed: Fixed • Continuous: Continuous rotation (for wheels) • Revolute: Rotation angle limit Contains information: • parent, child: link between two links • Origin: Relative location • axis, limit, dynamics: motion attributes
<code><visual></code>	Determine how to display the geometry of a link in RViz, Gazebo. It can be a simple shape (cylinder, box, sphere) or mesh (STL, DAE file).
<code><collision></code>	Similar to <code><visual></code> , but used to simulate a physical collision. The geometry is usually simpler to increase simulation performance.
<code><inertial></code> or <code><xacro:cyl_inertia></code> , <code><xacro:box_inertia></code> , <code><xacro:sphere_inertia></code>	Declare physical parameters such as: • Mass: Mass • inertia: moment of inertia. In Xacro, you use macros like <code>cyl_inertia</code> to simplify automatic calculations.
<code><gazebo reference="..."></code>	Specific configurations for simulation in Gazebo such as: • Material: Color Display • Mu1, Mu2: Friction coefficient • Control plugins if available
<code><xacro:macro name="..." params="..."></code>	Macro definitions (reusable templates) in Xacro. Parameters can be passed in to create repeatable components such as wheels, support wheels.
<code><xacro:caster></code> , <code><xacro:wheel></code>	Call the declared macro to create the corresponding object. For example, a caster is a ball caster, a wheel is a motor wheel. Transmit parameters such as prefix to distinguish names, pos_x, pos_y to locate them.
<code><origin rpy="..." xyz="..."></code>	Define the relative position between the joint and the link (or sensor), with xyz being the coordinates (meters) and rpy being the angle of rotation in roll-pitch-yaw (radian) order.

<code><axis xyz="..."></code>	With a rotary joint, the axis of rotation is defined (e.g., the wheel usually rotates around the Y axis: 0 1 0).
<code><limit></code>	Joint movement limits such as: • Effort: Maximum force • Velocity: maximum speed
<code><dynamics></code>	Describe the kinematic properties of the joint such as: • Damping: Damping • Friction: Friction
<code><geometry></code>	Describe the geometry for the visual or collision section, which can be: • box, cylinder, sphere • mesh: use STL or DAE files
<code><xacro:sensor_d435i></code>	The macro is defined in the <code>_d435i.urdf.xacro</code> file, which attaches the Intel RealSense D435i sensor to the robot. Parameters such as location (origin) and usage mode can be transmitted.
<code><mesh filename="..."></code>	Used to download the 3D model of the element from the STL file. In the example, <code>lds.stl</code> is the model of the LDS Lidar.
<code><color rgba="..."></code>	It belongs to the <code><material></code> tag, which defines colors according to the RGBA standard with values from 0 to 1.

An example code is *Figure 4.3*:

```

113 </xacro:macro>
114
115 <xacro:wheel prefix = "left" pos_y="1"/>
116 <xacro:wheel prefix = "right" pos_y="-1"/>
117
118 <joint name="laser_joint" type="fixed">
119   <parent link="base_link"/>
120   <child link="laser_link"/>
121   <origin xyz="${base_radius/2} 0.0 ${base_height/2 + laser_height/2}" rpy="0.0 0.0 0.0"/>
122 </joint>
123
124 <link name="laser_link">
125   <xacro:box inertia mass = "${laser_mass}" x = "${laser_length}" y = "${laser_width}" z = "${laser_height}"/>
126
127   <visual>
128     <geometry>
129       <mesh filename="package://mascot/meshes/lds.stl" scale="0.002 0.002 0.002"/>
130     </geometry>
131     <material name="grey"/>
132   </visual>
133
134   <collision>
135     <geometry>
136       <mesh filename="package://mascot/meshes/lds.stl" scale="0.002 0.002 0.002"/>
137     </geometry>
138   </collision>
139 </link>
140
141 <gazebo reference="laser_link">
142   <material>Gazebo/Grey</material>
143 </gazebo>
144
145 <xacro:sensor_d435i parent="base_link" name="d435i" use_nominal_extrinsics="true">
146   <origin xyz="-0.011 -0.018 0.137" rpy="0 0 0"/>
147 </xacro:sensor_d435i>
148
149
150 </robot>

```

Figure 4.3: The red frame creates wheel's name; The green frame was creates a joint; The blue frame designed LiDAR details.

In red frame call prefix to create wheel's name is left and right along the Y axis.

In green frame created a joint its name is "laser_joint" and fixed type (not moving), main body is "base_link" and child link is LiDAR "laser_link". The LiDAR is positioned at

half the radius of the cylindrical body ($\text{base_radius}/2$) along the x-axis, meaning it is mounted at the front. Along the y-axis, the value $y=0.0$ indicates that the LiDAR is centered horizontally. The z-axis coordinate is calculated as $\text{base_height}/2 + \text{laser_height}/2$, meaning the LiDAR is mounted directly on top of the robot's body, with its center aligned with the top surface of the chassis. (*Appendix 5 provides detailed information on the moment of inertia*)

In blue frame, this is the part that is designed for LiDAR details, named “laser_link”. Macro calculation of inertia of a rectangular object, including: x/y/z: length/width/height and mass (volume).

In pink frame added the D435i camera and attached it to the robot's body. Name the sensor “d435i” and use the default calibration parameters.

4.2.2 Environment programming

Unlike robot configuration written in Xacro, the environment is described in an SDF file.

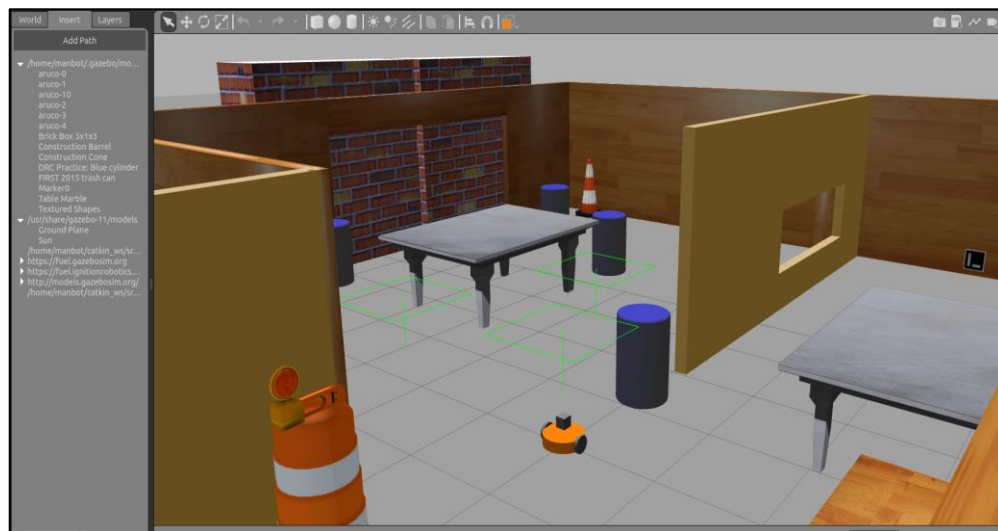


Figure 4.4: Testing environment for this Capstone Project

This can be done by writing an SDF file manually or arranging objects directly in Gazebo using built-in and external libraries. Once the 3D design is complete, an SDF file is generated, describing the world (world), objects (model), sensors, environment (land, sky, light) and plugins.

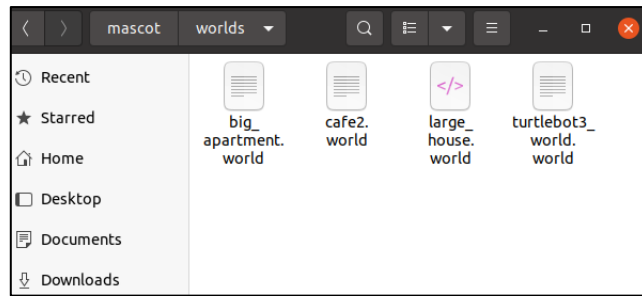


Figure 4.5: My configuration SDF file is named 'big_apartment.world'

The main tag in an SDF file to create simulation environment as shown in Table 4.2 include: `<world>`, `<light>`, `<model>`, `<pose>`, `<link>`, `<collision>`, `<visual>`, `<geometry>`, `<material>`, `<inertial>`, `<gravity>`, `<physics>`, `<scene>`.

Table 4.2: Details of the tags that build the simulation environment

Tag	Function / Description
<code><sdf version='1.7'></code>	Original card, declaring the SDF version in use (here is 1.7).
<code><world name='...'></code>	The main simulated world description, including all objects, lights, robots, sensors...
<code><light></code>	Create and configure light sources (such as sun, spotlight).
<code><model></code>	Represents a model (object, robot, ground, obstacle...).
<code><pose></code>	Determine the position and orientation (x, y, z, roll, pitch, yaw) of the model in space.
<code><link></code>	The subcomponent of the model, which represents a physical part (with collisions, displays, etc.).
<code><collision></code>	Geometric blocks are used to simulate physical collisions.
<code><visual></code>	The geometry block is used to display the model in the Gazebo.
<code><geometry></code>	Declare the shape of the object (box, sphere, cylinder, mesh...).

<material>	Set the color or material to display for the object.
<inertial>	Describe mass and inertia for physical calculations.
<gravity>	The gravity setting in the simulation is 0 0 -9.8
<physics>	Physical engine configuration: simulation speed, accuracy, time step.
<scene>	Set the backlight, background color of the scene.
<spherical_coordinates>	Setting up a global spherical coordinate system (used for outdoor simulations, GPS, etc.).
<state>	Save the current state of all models (pose, velocity, time).
< plugin>	Attach plugins to expand functions such as robot control, ROS sensors, etc.

The configuration SDF file used in this project is named 'big_apartment.world'.

```

1  <sdf version='1.7'>
2    <world name='default'>
3      <light name='sun' type='directional'>
4        <cast_shadows>1</cast_shadows>
5        <pose>0 0 10 0 -0 0</pose>
6        <diffuse>0.8 0.8 0.8 1</diffuse>
7        <specular>0.2 0.2 0.2 1</specular>
8        <attenuation>
9          <range>1000</range>
10         <constant>0.9</constant>
11         <linear>0.01</linear>
12         <quadratic>0.001</quadratic>
13       </attenuation>
14       <direction>0.5 0.1 -0.9</direction>
15       <spot>
16         <inner_angle>0</inner_angle>
17         <outer_angle>0</outer_angle>
18         <falloff>0</falloff>
19       </spot>
20     </light>

```

a)

```

6031    </link>
6032    <pose>-4.49485 -3.01315 0 0 -0 0</pose>
6033  </model>
6034  <model name='aruco-2'>
6035    <static>1</static>
6036    <link name='link'>
6037      <visual name='visual'>
6038        <geometry>
6039          <mesh>
6040            <uri>model://aruco-2/meshes/aruco-2.dae</uri>
6041            <scale>0.6 0.6 0.6</scale>
6042          </mesh>
6043        </geometry>
6044      </visual>
6045      <self_collide>0</self_collide>
6046      <enable_wind>0</enable_wind>
6047      <kinematic>0</kinematic>
6048    </link>
6049    <pose>-4.49974 -3.35469 0 0 -0 0</pose>
6050  </model>
6051  <model name='aruco-3'>
6052    <static>1</static>
6053    <link name='link'>

```

b)

Figure 4.6: a) The <light> tag is used to configure lighting settings; b) The Aruco-2 marker is defined within the <model> tag, with a scale factor of 60% applied along the x, y, and z axes

And the simulation scene includes models such as *ground_plane*, *Building*, *table_clone*, *bookshelf*... which together create a complex and diverse environment to test the robot's performance.

In autonomous robot development, building a virtual simulation environment is essential for testing positioning and navigation algorithms before real-world deployment.

4.2.3 Realistic Parameters for Robot

From both of these files: `big_apartment.world` and `test.urdf.xacro`. In *Figure 4.7*, the yellow frame is an example declaration for the robot's body

```
home > manbot > catkin_ws > src > mascot > urdf > test.gazebo.xacro
1 <?xml version="1.0"?>
2 <robot xmlns:xacro="http://www.ros.org/wiki/xacro">
3
4 <xacro:property name="base_height" value="0.1" />
5 <xacro:property name="base_radius" value="0.2" />
6 <xacro:property name="base_mass" value="5" />
7
8 <xacro:property name="wheel_height" value="0.03" />
9 <xacro:property name="wheel_radius" value="0.08" />
10 <xacro:property name="wheel_mass" value="0.8" />
11
12 <xacro:property name="caster_radius" value="0.05" />
13 <xacro:property name="caster_mass" value="0.5" />
14
15 <xacro:property name="laser_length" value="0.06" />
16 <xacro:property name="laser_width" value="0.06" />
17 <xacro:property name="laser_height" value="0.065" />
18 <xacro:property name="laser_mass" value="0.5" />
19
```

Figure 4.7: An example declaration for the robot's body

The physical parameters of each component of the robot such as the height, radius, and mass of the base, wheels, caster wheel and laser sensor are declared as illustrated in *Figure 4.8*. These values will be used in the subsequent sections to construct the geometric model and to compute inertial properties.

```
home > manbot > catkin_ws > src > mascot > urdf > test.gazebo.xacro
2 <robot xmlns:xacro="http://www.ros.org/wiki/xacro">
19
20 <xacro:macro name="cyl_inertia" params="mass r h">
21 <inertial>
22 <mass value="${mass}" />
23 <inertia ixx="${(1/12)*mass*(3*r*r+h*h)}" ixy="0.0" ixz="0.0"
24 iyy="${(1/12)*mass*(3*r*r+h*h)}" iyz="0.0"
25 izz="${(1/2)*mass*r*r}" />
26 </inertial>
27 </xacro:macro>
28
29 <xacro:macro name="box_inertia" params="mass x y z">
30 <inertial>
31 <mass value="${mass}" />
32 <inertia ixx="${(1/12)*mass*(y*y+z*z)}" ixy="0.0" ixz="0.0"
33 iyy="${(1/12)*mass*(x*x+z*z)}" iyz="0.0"
34 izz="${(1/12)*mass*(y*y+x*x)}" />
35 </inertial>
36 </xacro:macro>
37
38 <xacro:macro name="sphere_inertia" params="mass r">
39 <inertial>
40 <mass value="${mass}" />
41 <inertia ixx="${(2/5)*mass*r*r}" ixy="0.0" ixz="0.0"
42 iyy="${(2/5)*mass*r*r}" iyz="0.0"
43 izz="${(2/5)*mass*r*r}" />
44 </inertial>
45 </xacro:macro>
46
```

Figure 4.8: The yellow, pink, and red frames respectively represent the formulas for calculating the moment of inertia of a cylinder, a box and a sphere

In this macro I used a physics formula to calculate the moment of inertia for cylindrical, rectangular and spherical objects.

4.2.4 Developing image processing

4.2.4.1 Designing the Aruco Marker image processing workflow

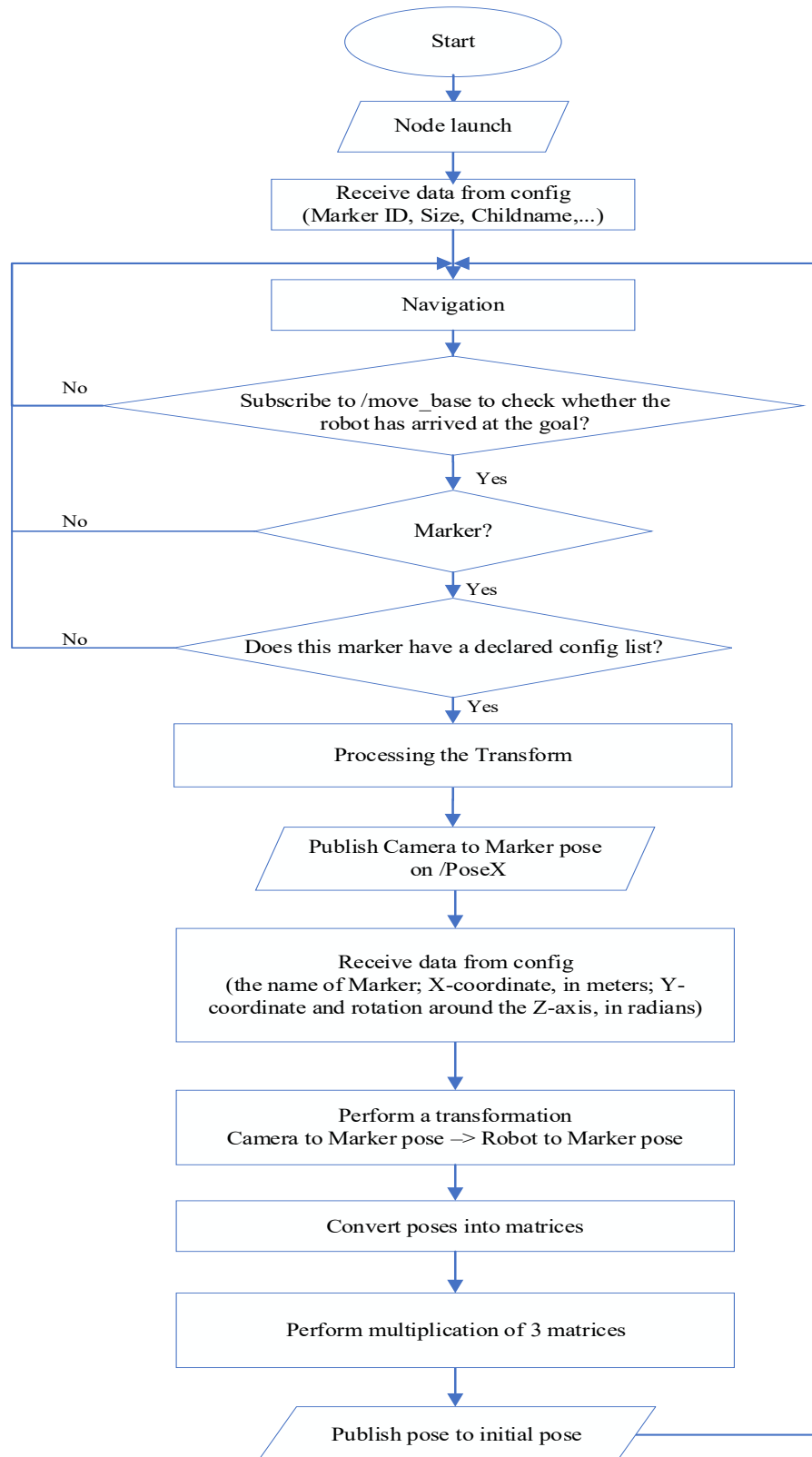


Figure 4.9: Aruco Marker image processing flowchart

4.2.4.2 Detection Aruco Marker tag

Figure 4.10 below illustrates the algorithm flowchart of the *multiple.cpp* node.

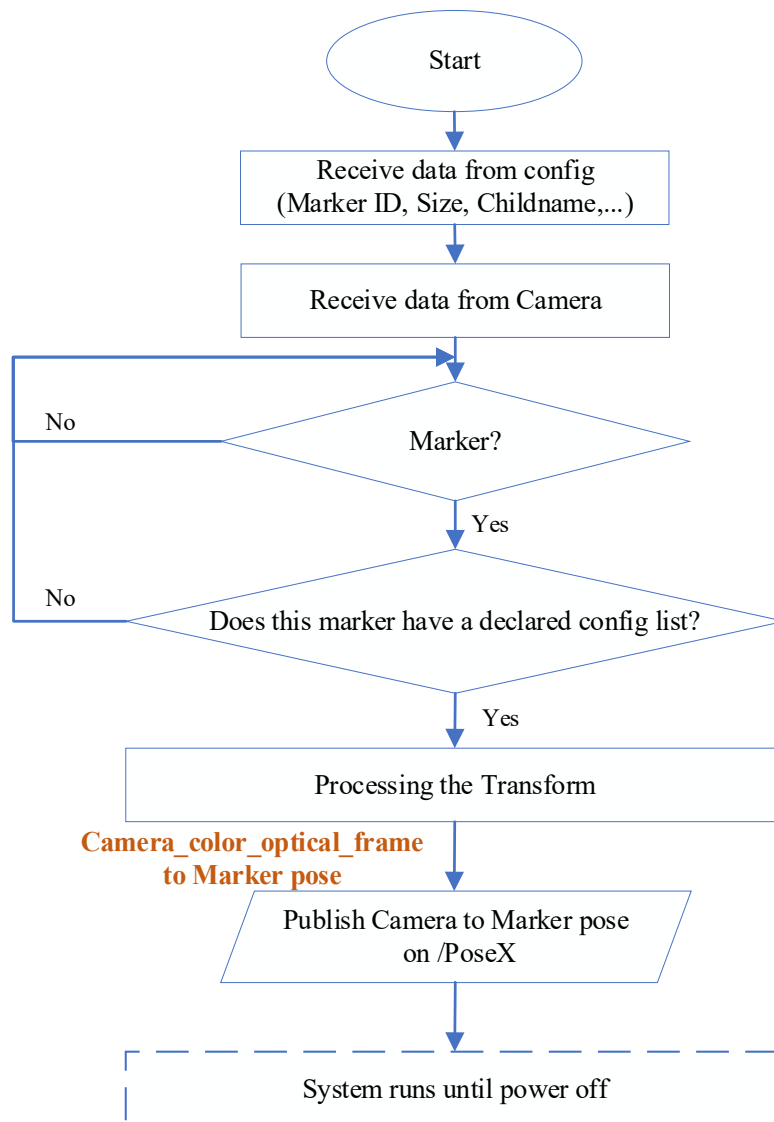


Figure 4.10: Detection Aruco Marker tag flowchart

- Initializing the ROS Node and necessary parameters. The *aruco_simple* node is initialized. It reads parameters from a launch file such as:
 - `marker_ids`: List of marker IDs to detect
 - `child_names`: Names of corresponding child frames for each marker
 - `parent_name`: Name of the root frame (typically camera or map)
 - `marker_size`: Actual size of the marker (in meters)
 - `image_is_rectified`: Whether the input image has been rectified or not.

