TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

B-12-BME-2019-2024

DEVELOPMENT AND TESTING OF AUTONOMOUS MOBILE ROBOT FOR MATERIAL HANDLING

Nirmal Prasad Panta (076BME025)

Pawan Shrestha (076BME027)

Prince Panta (076BME029)

Saki Basnet (076BME036)

A PROJECT REPORT
SUBMITTED TO THE DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING
IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF BACHELOR IN MECHANICAL ENGINEERING

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING
LALITPUR, NEPAL

April, 2024

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A FINAL REPORT ON
DEVELOPMENT AND TESTING OF AUTONOMOUS MOBILE ROBOT FOR MATERIAL HANDLING

By:

Nirmal Prasad Panta (076BME025)

Pawan Shrestha (076BME027)

Prince Panta (076BME029)

Saki Basnet (076BME036)

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

LALITPUR, NEPAL

April, 2024

COPYRIGHT PAGE

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING, PULCHOWK CAMPUS

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a project report entitled "DEVELOPMENT AND TESTING OF AUTONOMOUS MOBILE ROBOT FOR MATERIAL HANDLING" submitted by Nirmal Prasad Panta, Pawan Shreshta, Prince Panta, and Saki Basnet in partial fulfillment of the requirements for the degree of Bachelor of Mechanical Engineering.

Supervisor, Dr. Surya Prasad Adhikari
Associate Professor
Department of Mechanical and Aerospace Engineering
IOE, Pulchowk Campus

External Examiner, Er. Rojesh Man Shikhrakar
Senior Manager
Fusemachines Nepal

Tribhuvan University
Institute of Engineering
Department of Mechanical and Aerospace
Pulchowk Campus

Committee Chairperson, Dr. Sudip Bhattrai
Assistant Professor
Head, Department of Mechanical and Aerospace Engineering
IOE, Pulchowk Campus

Date: 2024 / 04 / 10

ABSTRACT

The adoption of autonomous mobile robots (AMRs) for material handling has witnessed significant growth in various industries, including manufacturing, healthcare, and the service sector. To stay competitive in this era of automation, businesses are increasingly transitioning from human labor to AMRs for efficient transportation and material handling. This project involves developing and testing AMR that utilizes Simultaneous Localization and Mapping (SLAM) and Nav2 (Navigation2) for precise navigation and Computer Vision (CV) for enhanced material handling capabilities. In the first phase, design and development of the mobile robot is done. The second phase involves incorporating SLAM and Nav2 for autonomous mobility of the robot, enabling it to navigate complex environments with accuracy and efficiency and finally, in the third phase, OpenCV is integrated into the autonomous mobile robot for ArUco tag detection for material handling operations. Therefore, in these three phases, we have developed and tested an AMR for material handling purpose using ROS2, SLAM, Nav2 and OpenCV. The process used in this project can be a clear guideline on completing similar projects related to autonomous mobile robots using ROS2 in various areas ranging from manufacturing industries to service industries.

Keywords: *AMR, Design, SLAM, ROS, ROS2 Control, Nav2, CV, Testing*

ACKNOWLEDGEMENT

We are sincerely grateful to the Department of Mechanical and Aerospace Engineering, IOE, Pulchowk Campus, Lalitpur, and Head of Department Dr. Sudip Bhattarai for providing us an invaluable opportunity to undertake a project that allows us to delve into, enhance, and apply the knowledge and skills we have acquired throughout our Bachelor of Mechanical Engineering journey.

We express sincere gratitude to our supervisor Assistant Professor Dr. Surya Prasad Adhikari for his priceless guidance throughout this project. We would also like to thank Robotics Club, Pulchowk Campus for providing us with the required workspace and resources for the project. We are really indebted to Er. Nitesh Subedi, Er. Ramraj Khanal and Mr. Sabin Shrestha and all other members of the Robotics Club for assisting us with the problems that we encountered during the project.

Finally, we would also like to convey our thanks to the teachers, seniors, juniors, and our friends who have helped us in any way in completion of this project.

Nirmal Prasad Panta (076BME025)
Pawan Shrestha (076BME027)
Prince Panta (076BME029)
Saki Basnet (076BME036)

TABLE OF CONTENTS

COPYRIGHT PAGE ..... **ii**

ABSTRACT ..... **iv**

ACKNOWLEDGEMENT ..... **v**

TABLE OF CONTENTS ..... **vi**

LIST OF FIGURES ..... **ix**

LIST OF TABLES ..... **xii**

ABBREVIATIONS ..... **xiii**

SYMBOLS ..... **xiv**

CHAPTER 1: INTRODUCTION **1**

1.1 Background 1

1.2 Problem Statement 3

1.3 Objectives 4

1.3.1 Main Objective 4

1.3.2 Specific Objectives 4

CHAPTER 2: LITERATURE REVIEW **5**

2.1 Past Studies 5

2.2 Relevant Theories 9

2.2.1 Autonomous Mobile Robot (AMR) 9

2.2.2 AMR dynamic model 10

2.2.3 Robot Operating System (ROS) 11

2.2.4 Robot Operating System 2 (ROS2) 12

2.2.5 ROS2 Control 15

2.2.6 Joystick in ROS 18

2.2.7 tf2 18

2.2.8 Simultaneous Localization and Mapping (SLAM) 21

2.