Meanwhile, the node initializes dynamic reconfigure to adjust the marker detection algorithm in real-time.

- It subscribes to receive images and camera information. The node subscribes to:
 - Images from */image* topic
 - Camera calibration information from the */camera_info* topic
 - Upon receiving CameraInfo, the node converts it to Aruco parameters and then unsubscribes to save resources.

- Marker detection and pose calculation. When there is an input image, the node uses the Aruco library to:

Whenever a new image is sent to the */image* topic, the *image_callback* is invoked to process it. Here, the Aruco library is used to detect markers in the image. For each detected marker, the node checks if the marker's ID is in the list to be tracked. If it is, the node calculates the pose of the marker relative to the camera, then applies a 180-degree rotation (yaw axis) to correct the orientation of the marker's coordinate frame. The result is a Transform passed through the ROS TF system, and the pose of the marker is also sent as a package containing standard messages (*geometry_msgs/Pose*) to separate topics like */pose0*, */pose1*, etc.

- The node sends the resulting image to ROS topics for debugging. It uses *image_transport* to send two types of images: a colored image with marker information and a 3D block sent to the */result* topic, and a binary image (thresholded image) sent to the */debug* topic. The debug image is a useful tool to check the algorithm's effectiveness under different lighting conditions, while the resulting image helps visually observe what the node is detecting.

4.2.4.3 Automatic relocalization after reaching Navigation goal

I have created a dedicated node for the relocalization as demonstrated in *Figure 4.11* task using Aruco Marker to calculate and update the position on the map: *aruco_relocalizer.py*.

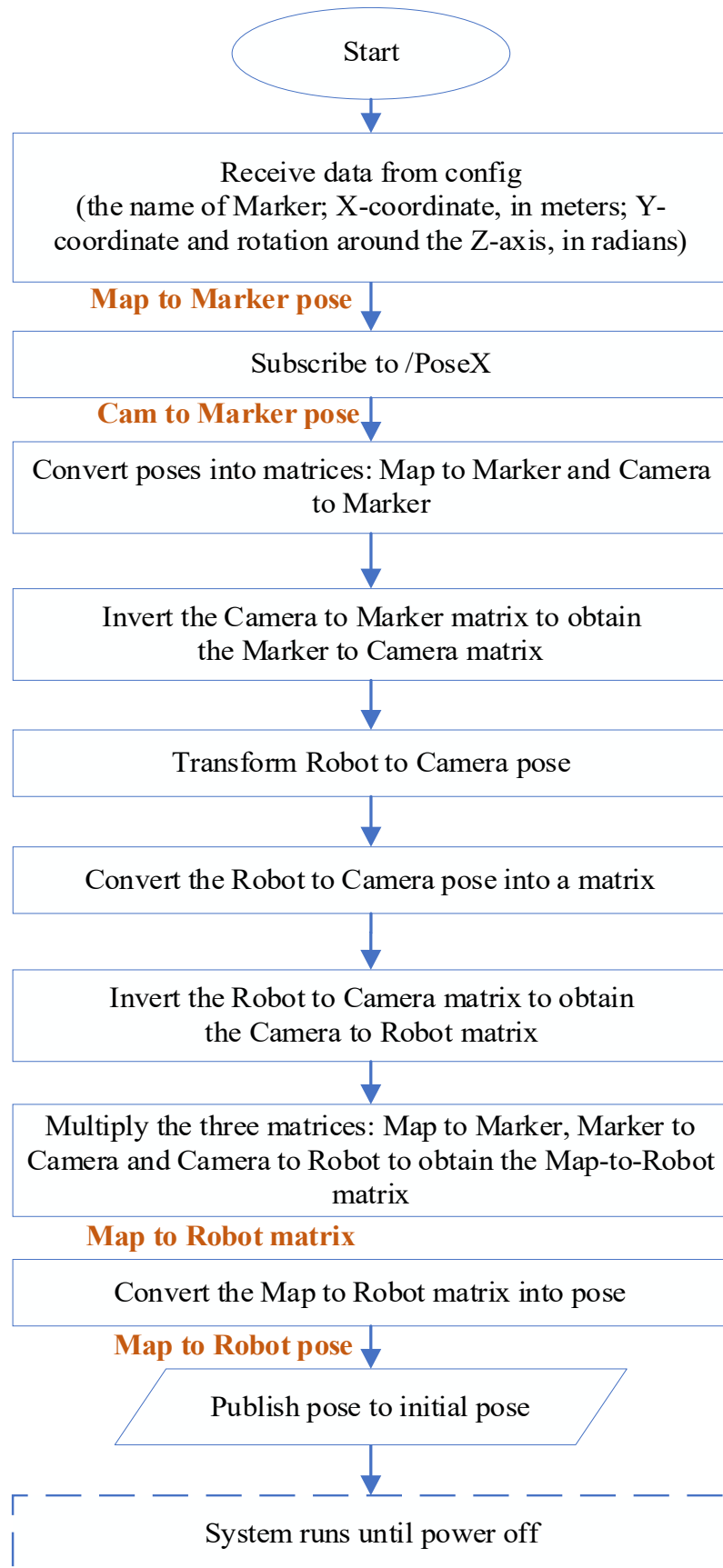


Figure 4.11: Relocalization flowchart

To relocalize automatically, I created an additional node as shown in *Figure 4.12* called *goal_relocalize.py* to indirectly call the node *aruco_relocalizer.py* after the navigation process is complete.

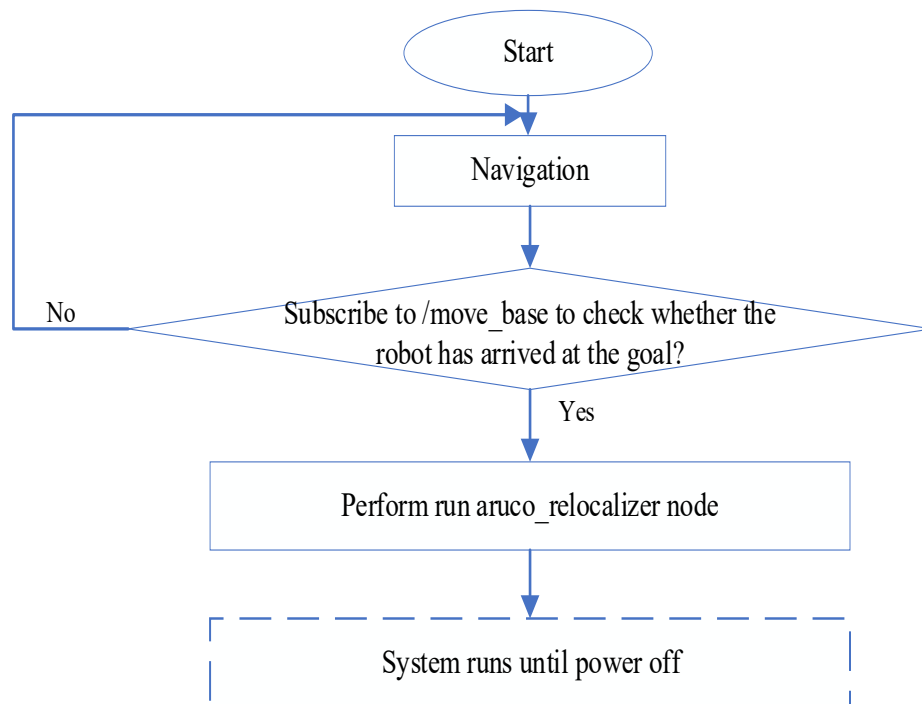


Figure 4.12: Automatic relocalization flowchart

The algorithm begins at the **Start** block, marking the initialization of the automatic relocalization procedure. Once started, the robot proceeds with **navigation** to reach the predefined goal using a navigation system such as ROS's *move_base*.

And then, the system subscribes to the */move_base* topic to monitor the robot's progress. This step checks whether the robot has successfully arrived at the goal position. At this stage, the algorithm evaluates two possibilities:

- If the **robot has not yet reached the goal** (branch “No”), the system continues to loop and monitor the goal status.
- If the **robot has arrived at the goal** (branch “Yes”), the process advances to the next step.

Upon confirming that the robot has reached its destination, the system triggers the *aruco_relocalizer.py* node. This node uses the camera to detect Aruco Markers and performs a relocalization to refine the robot's pose with higher accuracy.

Chapter 5: RESULTS AND ANALYSIS IN THE SIMULATED ENVIRONMENT

5.1 Map scanning and saving in Gazebo simulation

After creating the Gazebo simulation environment with the markers, parameters, etc., I saved it in a package named "mascot". I will launch the "mascot" package to open the Gazebo environment on the screen with the command: `"roslaunch mascot gazebo.launch"`.

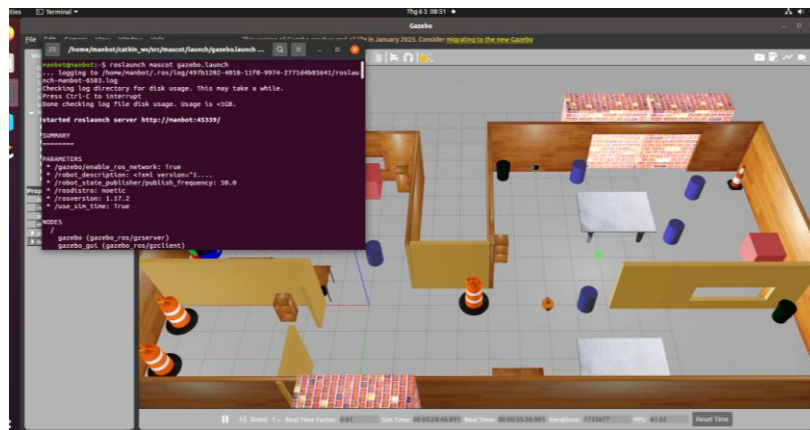


Figure 5.1: The Gazebo environment appears on the screen after executing launch

Next, I perform a LiDAR scan of the Gazebo environment to visualize it in 3D in the RViz environment, aiding in monitoring the robot's status, including images, point clouds, trajectories, or transformations between different coordinates. To accomplish this, I use the command: `"roslaunch gmapping slam_gmapping_pr2.py"`.

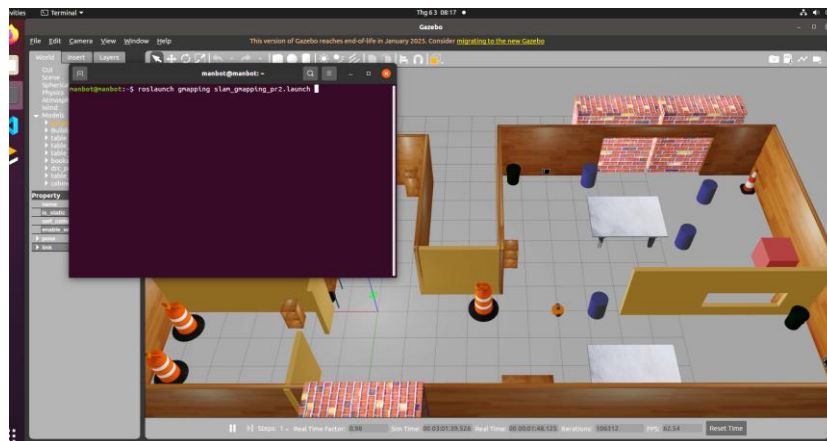


Figure 5.2: I am commanding scan map on Gazebo

At this point, the robot is already mapping, but to observe this process, I need to enter the command "rviz" to display the RViz environment on the screen. (*Details about RViz environment in Appendix 6*)

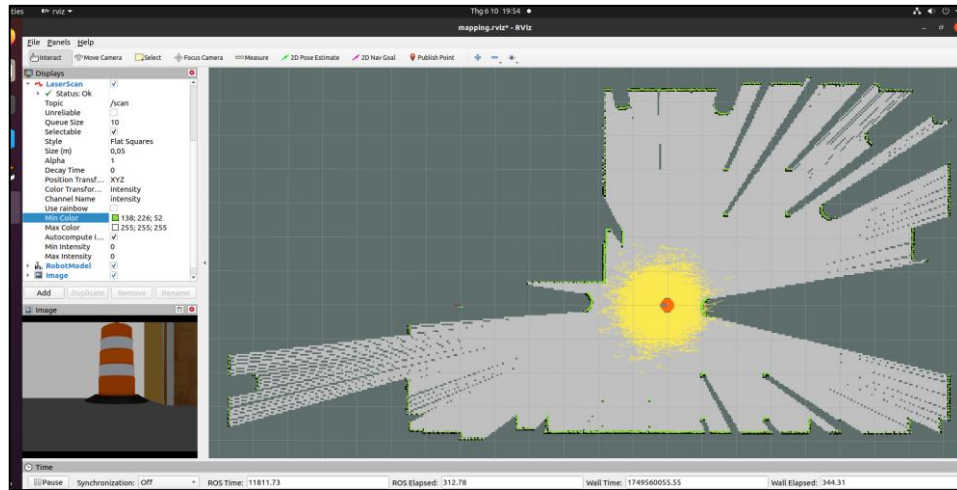


Figure 5.3: The robot is currently scanning the surrounding environment with Particle Filter, represented by small yellow arrows

I have already created a Python file to control the robot using the keyboard with the command: "roslaunch teleop_twist_keyboard teleop_twist_keyboard.py". I start controlling the robot to move to areas where the laser beams have not scanned yet to make the map more detailed.

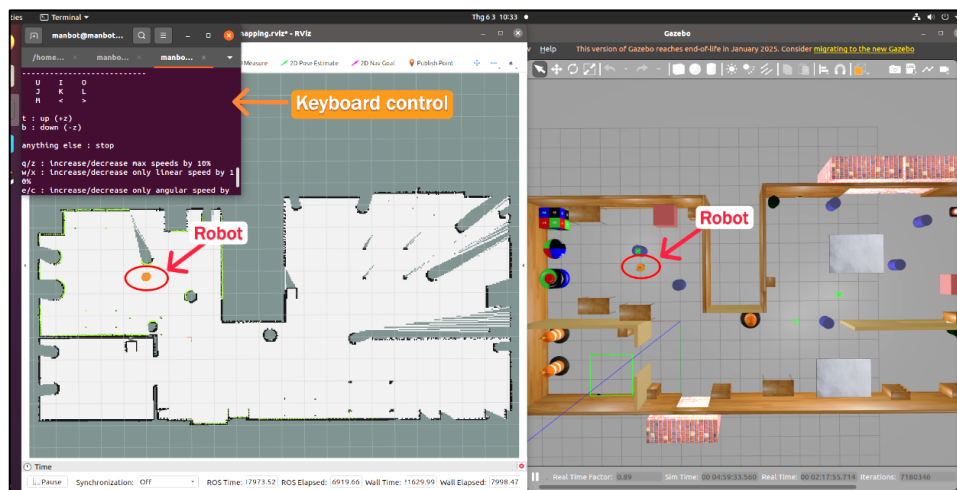
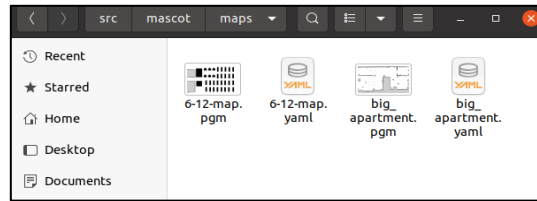


Figure 5.4: The robot's movement in Gazebo is immediately reflected in RViz; the robot's position in both environments is relatively consistent

When the scanning process is completed, I create a directory named "maps" within my "mascot" package.: "mkdir -p ~/catkin_ws/src/mascot/maps". I saved the map to this

directory with the name "*big_apartment*" using the command: `rosrun map_server map_saver -f ~/catkin_ws/src/mascot/maps/big_apartment`". (Detail about *map_server* at 8.2 in Appendix 8)



After the save command completes, I will have two types of map files generated: a map description file (.yaml) and a map image file (.pgm).

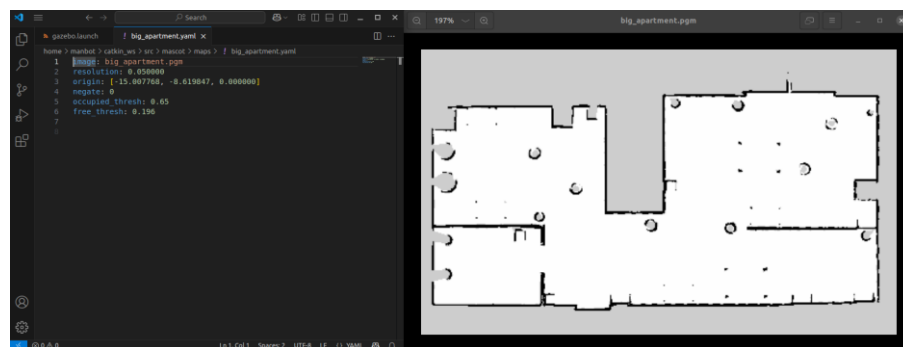


Figure 5.5: The map consists of two types of data: an image file and a metadata description file

5.2 Robot localization

After saving the map, I proceed with simulating localization. To open the previously saved RViz environment, I use the command: `roslaunch mascot navigation.launch` to test the Particle Filter and Scan Matching algorithms.

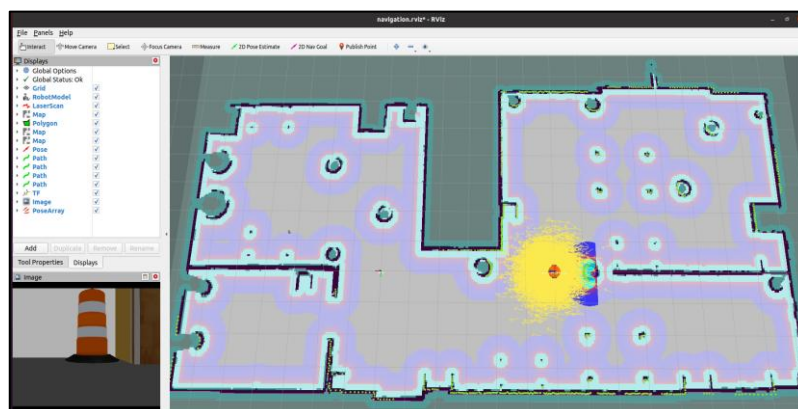


Figure 5.6: Configure the RViz environment with *GlobalMap* and *LocalMap*

We have two ways to determine the position of the robot on the map:

- View the robot's position directly in the Gazebo environment on the World tab.
- Use command “`roslaunch tf_echo map base_link`” for the position to appear in the terminal

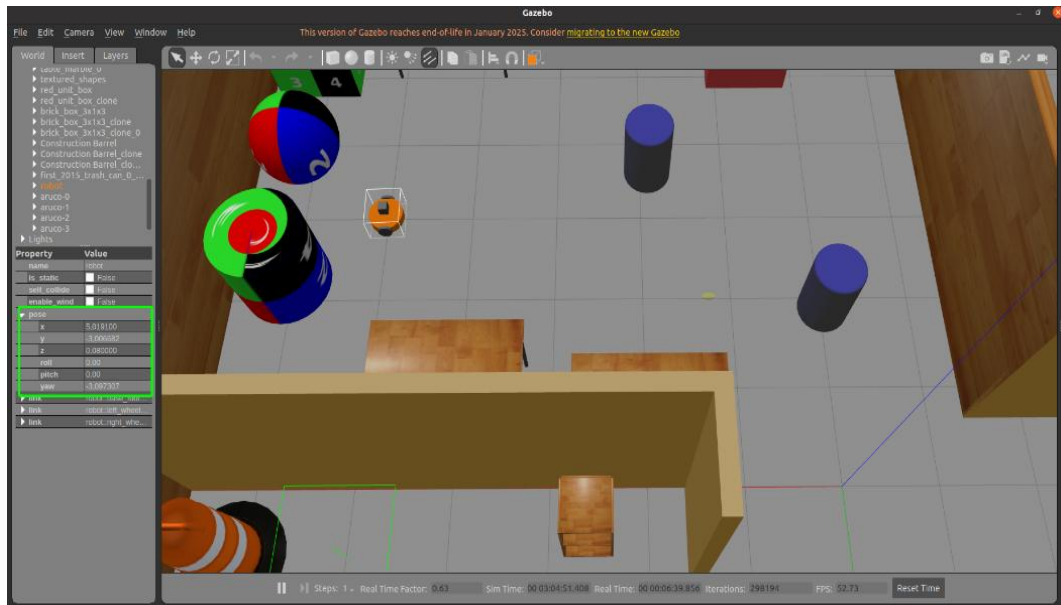


Figure 5.7: The robot's position relative to the map in Gazebo is $(x, y, z) = (5.019100, -3.006682, 0.000000)$ with a yaw of -3.097307 radians

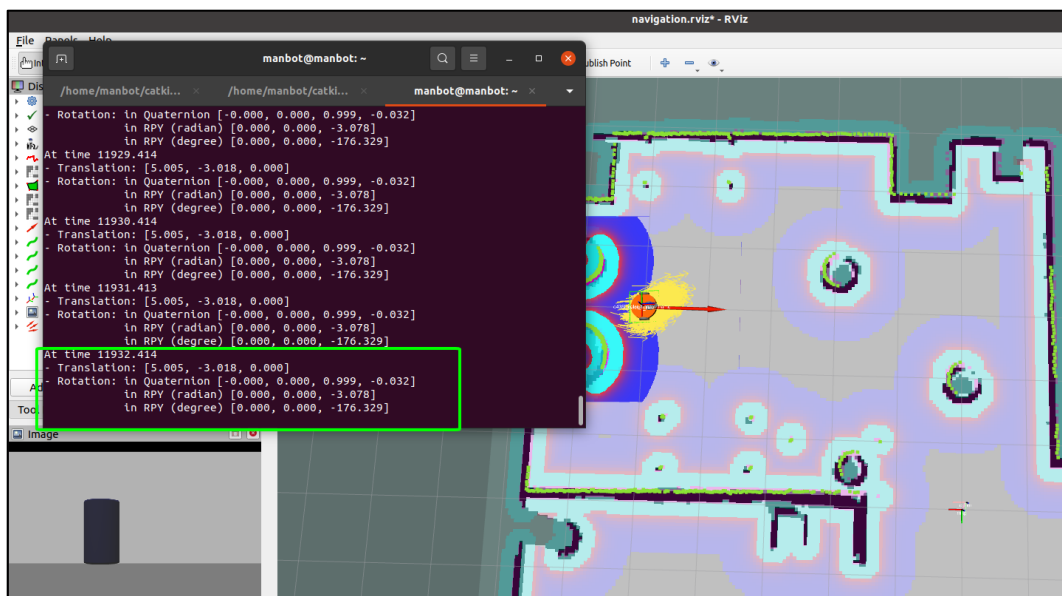


Figure 5.8 The robot's position relative to the map in RViz is $(x, y, z) = (5.006, -3.018, 0.000)$ with a converted yaw angle from radians to degrees as -176.319 degrees

5.3 Obstacle avoidance

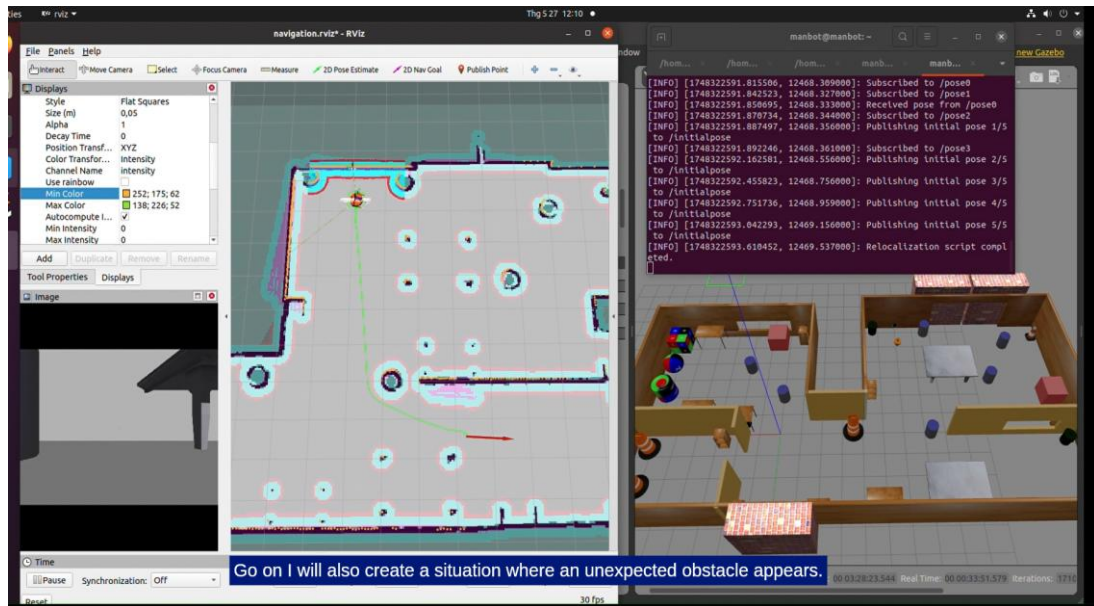


Figure 5.9: I am requesting the robot to navigate following the red directional arrow in RViz

To test the obstacle avoidance capability, I created an unexpected scenario by placing an obstacle directly on the Global Planner's path (details about Global Planner at 7.1 Appendix 7) with green line.

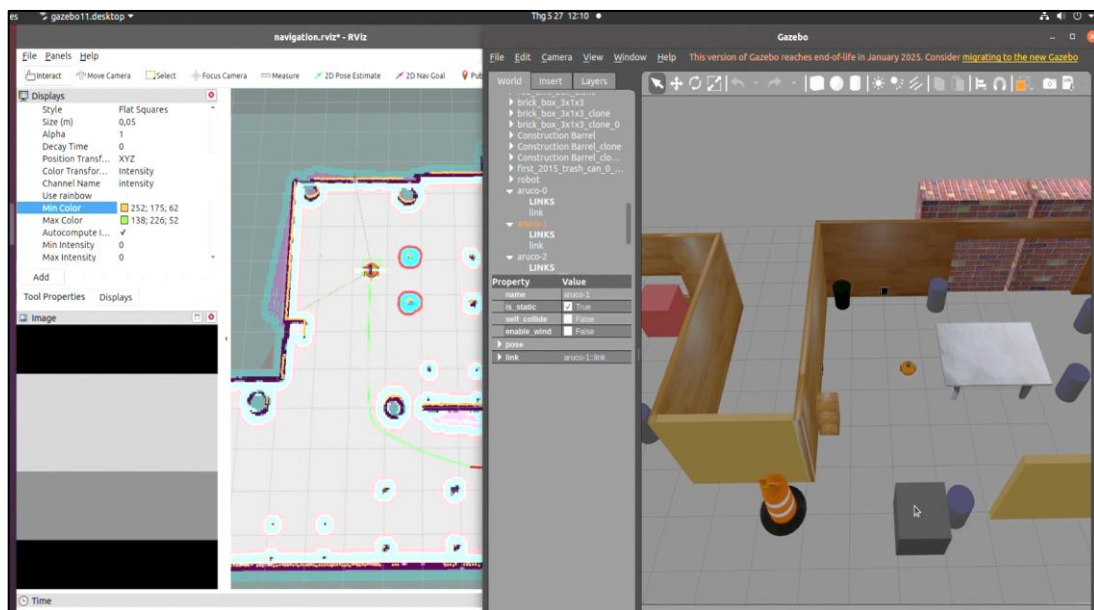


Figure 5.10: I place a black cube along the path of the robot to test obstacle avoidance algorithms

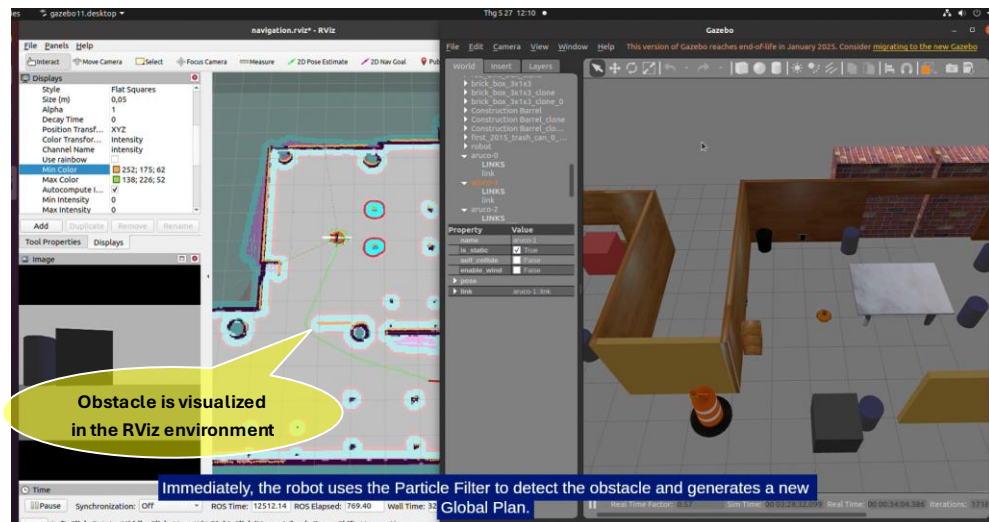


Figure 5.11: A new Global Planner has been generated by the robot

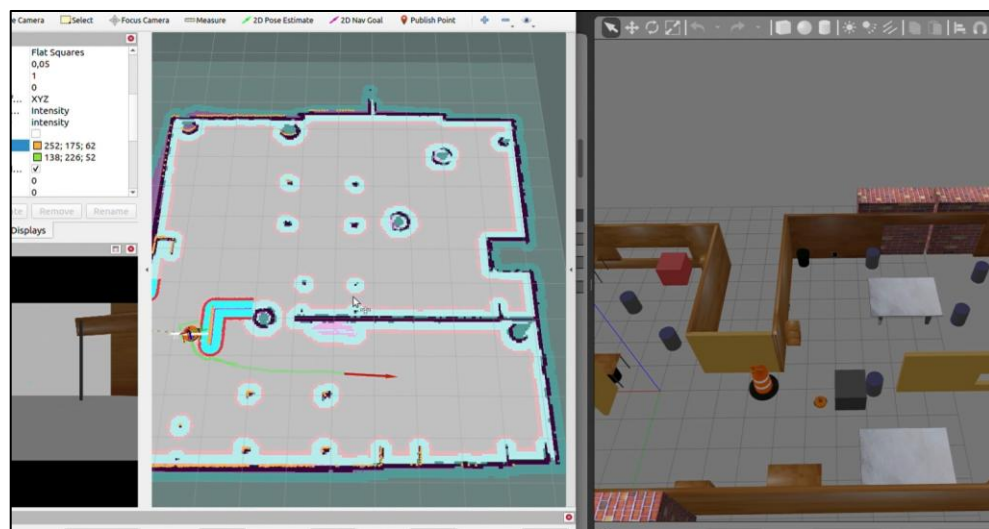


Figure 5.12: A comparison of the two environments is illustrated in this image

Simulation results show that the robot can maintain good localization performance even when the environment changes, demonstrating the stability and effectiveness of the tested localization algorithm.

5.4 Aruco Marker detection and Pose estimation

The command "`roslaunch aruco_ros multiple.launch`" is used to run the node "multiple" in the `aruco_ros` package to detect and output the pose of Aruco Markers on the screen. The node "multiple" analyzes images from the camera, recognizing Aruco Markers with IDs listed in the configuration. For each detected Marker, the node publishes a corresponding pose to topics like `/pose0`, `/pose1`, etc. Each pose represents the relative position between the camera and the Marker.

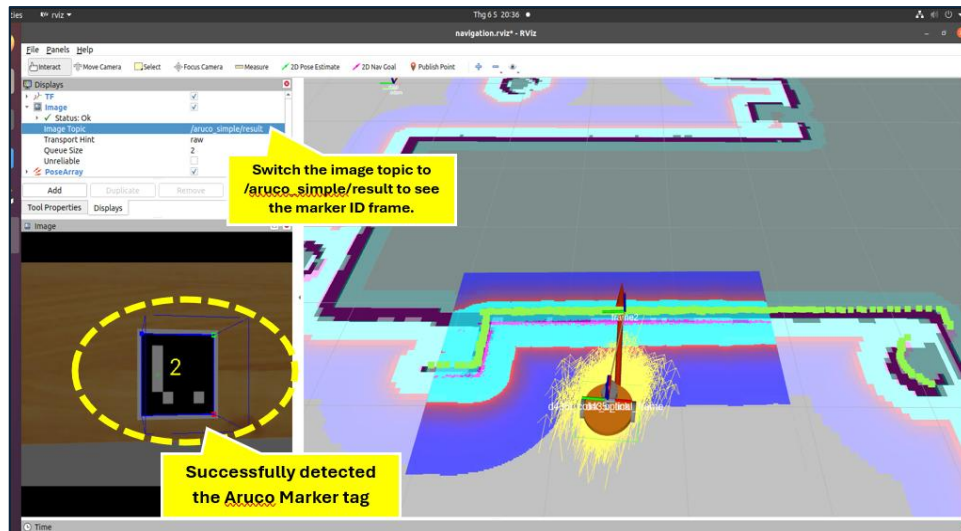


Figure 5. 13: The topic /image has been converted to /aruco_simple/result for observing the IDs

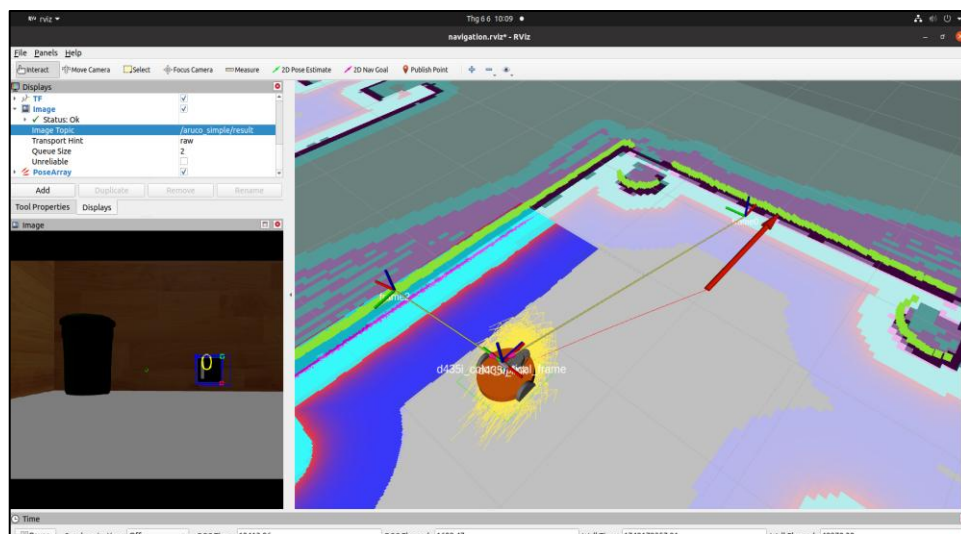


Figure 5.14: The robot has captured frame2 and frame0

5.5 Robot Relocalization Process

I will create scenarios using "2D Pose Estimate" and "2D Nav Goal" tools. These tools allow for dragging or directing the robot in both the Gazebo and RViz environments. To intentionally introduce a localization error for test the relocalization algorithm, as illustrated in *Figure 5.15* below, the robot was manually displaced from its actual position. In the figure:

- The blue ellipse on the right indicates the current exact position in the Gazebo environment.

- The blue ellipse on the left represents the correct position where the robot should appear in RViz.
- The red ellipse on the left shows the robot's estimated pose after being manually displaced in the RViz environment.



Figure 5.15: I used the Pose Estimate tool in RViz to manually displace the robot in Gazebo for testing the localization algorithm

I ran the node `goal_relocalize` with the command: `“rosrun aruco_ros goal_relocalize.py”` has successfully logged the line "Received pose from /pose2" and published it to /initialpose as shown in Figure 5.16.

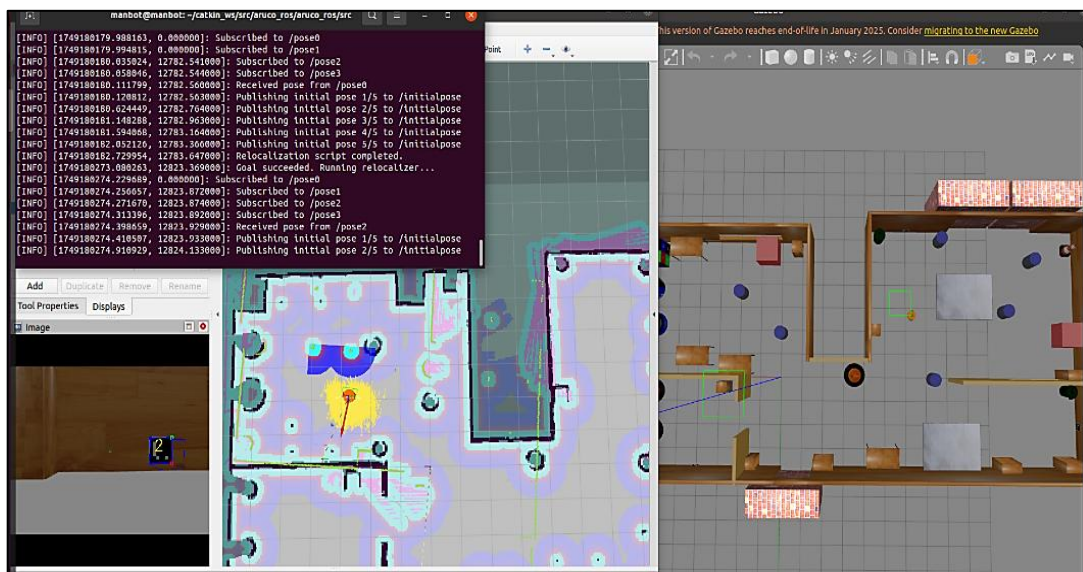


Figure 5.16: The node `goal_relocalize` has outputted the line "Received pose from /pose2" and published it to /initialpose

The robot navigated to the Aruco Marker location as demonstrated in *Figure 5.17*, triggering relocalization upon arrival.

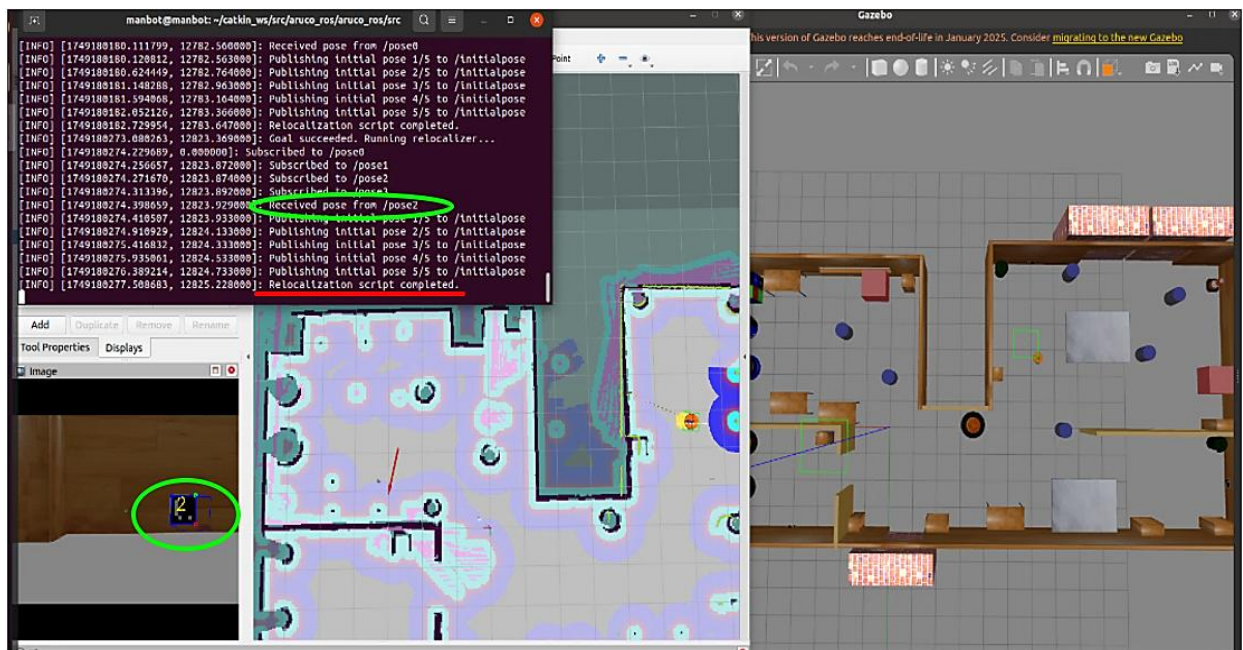


Figure 5.17: Relocalization successful at /pose2

The node `goal_relocalize` has published "Relocalization script completed". Relocalization was successful.

5.6 Experimental measurement and error analysis

Measurement procedure: After each instance where the robot detects a marker and performs relocalization, I record the coordinates (x, y, yaw) from `/initialpose` and then conduct 5 relocalization attempts, checking the discrepancies between the positions. Direct measurement means the camera is facing directly at the plane containing the Marker, with the camera's z-axis perpendicular to the Marker's plane. This is to validate that my Relocalization algorithm functions correctly under ideal conditions, without noise as in real-world scenarios. This serves as a basis for comparing angular deviations and thereby evaluating the system's reliability.

→ If the discrepancies are small (below 5-10cm and under 5 degrees), then the relocalization system operates stably.

At Aruco Marker with ID=0, the fixed position of the tag is: x= -6.0; y= -6.35; yaw= -1.5708

Table 5.1: Relocalization Result for Marker ID 0

Time	Distance (m)	Gazebo			Relocalization			Error		
		x (m)	y (m)	yaw (degree)	x (m)	y (m)	yaw (degree)	Δx (m)	Δy (m)	Δyaw (degree)
1	0.5	-6	-5.85	89.51	-6.02	-5.88	89.51	-0.02	-0.03	0.00
2	0.75	-6	-5.60	89.51	-6.00	-5.64	89.81	0.00	-0.04	0.30
3	1	-6	-5.38	89.51	-6.00	-5.43	89.79	0.00	-0.05	0.28
4	1.25	-6	-5.10	89.51	-6.00	-5.16	89.82	0.00	-0.06	0.31
5	1.5	-6	-4.85	89.51	-5.92	-4.91	91.22	0.08	-0.06	1.71

- Average error $\overline{\Delta x} = 0.02\text{m}$, $\overline{\Delta y} = 0.05\text{m}$ and angle $\overline{\Delta yaw} = 0.52^\circ$
- X-axis error: $-0.02\text{ m} \leq \Delta x \leq 0.08\text{ m}$; Y-axis error: $-0.06\text{ m} \leq \Delta y \leq -0.03\text{ m}$; and orientation error: $0^\circ \leq \Delta yaw \leq 1.71^\circ$.

At Aruco Marker with ID=1, the fixed position of the tag is: x= -14.15; y= 2.0; yaw= 3.1416

Table 5.2: Relocalization Result for Marker ID 1

Time	Distance (m)	Gazebo			Relocalization			Error		
		x (m)	y (m)	yaw (degree)	x (m)	y (m)	yaw (degree)	Δx (m)	Δy (m)	Δyaw (degree)
1	0.5	-13.65	2	179.88	-13.67	2.00	179.83	-0.02	0.00	-0.05
2	0.75	-13.40	2	179.88	-13.44	2.00	179.84	-0.04	0.00	-0.04
3	1	-13.15	2	179.88	-13.19	2.00	179.96	-0.04	0.00	0.08
4	1.25	-12.90	2	179.88	-12.96	2.00	179.94	-0.06	0.00	0.06
5	1.5	-12.65	2	179.88	-12.72	1.99	179.61	-0.07	-0.01	-0.27

- Average error $\overline{\Delta x} = 0.05\text{m}$, $\overline{\Delta y} = 0$ and angle $\overline{\Delta yaw} = 0.1^\circ$
- X-axis error: $-0.07\text{ m} \leq \Delta x \leq -0.02\text{ m}$; Y-axis error is extremely small, with 4 out of 5 measurements being 0.00 m and only one instance at -0.01 m; and orientation error: $-0.27^\circ \leq \Delta yaw \leq 0.08^\circ$.

At Aruco Marker with ID=2, the fixed position of the tag is: x= -4.21; y= -3.5; yaw= 0.0

Table 5.3: Relocalization Result for Marker ID 2

Time	Distance (m)	Gazebo			Relocalization			Error		
		x (m)	y (m)	yaw (degree)	x (m)	y (m)	yaw (degree)	Δx (m)	Δy (m)	Δyaw (degree)
1	0.5	-4.71	-3.5	0.44	-4.69	-3.49	0.44	0.02	0.01	0.00
2	0.75	-4.96	-3.5	0.44	-4.92	-3.50	0.44	0.04	0.00	0.00
3	1	-5.21	-3.5	0.44	-5.17	-3.50	0.42	0.04	0.00	-0.02
4	1.25	-5.46	-3.5	0.44	-5.40	-3.50	0.47	0.06	0.00	0.03
5	1.5	-5.71	-3.5	0.44	-5.64	-3.49	0.49	0.07	0.01	0.05

- Average error $\overline{\Delta x} = 0.05\text{m}$, $\overline{\Delta y} = 0$ and angle $\overline{\Delta yaw} = 0.02^\circ$

- X-axis error: $0.02 \text{ m} \leq \Delta x \leq 0.07 \text{ m}$; Y-axis error is extremely small, with 3 out of 5 measurements being 0.00 m and two instances at 0.01 m ; and orientation error: $-0.02^\circ \leq \Delta \text{yaw} \leq 0.05^\circ$.

At Aruco Marker with ID=3, the fixed position of the tag is: $x = 4.0$; $y = 3.7$; $\text{yaw} = 1.5708$

Table 5.4: Relocalization Result for Marker ID 3

Time	Distance (m)	Gazebo			Relocalization			Error		
		x (m)	y (m)	yaw (degree)	x (m)	y (m)	yaw (degree)	Δx (m)	Δy (m)	Δyaw (degree)
1	0.5	4	3.20	90.22	3.99	3.22	90.20	-0.01	0.02	-0.02
2	0.75	4	2.95	90.22	4.00	2.99	90.26	0.00	0.04	0.04
3	1	4	2.70	90.22	4.00	2.75	90.17	0.00	0.05	-0.05
4	1.25	4	2.45	90.22	4.00	2.51	90.17	0.00	0.06	-0.05
5	1.5	4	2.20	90.22	4.09	2.27	90.57	0.09	0.07	0.35

- Average error $\overline{\Delta x} = 0.02 \text{ m}$, $\overline{\Delta y} = 0.05$ and angle $\overline{\Delta \text{yaw}} = 0.1^\circ$
- X-axis error: $-0.01 \text{ m} \leq \Delta x \leq 0.09 \text{ m}$; Y-axis error: $0.02 \text{ m} \leq \Delta y \leq 0.07 \text{ m}$; and orientation error: $-0.05^\circ \leq \Delta \text{yaw} \leq 0.35^\circ$.

Table of average measurement values along the x-axis, y-axis, and angular deviation:

Table 5.5: Average Relocalization Error of the 4 Aruco Markers

ID Marker	$ \overline{\Delta x} $ (m)	$ \overline{\Delta y} $ (m)	$ \overline{\Delta \text{yaw}} $ (degree)
0	0.02	0.05	0.52
1	0.05	0	0.10
2	0.05	0	0.02
3	0.02	0.05	0.10
Average error of the 4 tags	± 0.03	± 0.03	± 0.19

Evaluation of Individual Aruco Markers:

- **Marker ID = 0:** Exhibits very low error along the X-axis (0.02 m) and an acceptable Y-axis error (0.05 m), but it has the highest orientation error ($\text{yaw} = 0.52^\circ$), with large fluctuations reaching a maximum error of 1.71° . This indicates instability in heading estimation, especially at greater distances.
- **Marker ID = 1:** Shows almost zero error in the Y-axis, low yaw error (0.10°), but a relatively higher X-axis error (0.05 m). Nevertheless, it remains generally stable at short ranges.

- **Marker ID = 2:** Delivers the most accurate and consistent results across all components, with zero Y-axis and yaw errors (0 m and 0.02° , respectively), and an X-axis error of 0.05 m. This marker is ideal for serving as a reference or origin point in the system.
- **Marker ID = 3:** Has very low X-axis error (0.02 m), moderate Y-axis error (0.05 m), and low yaw error (0.10°). It performs reliably across all components without any significant outliers.

Systematic Behavior:

There is a clear systematic trend in localization errors. For all Aruco markers (ID = 0, 1, 2, 3), the errors along the X-axis, Y-axis, and yaw angle tend to increase as the distance between the robot and the marker increases. This trend is particularly evident in the maximum absolute error values, which often occur at the farthest measurement point (Distance = 1.5 m). A possible explanation is the reduced image resolution of the marker at greater distances.

Stability and Repeatability:

- The analysis indicates that localization error does not increase linearly over time during continuous operation. Instead, the error tends to accumulate with increasing distance from the Aruco Marker. Notably, the largest deviations are typically observed when the robot is positioned at a distance of approximately 1.5 meters from the marker.
- The system exhibits strong stability and repeatability in both position and orientation estimation when operating at close range to the Aruco Markers. However, as the distance grows (e.g., 1.5 meters), both stability and repeatability degrade. This results in noticeably higher errors, particularly in the yaw (angular) component. The system achieves optimal performance when the robot remains in close proximity to reference markers.

5.7 Comparison with Expected Performance

The relocalization system was designed with specific expectations regarding accuracy and stability under ideal conditions. Specifically, it is expected that, when the robot's

camera is directly facing the Aruco Marker (i.e., the camera's z-axis is perpendicular to the marker plane), the system should produce positional errors below 5–10cm and angular errors (yaw) below 5° . This scenario serves as a controlled baseline for evaluating system performance without the influence of external noise.

Experimental results confirm that the system meets these expectations. Across all tested markers, the absolute positional errors along the X and Y axes remain consistently below 5 cm in most cases. The yaw error also stays well below the 5° threshold, with the highest deviation recorded at 1.71° for Marker ID = 0. These results validate the accuracy and repeatability of the relocalization algorithm under ideal measurement conditions. Consequently, the system demonstrates its capability to perform reliable and stable relocalization when environmental factors are minimized. This benchmark can serve as a solid foundation for future testing under more challenging, real-world conditions.

Improvement directions

The measurements presented above were conducted in a simulated environment, where operating conditions are ideal: the robot moves on a flat surface, is unaffected by vibrations, lighting is uniform, and there are no errors introduced by sensor hardware or image artifacts such as video noise, processing delays, or distortion due to camera angles. As a result, localization errors in the simulation are significantly lower compared to those in real-world conditions.

To enhance localization performance and reduce error in more realistic scenarios, the following improvements are proposed:

- Increase the density of Aruco Marker tags, utilize larger-sized markers (e.g., 15 cm), and ensure that the average distance between the robot and nearby markers remains below 1.5 meters whenever feasible.
- Redesign the spatial layout of markers to eliminate dead zones, i.e., areas where no detectable markers are available, which can negatively impact the robot's ability to localize itself.

Chapter 6: EXPERIMENTAL RESULTS AND EVALUATION

6.1 Mapping and saving in real environment

The testing area has dimensions of 10×5 meters (L×W), with a total area: 50m².



Figure 6.1: Marker tags are attached in the robot testing area

I started scanning the map with Gmapping algorithm: “`roslaunch gmapping slam_gmapping pr2.py`”.

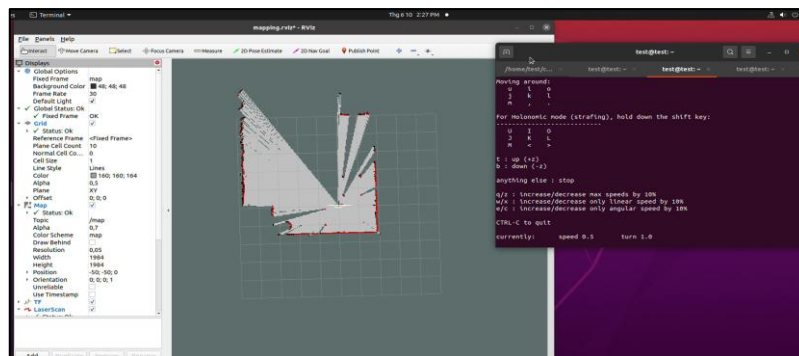


Figure 6.2: RViz was launched to visualize and monitor the mapping process

I am manually controlling my robot to perform a full scan of the environment.

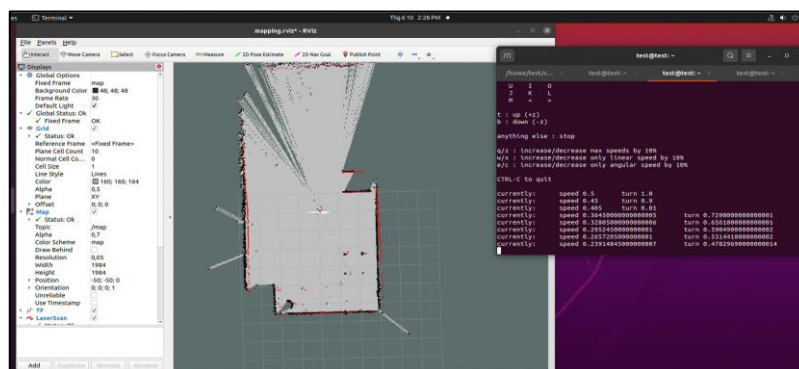


Figure 6.3: The robot is being directed to explore specific features within the map

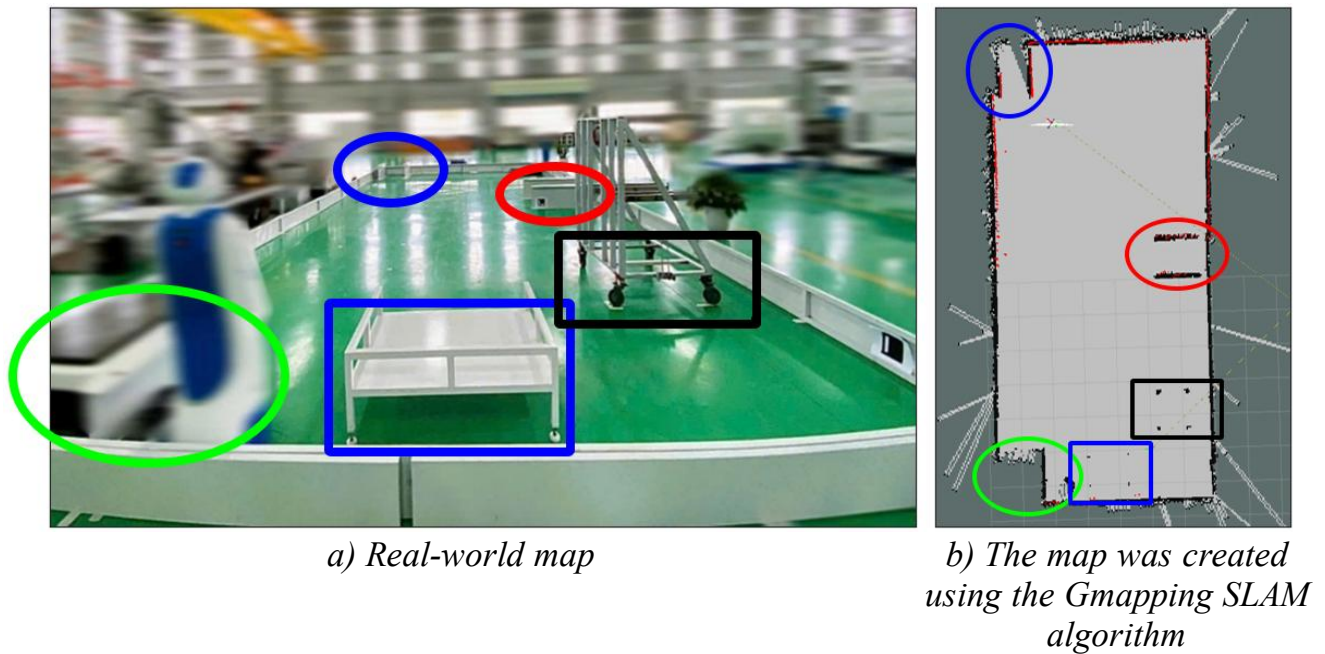
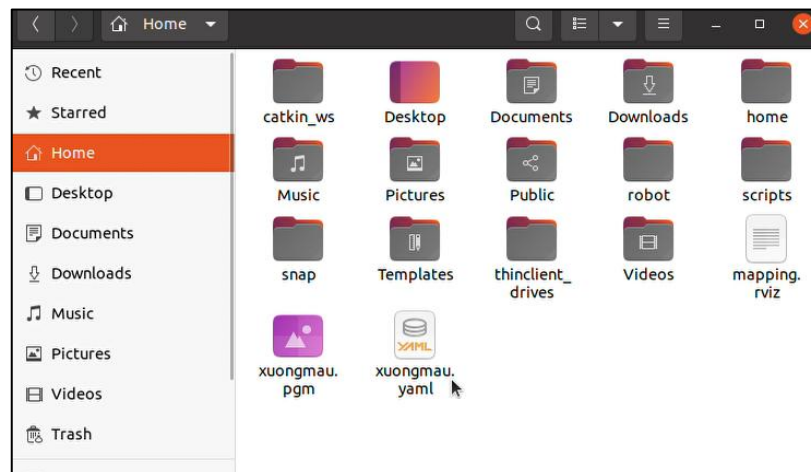


Figure 6.4: The figure shows matching features between the two maps

I saved the map with the name “xuongmau”. I will have two types of map files generated: a map description file (.yaml) and a map image file (.pgm)



6.3 Determining and recording the position of the Aruco marker tag

Unlike in the Gazebo simulation where marker positions are predefined, in the real-world environment the Aruco Marker tags were manually attached and their precise locations were unknown. Thus, it was necessary to first detect and record the positions of the markers in order to enable reliable identification during localization.

First, I entered the command “`roslaunch tf_echo map base_link`”. This command is used to measure the distance between the robot's base and the map frame. This method allows

me to extract the `yaw` orientation of the Aruco Marker tag with respect to the map frame.

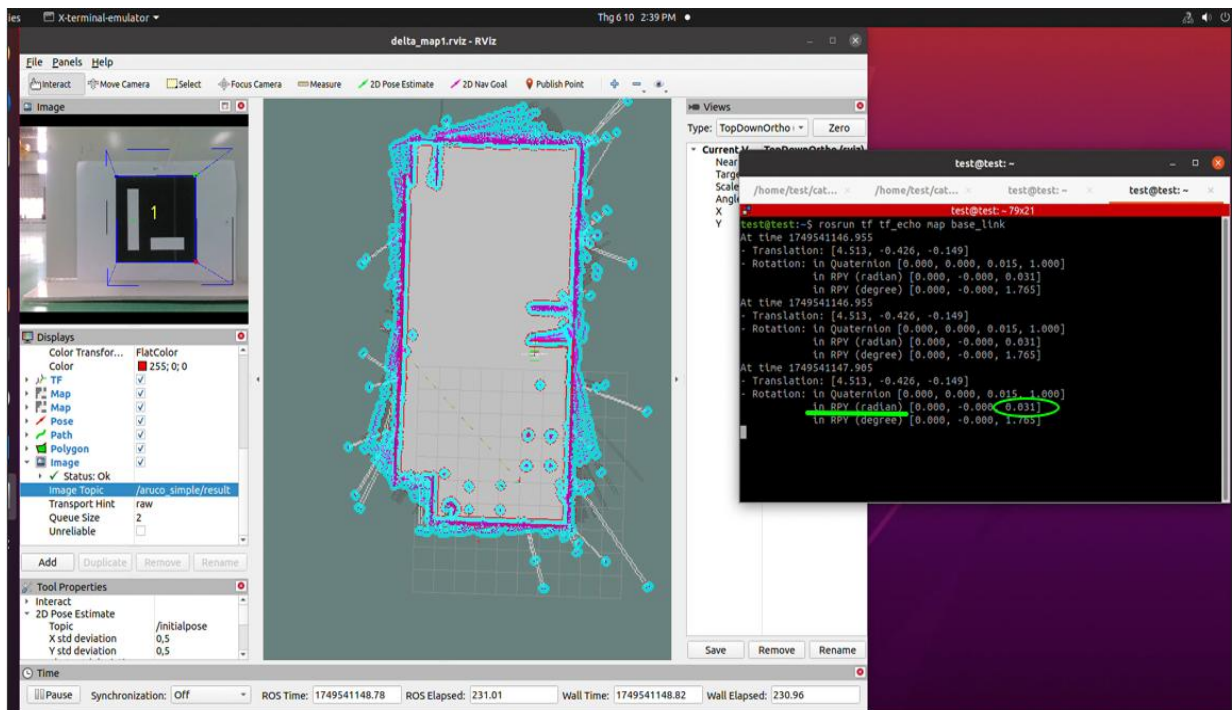


Figure 6.5: The yaw orientation is measured to be 0.031 radians

Next, to determine the x and y coordinates of the tag in the map frame, I entered the command: “`rostopic echo /clicked_point`”

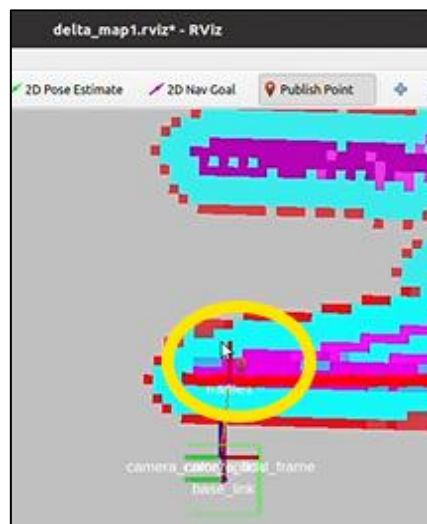


Figure 6.6: The Publish Point tool was used to click precisely at the origin of the frame1 coordinate frame

After that, I utilized the *Publish Point* tool to select the exact position at the `frame1` coordinate frame in RViz. The corresponding x and y coordinates of Marker_1 were then displayed in the terminal.

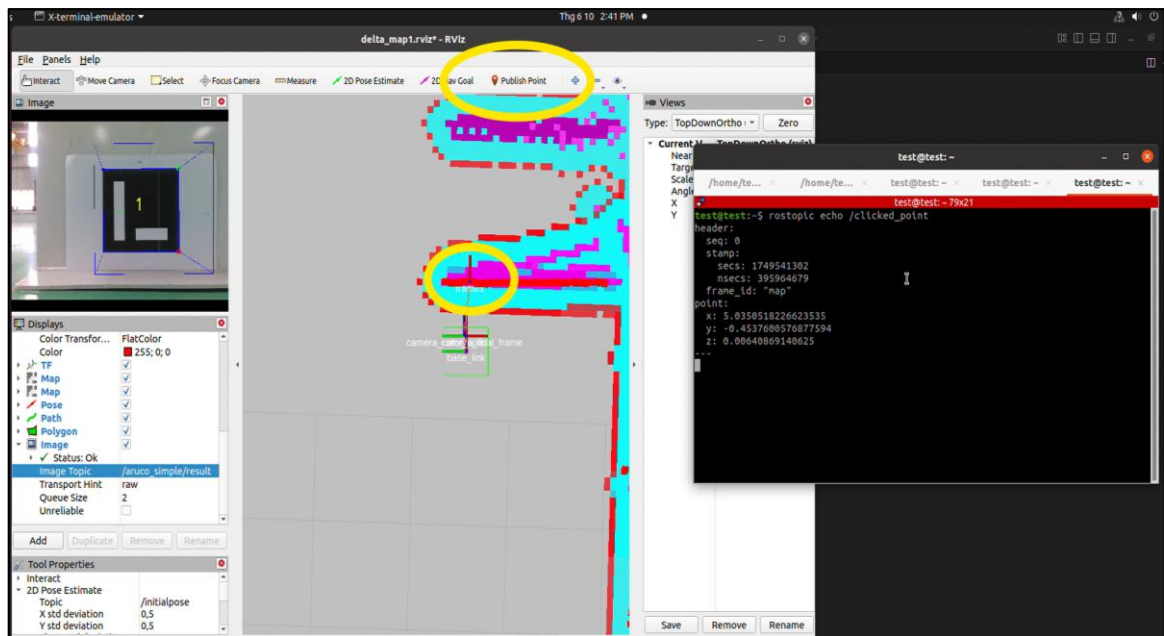


Figure 6.7: The x value is 5.035 meters and the y value is -0.453 meters

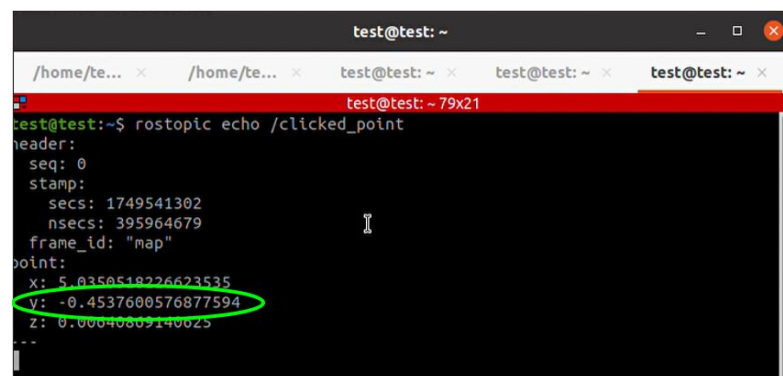
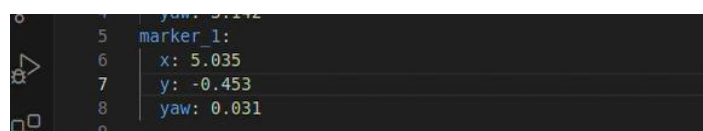
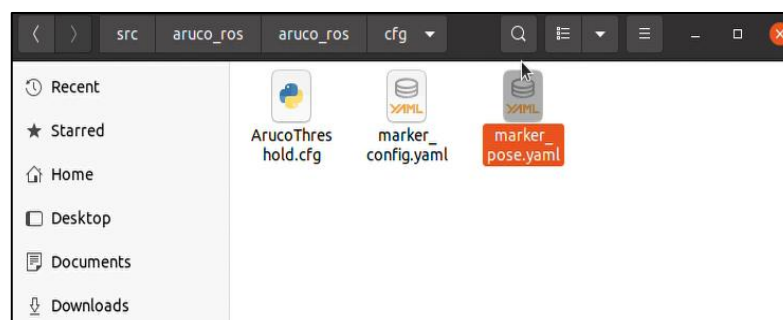


Figure 6.8: Enlarged image of the terminal

After obtaining the x, y, and yaw values, I saved them into my *marker_pose.yaml* file to serve as the fixed position of Marker_1.



A similar process was performed for Marker_0.

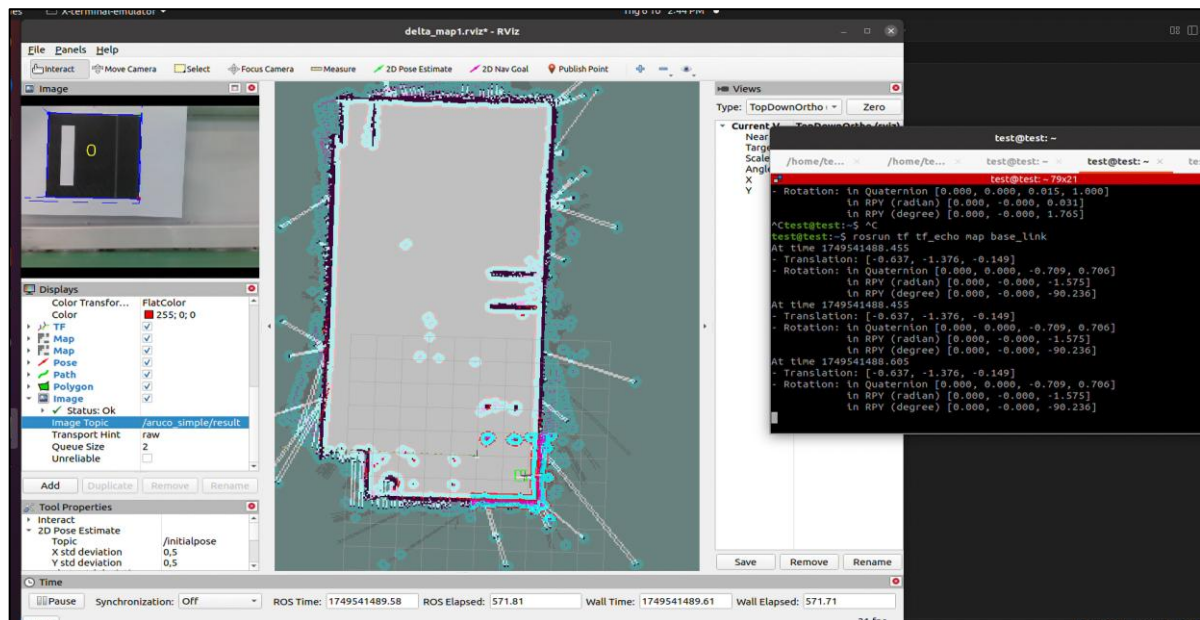


Figure 6.9: Command “`rostopic echo /clicked_point`” to get yaw orientation: -1.575

Then, I re-entered the command “`rostopic echo /clicked_point`” to obtain a more accurate position of the marker tag.

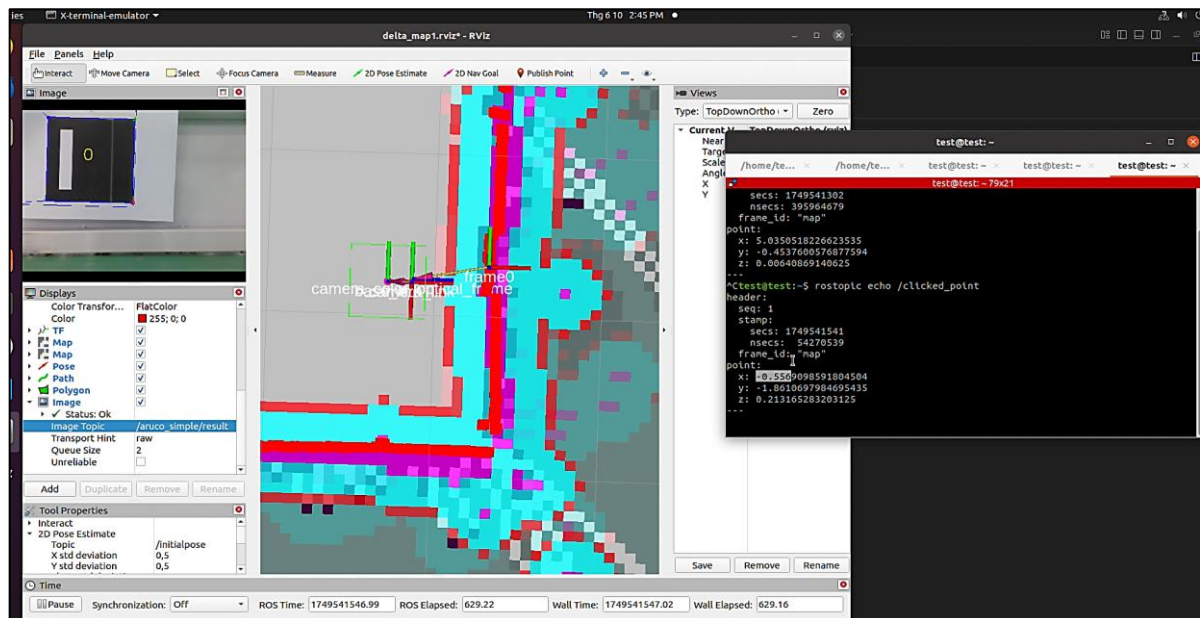
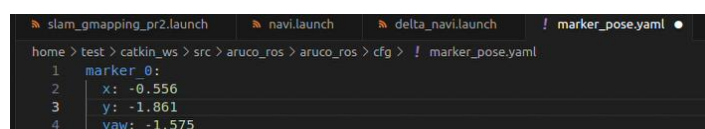


Figure 6.10: We have coordinate value with $x = -0.556$, $y = -1.861$

I obtained the values of Marker_0 as $x = -0.556$, $y = -1.861$ and yaw = -1.575



6.4 Detect Aruco markers to localize the robot within the map

I ran the command: “`roslaunch aruco_ros multiple.launch`” to detect and output the pose of Aruco Markers on the screen. The node "multiple" analyzes images from the camera, recognizing Aruco Markers with IDs subscribed in the configuration. For each detected Marker, the node publishes a corresponding pose to topics like `/pose0`, `/pose1`.

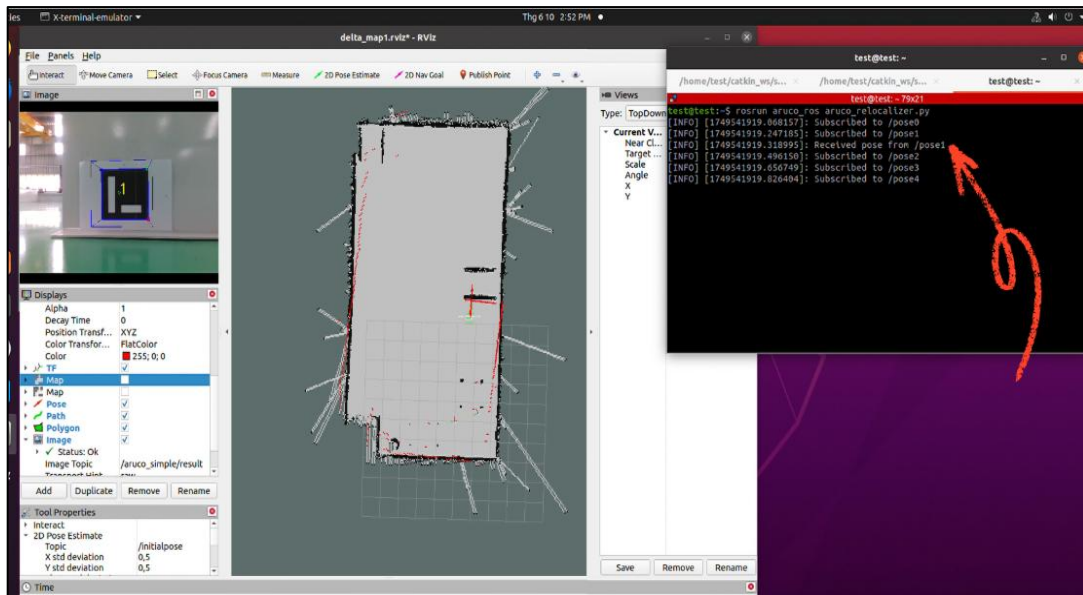


Figure 6.11: The `/pose1` topic was successfully received

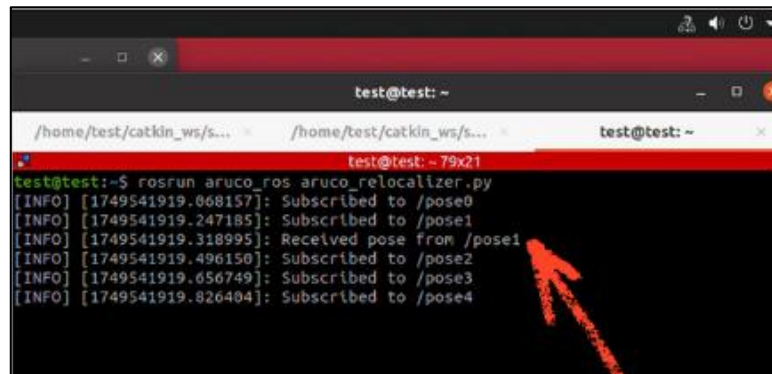


Figure 6.12: The terminal display was scaled up

6.5 Robot Relocalization Process

The robot performs relocalization using visual markers when it loses track of its position. I entered the command “`roslaunch aruco_ros goal_relocalize.py`” to make the robot perform relocalization using visual markers.

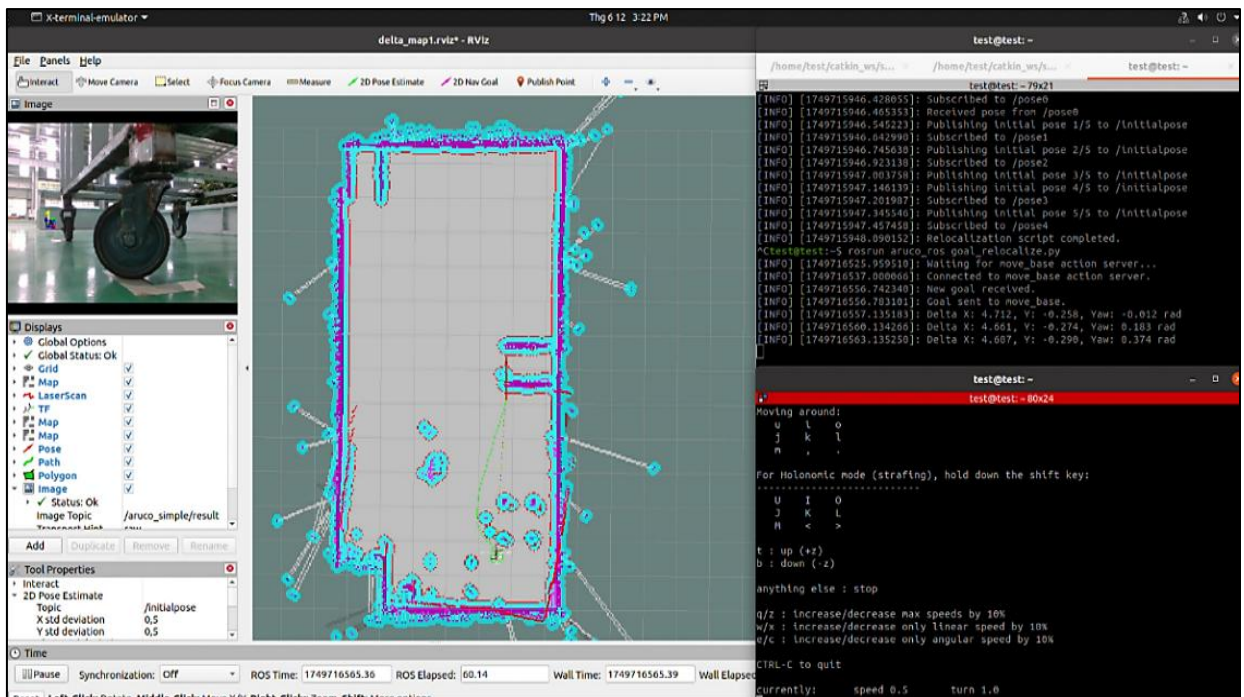


Figure 6.13: The robot started navigating

Upon completion of the navigation task, the system immediately proceeds to the next phase: robot relocalization as shown in Figure 6.14. Then, log message relocalization completed after the process had finished.

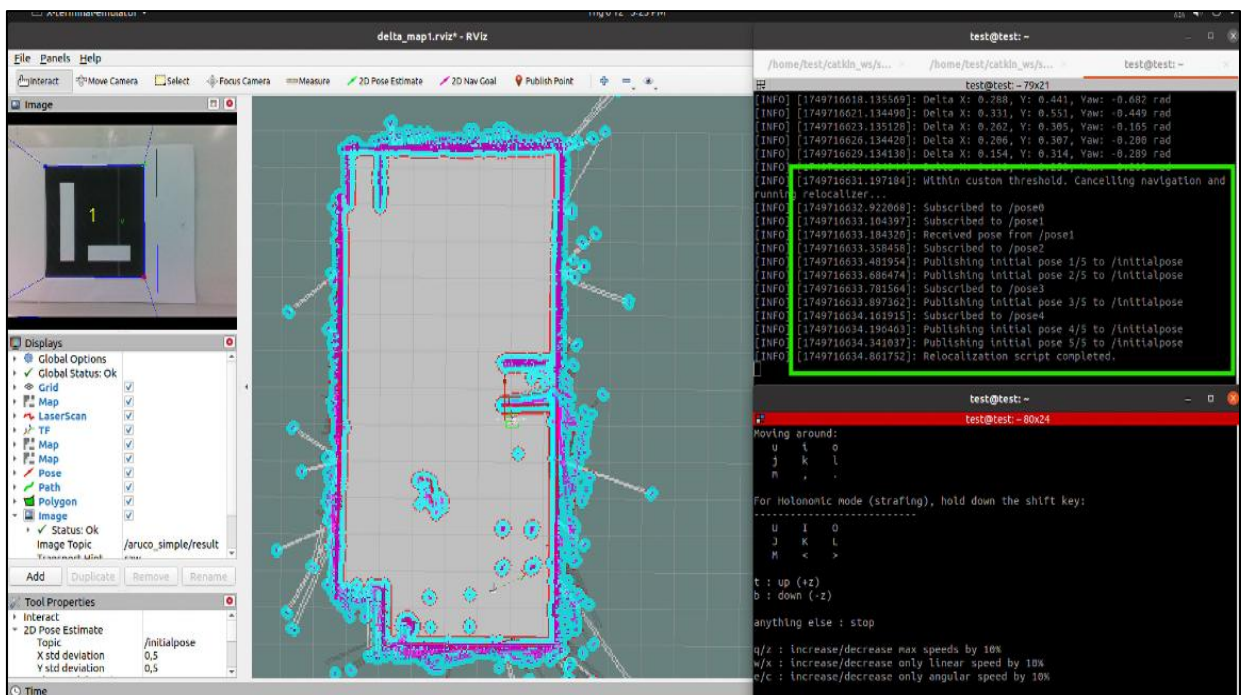


Figure 6.14: Relocalization of the robot was completed successfully

6.6 Practical Evaluation of Robot Relocalization

At Aruco Marker with ID=0, the fixed position of the tag is: $x = -0.556$; $y = -1.861$ and $yaw = -1.575$

Table 6.1: Relocalization Result for Marker ID 0 at the Testing Center

Time	Gazebo			Relocalization			Error		
	x (m)	y (m)	yaw (degree)	x (m)	y (m)	yaw (degree)	Δx (m)	Δy (m)	Δyaw (degree)
1	-0.58	-1.27	-86.32	-0.62	-1.28	-85.8	-0.04	-0.01	0.52
2	-0.62	-1.28	-82.56	-0.66	-1.28	-81.3	-0.04	0.00	1.26
3	-0.81	-1.08	-77.15	-0.83	-1.06	-77.9	-0.02	0.02	-0.75
4	-0.19	-0.86	-111	-0.15	-0.87	-109.81	0.04	-0.01	1.19
5	-0.1	-0.52	-104.7	-0.01	-0.58	-103.6	0.09	-0.06	1.10
6	-0.51	-0.58	-90.3	-0.31	-0.61	-89.4	0.20	-0.03	0.90

- Average error $|\Delta x| = 0.07\text{m}$, $|\Delta y| = 0.02\text{m}$ and angle $|\Delta yaw| = 0.95^\circ$

At Aruco Marker with ID=1, the fixed position of the tag is: $x = 5.035$; $y = -0.453$ and $yaw = 0.031$

Table 6.2: Relocalization Result for Marker ID 0 at the Testing Center

Time	Gazebo			Relocalization			Error		
	x (m)	y (m)	yaw (degree)	x (m)	y (m)	yaw (degree)	Δx (m)	Δy (m)	Δyaw (degree)
1	4.54	-0.42	1.6	4.56	-0.45	2.29	0.02	-0.03	0.69
2	4.31	-0.57	8.27	4.31	-0.61	9.52	0.00	-0.04	1.25
3	4.31	-0.81	22.57	4.30	-0.81	23.77	-0.01	0.00	1.20
4	3.73	-0.47	0.09	3.73	-0.42	-1.61	0.00	0.05	-1.70
5	4.07	-0.03	-27.36	4.11	-0.01	-27.66	0.04	0.02	-0.30
6	4.40	-0.08	-33.80	4.42	-0.06	-33.61	0.02	0.02	0.19

- Average error $|\Delta x| = 0.01\text{m}$, $|\Delta y| = 0.03\text{m}$ and angle $|\Delta yaw| = 0.89^\circ$

Table of average measurement values along the x-axis, y-axis, and angular deviation:

Table 6.3: Average Relocalization Error at two positions

ID Marker	$ \Delta x $ (m)	$ \Delta y $ (m)	$ \Delta yaw $ (degree)
0	0.07	0.02	0.95
1	0.01	0.03	0.89
Average error of the 2 tags	± 0.04	± 0.02	± 0.92

Evaluation of actual results:

After the actual testing process on the robot, the repositioning algorithm using Aruco Marker showed the effectiveness within the expected threshold that the company set

with the target for me is that the error is within $x, y \pm 0.1$ (m), deviation angle ± 5 (degrees), with the following noteworthy points:

- **Marker ID = 0:**

- + Fixed position: $x = -0.556$, $y = -1.861$, $\text{yaw} = -1.575$.
- + The results show that the position error along the X and Y axes is stable. However, the yaw deviation is quite large due to the limited viewing angle of the camera and unfavorable marker observation conditions (narrow, hidden), affecting the accuracy of positioning.

- **Marker ID = 1:**

- + Fixed position: $x = 5.035$, $y = -0.453$, $\text{yaw} = 0.031$.
- + The repositioning results are very stable, thanks to the camera's scanning angle located directly with the marker, which increases accuracy and reduces angle errors.

- **Summing up the actual average error:**

- + Average absolute error on two markers:
 - ± 0.04 m along the X-axis,
 - ± 0.02 m along the Y axis,
 - $\pm 0.92^\circ$ for the angle of rotation (yaw).
- + These results are perfectly satisfactory for the robotic navigation applications in this Capstone Project. However, the angular error is still quite large, it should be noted when arranging the marker to ensure that the camera has a clear viewing angle and sufficient scanning surface area.

- **General conclusion:** The repositioning algorithm using Aruco Marker works effectively in real-world conditions, especially in marker locations with favorable viewing angles. However, it is necessary to further improve the ability to detect

markers in changing lighting conditions and distances, as well as optimize camera calibration to reduce angular errors.

Comparison between simulation and reality:

- Accuracy:

- + Simulation Environment: The algorithm is very accurate, with an error of less than ± 0.05 m and $\pm 0.19^\circ$, indicating ideal repositioning when not affected by sensor noise or light conditions.
- + Realistic environment: Errors increase, especially angular error (yaw) with an average of up to $\pm 0.92^\circ$, due to the influence of many practical factors such as lighting, noise from sensors, and camera viewing angles.

- Stability:

- + In the simulation: Markers such as ID=2 show very high stability, with almost negligible errors between measurements.
- + In fact: Despite slight fluctuations between measurements, the algorithm maintains quite good stability, especially at ID=1 where the camera has a favorable viewing angle.
- + Recommendation: Special attention should be paid to the location and fixation of the markers to ensure the repositioning effect in actual implementation.

- Environmental Effects:

- + The stark difference between the simulated and actual environment shows that the algorithm is sensitive to physical factors such as light, distance, reflectivity, and sensor error, which especially affects the yaw angle error.

Summarize the advantages and limitations of the algorithm:

- Advantage:

- + The repositioning algorithm using Aruco Marker gives good results in both simulated and real environments, with small errors and flexible applicability.
 - + **Marker ID=2** (in simulation) and **ID=1** (in practice) are ideal markers with low and stable errors, suitable as primary reference points in navigation systems.
 - + This is a solid foundation for the development of repositioning systems for mobile robots using LiDAR and cameras, especially in AMR systems.
- **Restrict:**
- + The yaw angle error in practice is significantly higher than that of the simulation, mainly due to the difficulty of detecting markers at inclined angles or narrow viewing angles.
 - + The algorithm needs to be further optimized to better adapt to the actual environment, possibly through solutions such as:
 - Added a noise reduction filter for camera data.
 - Use a high-resolution camera.
 - Combine with information from other sensors (IMU, LiDAR) to improve angular positioning.
- **Development direction:**
- + Combine information from LiDAR, cameras, encoders, and IMUs to build a powerful positioning system that does not depend on a single data source.
 - + Application of algorithms such as Extended Kalman Filter (EKF) to perform multi-sensor positioning with high reliability.
 - + Develop a marker layout diagram that covers the entire activity area, ensuring that each location is within the line of view of at least one marker.

CONCLUSION

General conclusion:

The project has completed the construction and deployment of a navigation robot system using Aruco Marker, integrating multi-sensor navigation and the ROS development environment. The system works stably in both simulated and actual industrial environments.

Affirming the results achieved:

- The robot can determine its position and orientation in real time based on Aruco Marker recognition at distances under 1.5 meters.
- The relocalization system operates effectively, with a positional error of less than 10 cm and an angular deviation below 5 degrees which the company's requirements.
- Navigation, obstacle avoidance and mapping functions have been successfully validated in both simulated and real-world environments.
- The robot's stability has been calibrated and verified during operation, ensuring the required localization accuracy.

Contributions of the topic:

- Effectively applied existing technologies and equipment for visual marker detection, contributing to reduced research and development costs for the company.
- Proposed and implemented a high-accuracy marker-based relocalization solution suitable for industrial environments.
- Developed a multi-sensor integration model on the ROS platform, which is flexible and scalable for various robotic applications.
- Completed a detailed report on experimental results and feedback, supporting technology transfer and product development within the company.

Challenges:

- The accuracy of relocalization using Aruco Markers decreases under reflective lighting conditions and at longer distances, causing slower path finding responses due to signal interference.
- The wheel mechanism had to be adjusted multiple times, as the uneven testing surface at the company led to instability, which prolonged the experimentation process.
- The system is not optimal for fast-changing environments or many obstacles that move continuously.

Proposals and recommendations:

- Expand research to address multi-robot coordination in industrial environments.
- Utilize Aruco Markers as task checkpoints in the factory, such as reference points for returning the robot to the charging station.

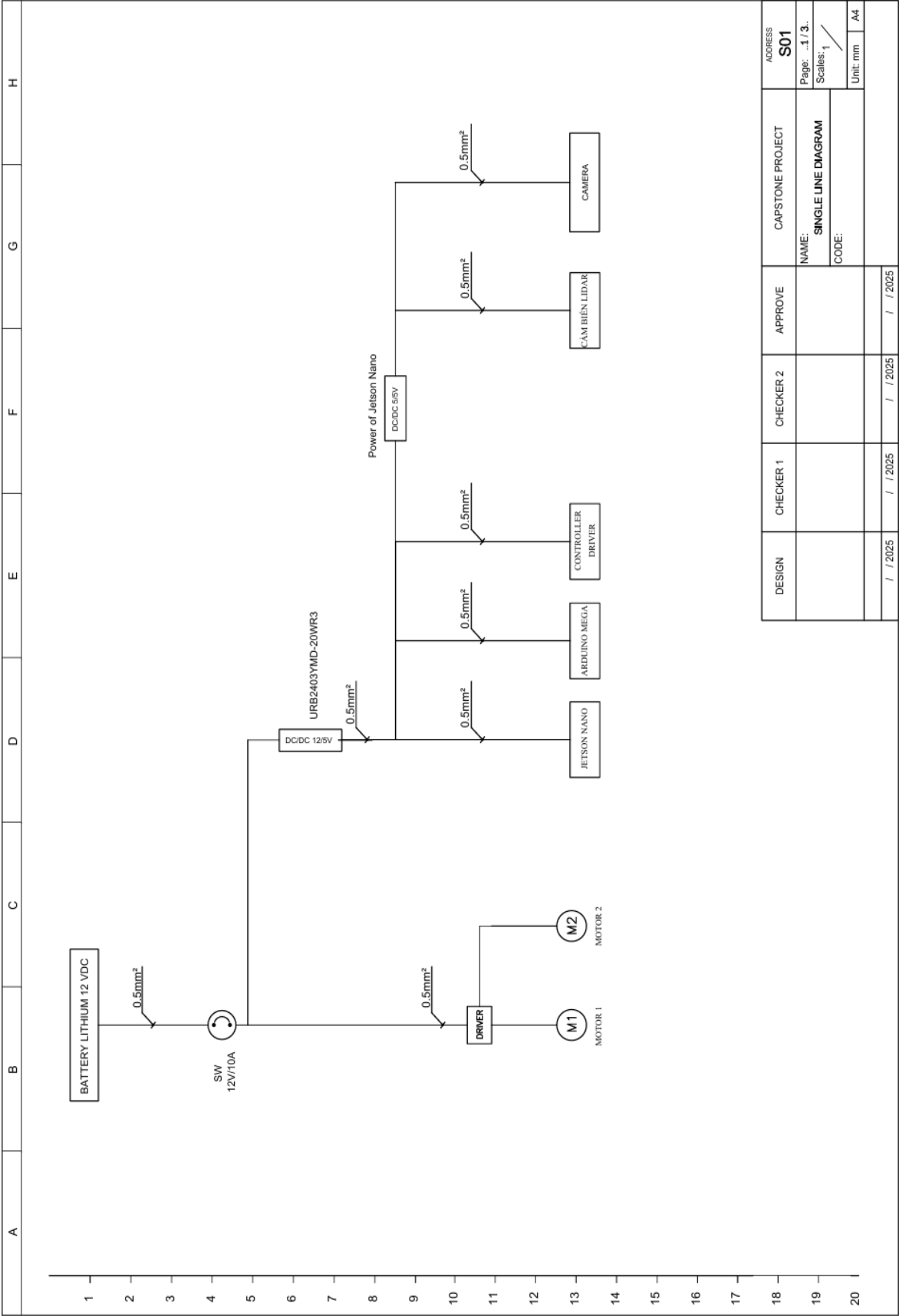
REFERENCES

- [1] Nathan E. Sanders, Elif Şener, Karen B. Chen, "Robot-related injuries in the workplace: An analysis of OSHA Severe Injury Reports," *Journals & Books Science Direct*, p. 1, 2024.
- [2] Lentin Joseph, Jonathan Cacace, Mastering ROS for Robotics Third Edition, UK: Packt Publishing Ltd, 2021.
- [3] R. A. Torrico, "Circuit Cellar Magazine Inspiring the Evolution of Embedded Design," *Circuit Cellar Magazine*, 01 03 2021. [Online]. Available: <https://circuitcellar.com/research-design-hub/design-solutions/intro-to-robot-operating-system-ros-part-1/>.
- [4] MathWorks, "Why SLAM Matters," [Online]. Available: <https://www.mathworks.com/discovery/slam.html>.
- [5] Caterina Rizzi, Francesca Campana, Michele Bici, Francesco Gherardini, Tommaso Ingrassia, Paolo Cicconi, Design Tools and Methods in Industrial Engineering II, Rome, Italy, 2021.
- [6] Aicha Manar ABBAD, Ines HAOUALA, Amanzhol RAISOV, Roumaissa BENKREDDA, "Low cost mobile navigation using 2D-SLAM in complex environments," *ResearchGate*, July 2023.
- [7] kscottz, Wyverald, "cartographer-project," github, 2023. [Online]. Available: <https://github.com/cartographer-project/cartographer/actions>.
- [8] Qiujie Li, Hongyi Zhu, "Performance evaluation of 2D LiDAR SLAM algorithms in simulated orchard environments," *ELSEVIER*, June 2024.
- [9] Recep Behlül Şahin, Muharrem Mercimek, "Fiducial Markers Aided Position Estimation for Vertical Landing," *The European Journal of Research and Development*, p. 31, 2023.

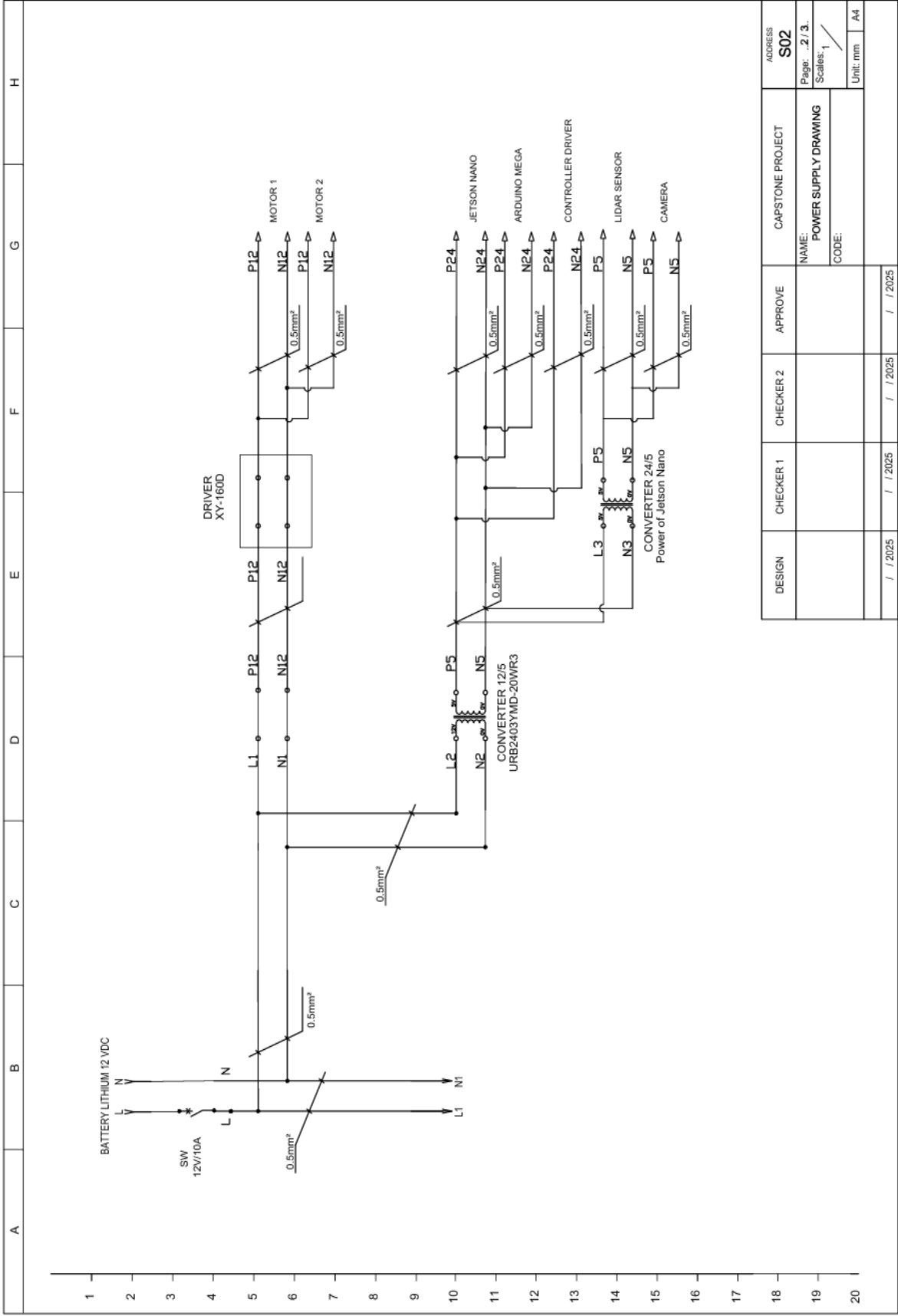
[10] P. Đ. Phước, Robot Công nghiệp, Hà Nội: NXB. Xây dựng, 2007.

[11] Phạm Trường Tùng, Đào Minh Đức, "Cơ sở thiết kế Robot Công nghiệp," NXB. Đại học Huế, pp. 16, 17, 18.

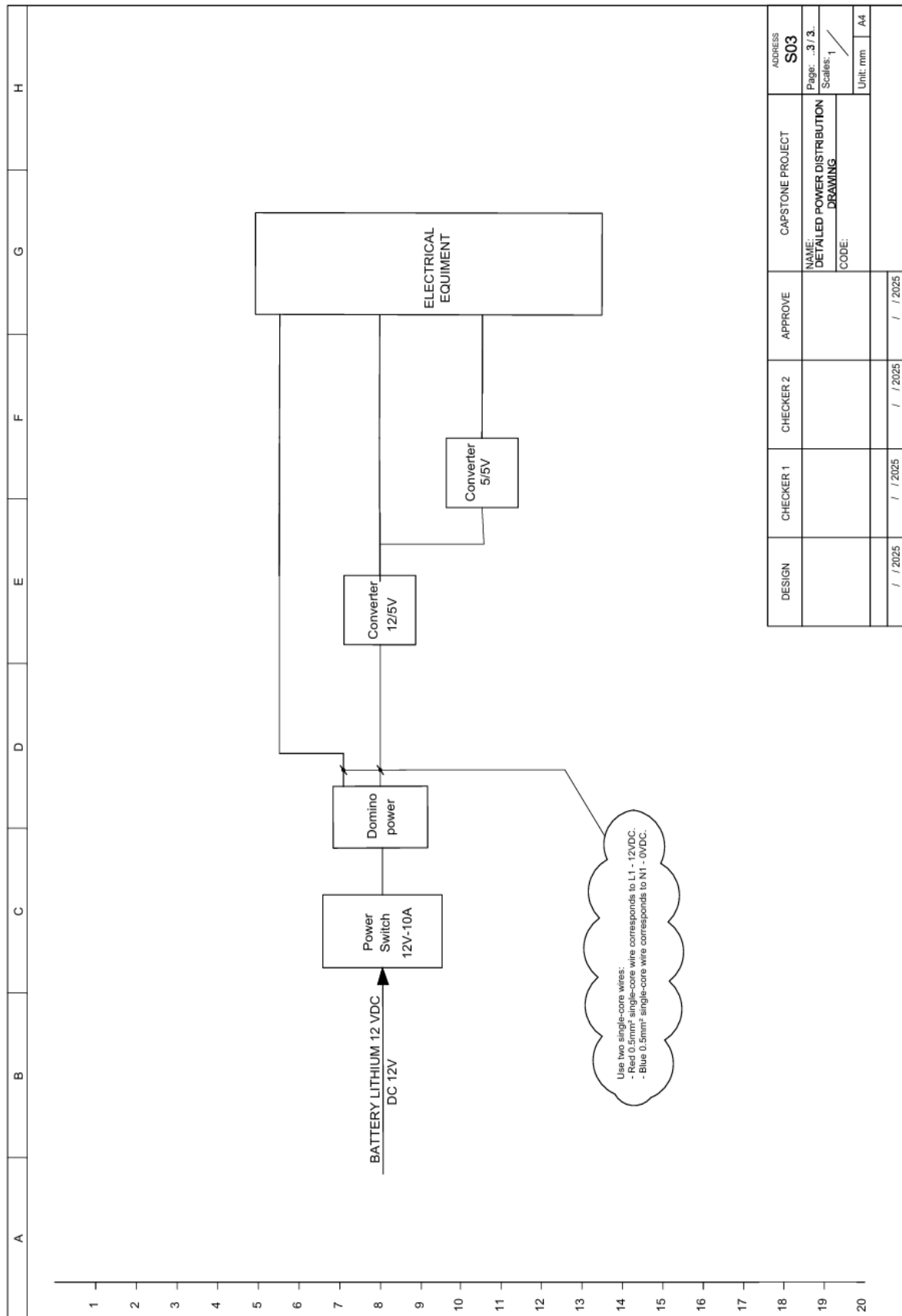
APPENDIX 1: SYSTEM DIAGRAM – SINGLE LINE DIAGRAM



APPENDIX 2: SYSTEM DIAGRAM – POWER SUPPLY DRAWING



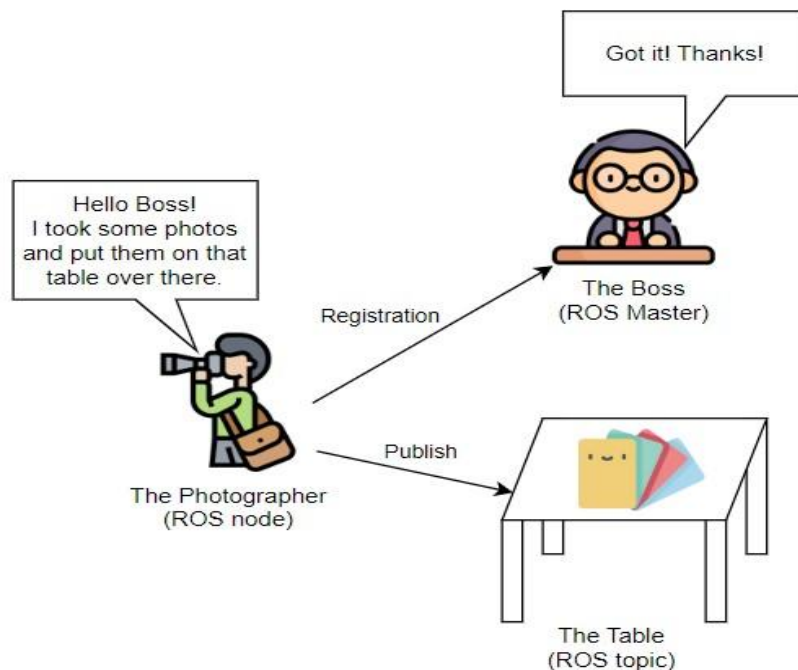
APPENDIX 3: SYSTEM DIAGRAM – DETAILED POWER DISTRIBUTION DRAWING



APPENDIX 4: ROS

4.1 ROS Master and Nodes

To easily understand Node, let's imagine our camera node as a Photographer who starts working in an office of a magazine. On the first day of his job, he takes some photos and notifies his Boss that he will leave these photos on a specific Table in the office. The Boss thanks him for the notification.



In our story, the Boss represents the ROS Master, which you always have to have when you use ROS. "The **ROS Master** provides **naming** and **registration** services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services" ([source](#)). ROS Master is called by using the command `roscore`. There should be only one ROS Master running at a time. If you try to call another one, there will be an error. The Photographer represents a **ROS Node** which must always register to Master when it starts similar to the Photographer notifying his Boss.

```

trinhnc@ubuntu:~$ roscore
... logging to /home/trinhnc/.ros/log/dab4ca34-73af-11ed-bc39-97f392b47a47/rosl
aunch-ubuntu-19581.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:35619/
ros_comm version 1.15.15

SUMMARY
=====
PARAMETERS
* /roscdistro: noetic
* /rosversion: 1.15.15
NODES
auto-starting new master
process[master]: started with pid [19590]
ROS_MASTER_URI=http://ubuntu:11311/

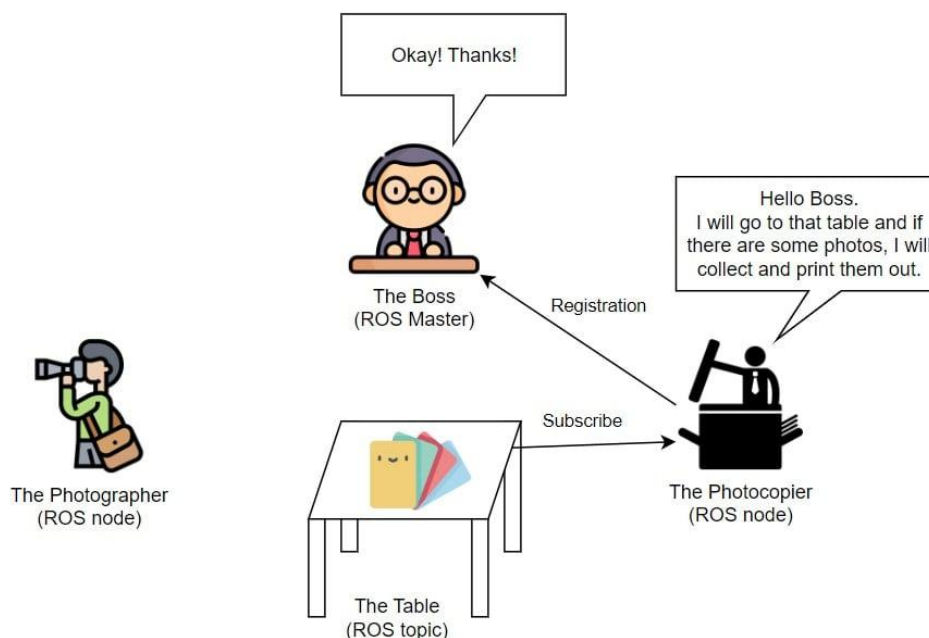
setting /run_id to dab4ca34-73af-11ed-bc39-97f392b47a47
process[rosout-1]: started with pid [19601]
started core service [/rosout]

```

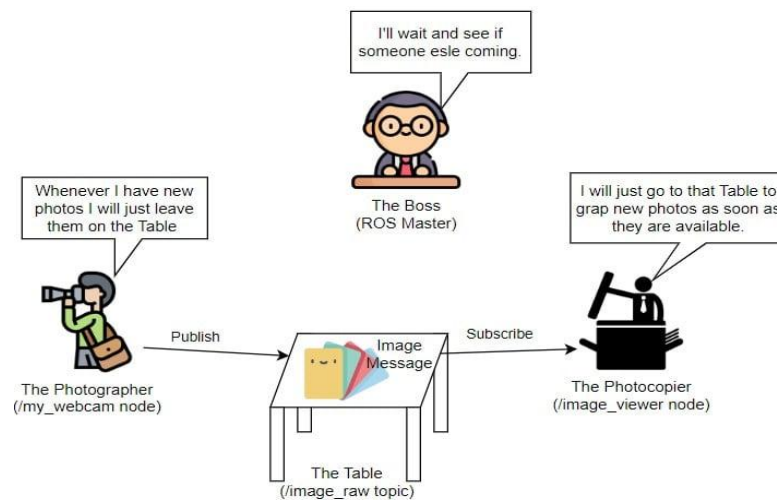
4.2 ROS Topic and Message

The photos themselves are Messages and they belong to the type sensor_msgs. That's why you had to include *sensor_msgs* when creating the package from the last chapter. The action of leaving photos on the Table is called **Publish** in ROS. Finally, the Table itself serves as a ROS Topic to which other nodes can *subscribe*.

What is *subscribe*? Now imagine there is another person (i.e. another Node) in our story called the Photocopier. This guy wants to take photos from the Table and print them out. What is he going to do? He of course needs to notify the Boss first and then if there are photos on the Table, he can collect and print them. The action of taking photos from the Table is called **Subscribe** in ROS.

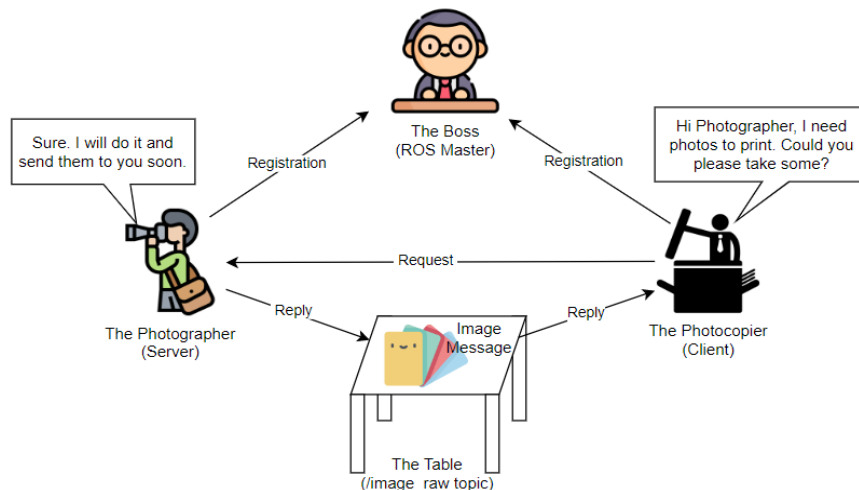


After notifying the Boss, both the Photographer and the Photocopier know about each other and they now just give/take photos to/from each other respectively.



4.3 ROS Service

In the previous section, the Photographer continuously publishes images while the Photocopier continuously subscribes to these images. The Photocopier now said only when he asked for an image, the Photographer should take one and send it to him. This request/reply interaction is basically the idea behind ROS Service. In our example, after both registrations (i.e. notifying the Boss) are done, the Photocopier acts as a Client sending a request to the Photographer who acts as a Server. After receiving the request, the Photographer grasps some images and sends them back to the Photocopier (actually he just leaves them on the Table as usual).



APPENDIX 5: MOMENT OF INERTIA

Declare the macros used to calculate the moment of inertia. The robot consists of three main components: a rectangular box-shaped body, cylindrical wheels, and spherical auxiliary wheels. We have the following corresponding formula for calculating torque:

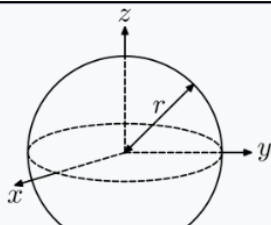
- Declare the dimensional parameters of robot components

```

3  <!-- base property -->
4  <xacro:property name="base_height" value="0.1" />
5  <xacro:property name="base_radius" value="0.2" />
6  <xacro:property name="base_mass" value="5" />
7  <!-- wheel property -->
8  <xacro:property name="wheel_height" value="0.03" />
9  <xacro:property name="wheel_radius" value="0.08" />
10 <xacro:property name="wheel_mass" value="0.8" />
11 <!-- caster property -->
12 <xacro:property name="caster_radius" value="0.05" />
13 <xacro:property name="caster_mass" value="0.5" />
14 <!-- laser property -->
15 <xacro:property name="laser_length" value="0.06" />
16 <xacro:property name="laser_width" value="0.06" />
17 <xacro:property name="laser_height" value="0.065" />
18 <xacro:property name="laser_mass" value="0.5" />
19

```

- 2 Casters:

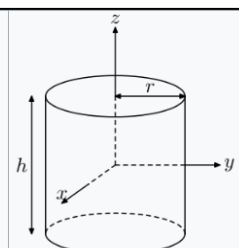
Hollow sphere of radius r and mass m		$I = \begin{bmatrix} \frac{2}{3}mr^2 & 0 & 0 \\ 0 & \frac{2}{3}mr^2 & 0 \\ 0 & 0 & \frac{2}{3}mr^2 \end{bmatrix}$
--	---	---

```

38 <xacro:macro name="sphere_inertia" params="mass r">
39   <inertial>
40     <mass value="{mass}" />
41     <inertia ixx="{(2/5)*mass*r*r}" ixy="0.0" ixz="0.0"
42             iyy="{(2/5)*mass*r*r}" iyz="0.0"
43             izz="{(2/5)*mass*r*r}" />
44   </inertial>
45 </xacro:macro>

```

- 2 Main wheels and body:

Solid cylinder of radius r , height h and mass m		$I = \begin{bmatrix} \frac{1}{12}m(3r^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{bmatrix}$
--	---	---

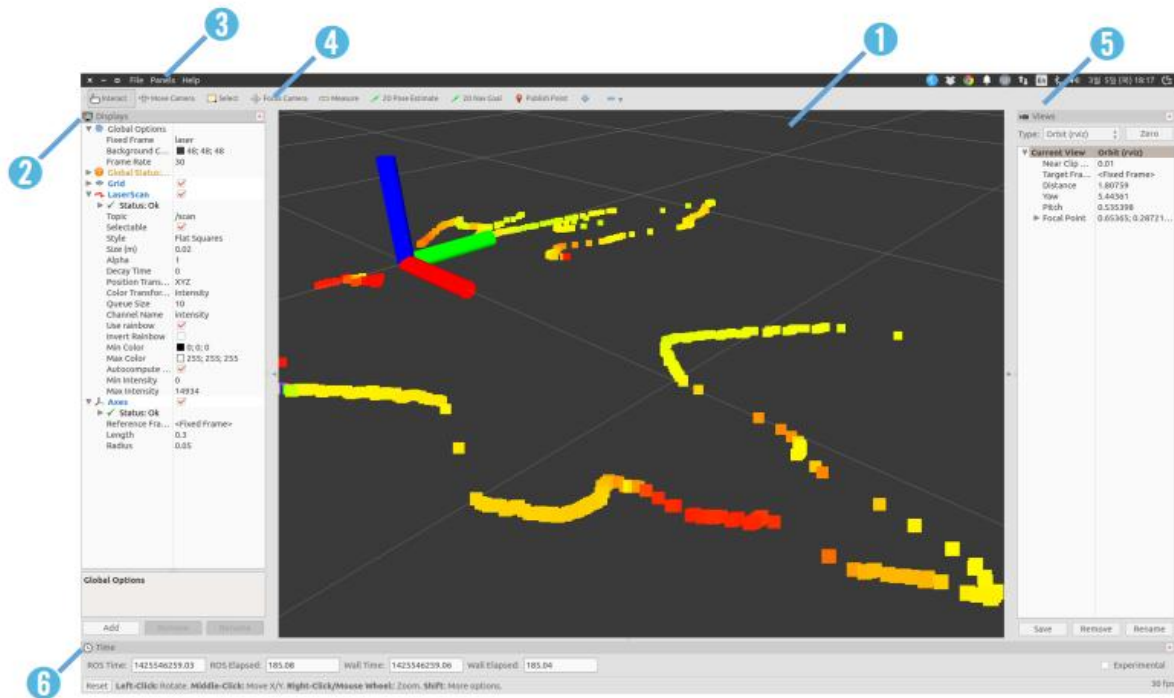
```

20 <xacro:macro name="cyl_inertia" params="mass r h">
21   <inertial>
22     <mass value="{mass}" />
23     <inertia ixx="{(1/12)*mass*(3*r*r+h*h)}" ixy="0.0" ixz="0.0"
24             iyy="{(1/12)*mass*(3*r*r+h*h)}" iyz="0.0"
25             izz="{(1/2)*mass*r*r}" />
26   </inertial>
27 </xacro:macro>

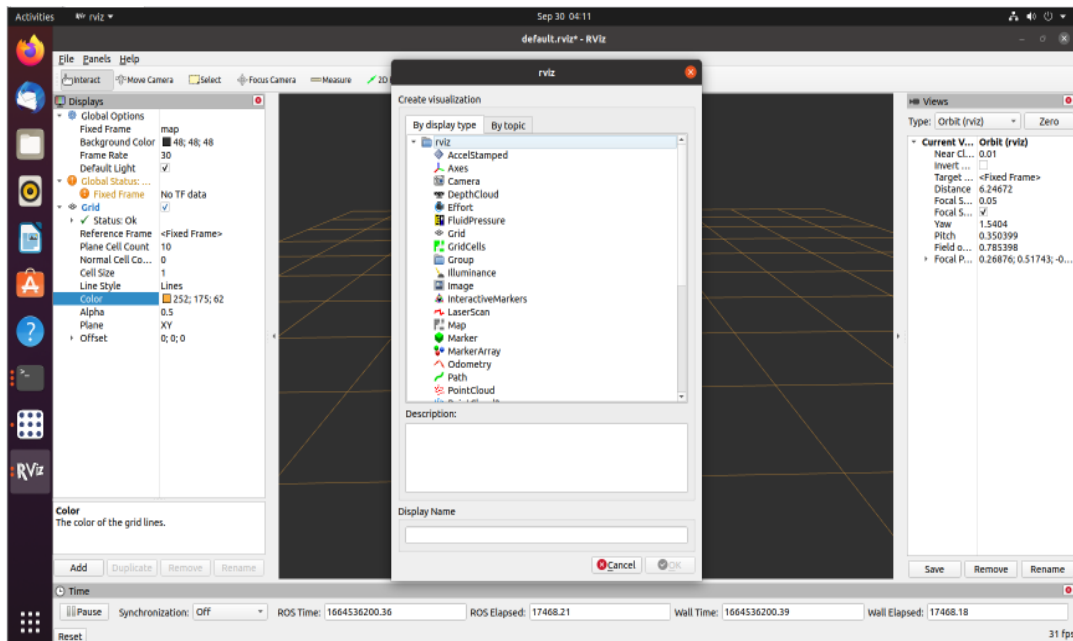
```

APPENDIX 6: COMPENENTS RVIZ ENVIRONMENT

6.1 Rviz On-Screen Components



- (1) – *3D view*: this black area is located in the middle of the screen. This is the main screen that allows us to see different data in 3D. Options such as background color, fixed frame, grid can be configured in the Global Options and Grid settings in the left column of the screen.
- (2) – *Screens*: the screen panel in the left column to select the data that we want to display from different topics. If we click [add] in the lower left corner of the control panel, the selection screen will appear. There are currently more than 30 types for us to choose from, which will be explained more later.



- (3) – *Menu*: The menu bar is located at the top of the screen. They can select commands to save or load the current displays, and can also select different tables.
- (4) – *Toolbar*: the tools are located below the menu bar, where we can select nodes for different functions such as interaction, camera movement, selection, changing camera standards, measuring distances, estimating 2D positions, 2D navigation target points, publishing points.
- (5) – *View*: the view panel configures the viewing angle of the 3D view.
- (6) – *Time*.

6.2. Rviz Display

The most frequently used menu in RViz is probably the Menu Displays⁵. The following are the details of the Menu.

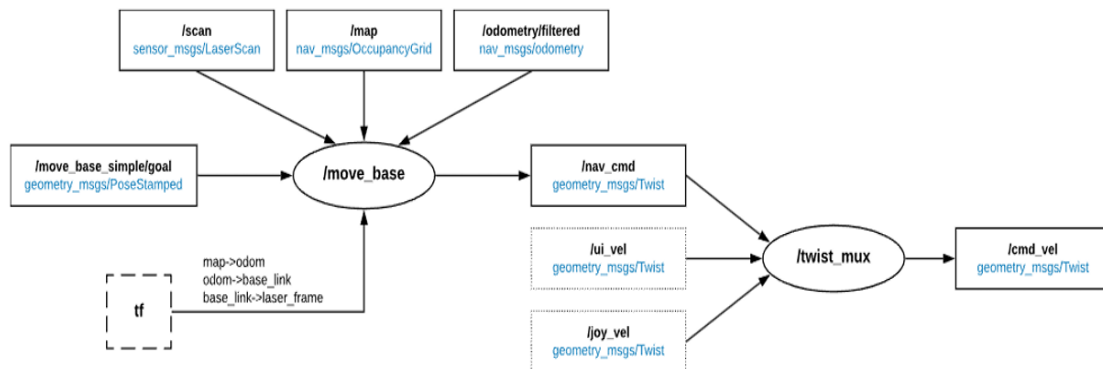
Icon Name	Describe
 Axes	XYZ axis display
 Camera	Create a new render window from the camera perspective
 Effort	Displays the force acting on each rotating joint of the Robot
 FluidPressure	Displays the pressure of liquids such as air and water
 Grid	2D or 3D grid display

 Grid cells	Show each tile of the grid, it is mainly used to show the obstacles in the navigation diagram
 Group	This is a container for grouping monitors, which allows us to manage the monitors that are being used into a group
 Illuminance	Illuminance Display
 Image	Display images in a new rendering window
 InteractiveMarkers	Display interactive markers. We can change the position (x,y,z) and rotate it with the mouse
 LaserScan	Displays laser scan values
 Map	Residential map display, used in navigation
 Marker	Display markers such as arrows, circles, triangles, rectangles, cylinders provided by RViz
 MarkerArray	Show multiple markers
 Path	Show the path of the robot used in navigation
 Point Cloud	Displays point cloud data. This is used to display sensor data from depth cameras such as kinect, xtion,...
 Point Cloud2	Point Cloud2 is compatible with the latest point cloud library (PCL), we can usually use PointCloud2
 Polygon	Polygon contour display, it is mainly used to display the outline of the robot in the 2D plane
 Pose	Display posture (position+orientation) in 3D. The pose is expressed as an arrow where the root of the arrow is the position (x,y,z) and the direction of the arrow is the direction
 Pose Array	Show multiple poses
 RobotModel	Robot model display
 TF	Display TF conversion coordinates used in ROS. It shows with xyz axes
 Temperature	Temperature Display

APPENDIX 7: MOVE_BASE PACKAGE

7.1 Move_base:

The Move Base is an ROS package that allows automatic control of the robot's movement from its current position to its assigned target in a 2D environment.

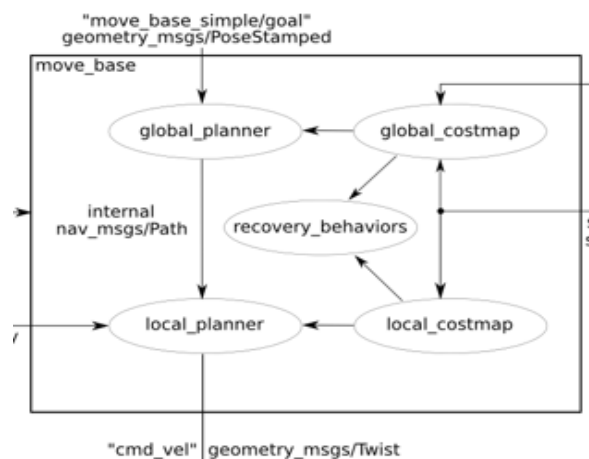


Relationship diagram between Move Base and the rest of the ROS Navigation Stack

Move Base collects environmental data from related packages to define a global plan in a known map in advance.

When receiving a move command from the Topic `"/move_base_simple/goal"`. The Move Base executes the movement command to the robot's motion actuators. In this simulation, Plugins `"cmd_vel"` take care of the movement for the robotic drivetrain.

The Navigation Stack uses two costmaps to store obstacle information.

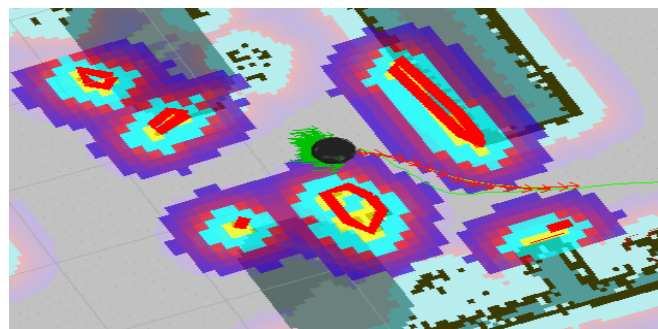


Components of move_base

- `Global_costmap`: used to create a long-term plan for the entire map.... For example, calculate the shortest path from A to B.
- `Local_costmap`: used to create short-term environmental plans.... For example, avoiding obstacles within the range of the robot visible in its way.

There are a number of configuration options that both have to follow, and some options for each map individually, so we'll have three configuration files.

`move_base` used to control the robot to move from its current position to its assigned target. It combines information from the AMCL, global planner, and local planner to create a safe and efficient path. `move_base` also manages special situations to avoid obstacles, handle errors, and ensure that the robot achieves its goals.



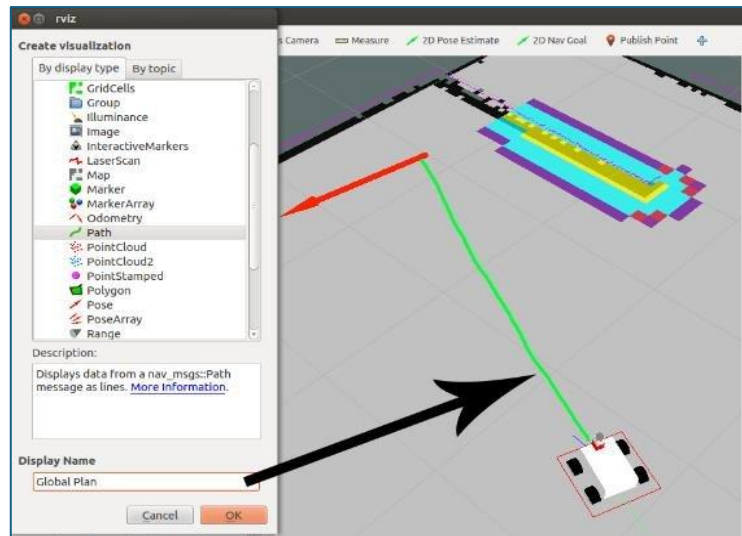
`Global_costmap` is a global map with a gray representation color, `Local_costmap` is a local map interpreted with custom colors

7.2 Global Planner

The Global Planner's mission is to find a safe and efficient path from the robot's current location to the designated target. The Navigation Stack provides three main Global Planner algorithms: `global_planner`, `navfn`, and `carrot_planner`.

I use the `navfn` algorithm with the wayfinding algorithm is **Dijkstra**.

Advantages	• Calculates reliable paths and finds the shortest route. Supports basic obstacle avoidance scenarios. • High computation speed, easy to use, suitable for applications that require stability and predictability.
Disadvantages	• Lacks flexibility in defining movement constraints, such as the robot's turning radius limitation. • Performs poorly in situations where the target is located near obstacles



The safe path is set up with a green line

7.3 Local Planner

- Local_planner use the "Dynamic Window Approach (DWA) as a method for planning obstacle avoidance and obstacle avoidance.

Algorithm used	Dynamic Window Approach (DWA): scans dynamic windows to search for potential paths and evaluates the feasibility of routes based on information about the robot's current speed and acceleration.
Advantage	• High reliability: DWAPlanerROS is capable of generating safe and efficient moving routes, ensuring the robot avoids collisions and reaches its goals. • High compatibility: Can work with robots capable of nonlinear motion, such as assembly robots or small mobile robots.
Defect	Search space limitation: DWAPlanerROS's route generation based on dynamic window scanning can limit the search space and miss better routes.

- Local_planner is responsible for calculating the velocity commands sent to the robot's mobile facility via base_control while avoiding obstacles that may collide in its path.
- It needs to set some configuration options based on the robot's specifications to get up and running.

- Create a file named "base_local_planner_params.yaml" as shown below.

```
TrajectoryPlannerROS:

# thông số robot
max_vel_x: 0.18      # vận tốc tối đa cho trục x (m/s)
min_vel_x: 0.08      # vận tốc tối thiểu cho trục x (m/s)

max_vel_y: 0.18      # vận tốc tối đa cho trục y (m/s)
min_vel_y: 0.08      # vận tốc tối thiểu cho trục y (m/s)

max_vel_theta: 1.0   # vận tốc tối đa cho trục theta (m/s)
min_vel_theta: -1.0  # vận tốc tối thiểu cho trục theta (m/s)
min_in_place_vel_theta: 1.0

acc_lim_x: 1.0       # giới hạn cho gia tốc trục x (m/s^2)
acc_lim_y: 0.0       # giới hạn cho gia tốc trục y (m/s^2)
acc_lim_theta: 0.6    # giới hạn cho gia tốc trục theta (m/s^2)

# dung sai điểm mục tiêu
xy_goal_tolerance: 0.10 # khoảng cách dung sai điểm mục tiêu (m) trục xy
yaw_goal_tolerance: 0.05 # dung sai điểm mục tiêu trục yaw (radian)

# Differential-drive robot configuration
holonomic_robot: true

# thông số mô phỏng chuyển tiếp
sim_time: 0.8         # thời gian quy đạo mô phỏng chuyển tiếp
vx_samples: 18         # số lượng mẫu trong không gian vận tốc trục x
vtheta_samples: 20     # số lượng mẫu trong không gian vận tốc trục y
sim_granularity: 0.05  # số lượng mẫu trong không gian vận tốc trục theta
```

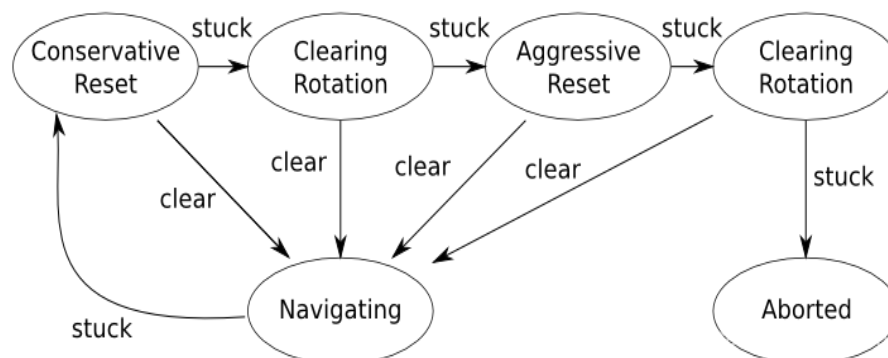
7.4 Base controller

The Base Controller is the component that controls the robot's movement. Its mission is to receive movement commands from the Planner and control the robot's mechanical parts to make movements based on that command.

```
1 shutdown_costmaps: false      # không cho costmap tắt khi robot dừng di chuyển
2 controller_frequency: 10.0    # tần số điều khiển local planner
3 planner_patience: 5.0         # thời gian tối đa tìm kiếm đường dẫn đến mục tiêu
4 controller_patience: 15.0     # thời gian cho tối đa của bộ điều khiển
5 conservative_reset_dist: 3.0 #
6 planner_frequency: 5.0
7 oscillation_timeout: 10.0
8 oscillation_distance: 0.2
```

7.5 Recovery Behavior

move_base Default Recovery Behaviors

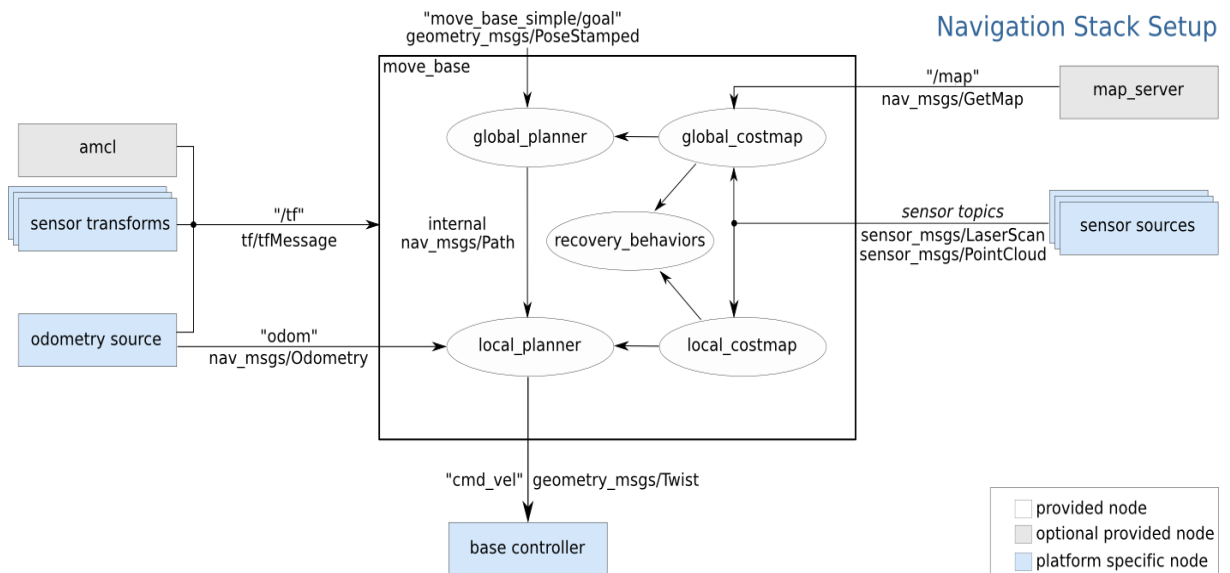


Recovery Behavior handles difficult or unexpected situations during the robot's maneuvering. When the robot is stuck, unable to find a valid path, or oscillating uncontrollably.

APPENDIX 8: NAVIGATION STACK

The Navigation Stack in ROS is a set of software and algorithms to control the robot to move in a 2D or 3D spatial environment. It is used to autonomously move the robot from the starting position to the target location safely, avoiding obstacles that the robot can see in its way.

8.1 Navigation Stack Structure



Navigation Stack Setup Diagram

To configure the Navigation Stack for a robot, we must provide the following packages of functions and inputs:

- **odometry source:** The robot's odometry data shows the robot's position relative to its starting position. The odom value will be published to the Navigation Stack with the message type **nav_msgs/Odometry**.
- **sensor transforms/tf:** The robot will publish the relationship between the robot coordinate frames using ROS tf.
- **base controller:** receives the output data and converts it into velocity commands sent to the robot.

- **sensor sources:** provide laser scan data or Navigation Stack to map the Robot environment. This data is combined with the odometry source to build the robot's Costmap. The sensor data is of the *sensor_msgs/LaserScan* type.

In addition to the above data, the Navigation Stack is built on three main components: AMCL, map_server, move_base.

8.2 Map_server:

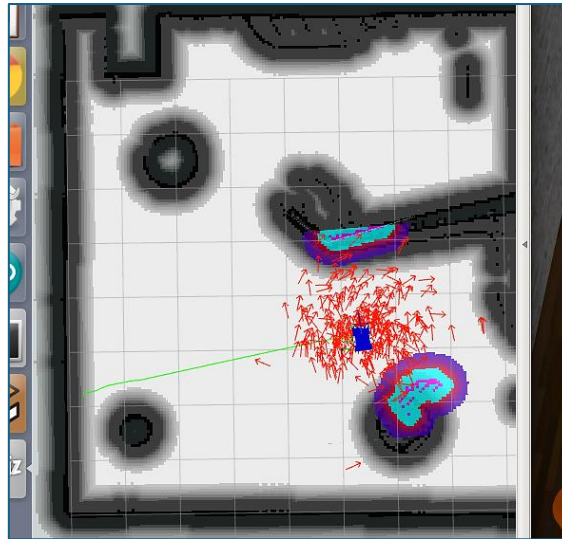
Map_server is a ROS Node that provides maps for the Navigation Stack. map_server provides environmental map information to AMCL, Global Planner, and Local Planner for use in the robot's planning and movement process. We use SLAM to build the map.

- In order for the robot to be able to navigate automatically, of course, it is indispensable to map the environment where it operates.
- Here, the map is made using SLAM technology combined with sensors mounted on the robot. LiDAR 2D mapping was presented in the slide "feasibility report on connecting LiDAR sensors to Jetson Nano". Mapping issues (e.g. 3D mapping rear camera combinations) will be improved in the near future.
- ROS contains a package called "*map_server*" consisting of 2 nodes that are used to store maps after creating and provide maps to the navigation system.
- Node "*map_saver*": used to save the map after it is created.
 - Run the command line "*roslaunch map_server map_saver -f my_map*".
 - After running *map_saver* we will receive 2 files as shown in the side image containing configuration information and images of the map.

8.3 AMCL

- To navigate around the map automatically, the robot needs to be able to position itself on the map. This is the function of the AMCL node.
- To do that, the AMCL node uses the MCL (Monte Carlo localization) algorithm.
- Basically, the AMCL node takes the input data from the Laser, the robot's road measurement value, the environmental map, and gives the estimated posture of the robot. The more the robot moves around the environment, the more data the

localization system will obtain, and the more accurate the posture estimation will be.

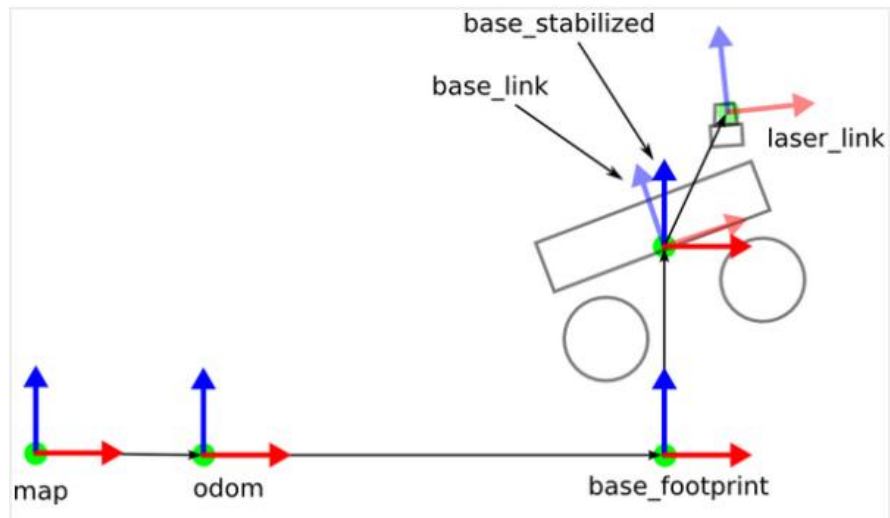


8.4 Sensor Transforms

Reading the data of the sensors is not enough. We need to convert the sensor data from its own coordinates to the reference coordinates of the Robot so that the Robot can use it.

Standard coordinate frames for mobile robots include: Map, Odom, Base_footprint, Base_link, Laser_link, etc.

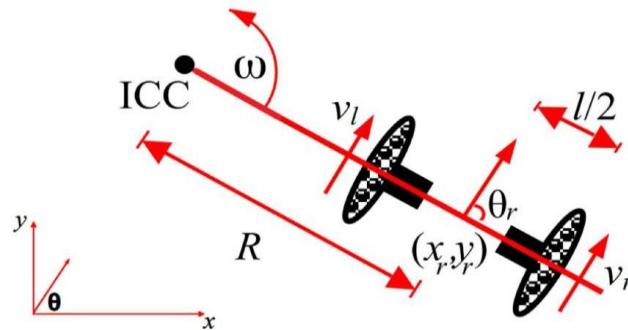
- Map: originated at a number of arbitrarily selected points, this coordinate frame is fixed, used to represent globally consistent distances.
- Odom: originated at the point where the robot was initialized, this coordinate frame is fixed, mainly used to represent local consistent distances.
- Base_footprint: Originated directly under the center of the robot, it is the 2D posture of the robot, this frame moves as the robot moves.
- Base_link: Originating at the center of rotation of the robot, this frame moves as the robot moves.
- Laser_link: Located at the center of the laser sensor, this frame is fixed relative to the base_link.



The placement of coordinate frames is determined by the right-handed rule, the figure on the slide in front of the red arrow corresponds to x, the blue arrow corresponds to z.

APPENDIX 9: DIFFERENTIAL DRIVE

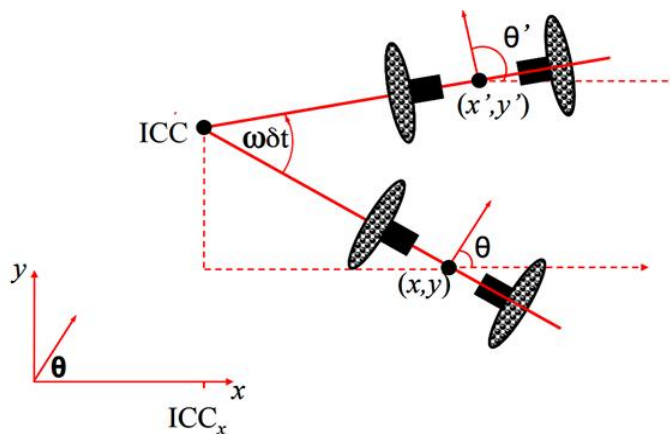
The system consists of 2 control wheels placed on the same common axle, each wheel can control the speed and direction independently called the transmission system differential dynamics.



Although the wheels can change speed independently when moving, the robot still must revolve around a point located on the common axis of the 2 wheels, this point is called ICC (Instantaneous Center of Curvature).

With the coordinate axis system (x, y) , we get the position of the robot as (x_r, y_r) which is at odds with the system

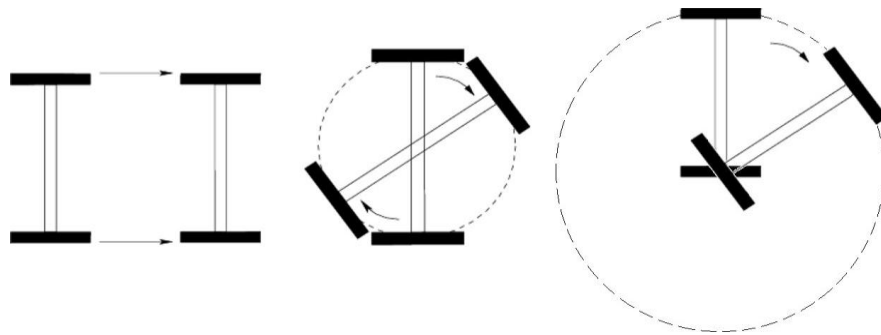
one-angle coordinates θ_r .



Since the rotation speed ω around ICC must be equal for both wheels, we have:

$$\begin{aligned} \omega \left(R + \frac{l}{2} \right) &= v_r \\ \omega \left(R - \frac{l}{2} \right) &= v_l \end{aligned} \quad \Rightarrow \quad \begin{aligned} R &= \frac{l}{2} \frac{v_r + v_l}{v_r - v_l} \\ \omega &= \frac{v_r - v_l}{l} \end{aligned}$$

- If $v_l = v_r$ then the robot moves in a straight line, $R = \infty$ and $\omega = 0$.
- If $v_l = -v_r$ then the robot moves clockwise with the center being the center of the 2-wheel seam, $R = 0$. The opposite will cause the robot to rotate counterclockwise.
- If $v_l = 0$ and $v_r \neq 0$ then the robot moves with the center as the left wheel and vice versa. In this case $R = \frac{l}{2}$



Forward kinematics: find the last position of the robot when there is a velocity of each wheel.

- From the velocity of v_r and v_l , we can find the location of ICC:

$$ICC = [x - R\sin(\theta), y + R\cos(\theta)]$$

- At the moment of $t+\delta t$, the robot's position would be:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \cos(\omega\delta t) & -\sin(\omega\delta t) & 0 \\ \sin(\omega\delta t) & \cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega\delta t \end{bmatrix}$$

Inverse kinematics: find the velocity of the wheels when the destination is known.

- We can describe the position of the robot moving in a specific direction θt and with a given velocity as follows:

$$x(t) = \frac{1}{2} \int_0^t [v_r(t) + v_l(t)] \cos[\theta(t)] dt$$

$$y(t) = \frac{1}{2} \int_0^t [v_r(t) + v_l(t)] \sin[\theta(t)] dt$$

$$\theta(t) = \frac{1}{l} \int_0^t [v_r(t) - v_l(t)] dt$$

- Some special cases:

- $v_r = v_l = v$

- $v_r = -v_l = v$

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x + v \cos(\theta) \delta t \\ y + v \sin(\theta) \delta t \\ \theta \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta + 2v \delta t / l \end{bmatrix}$$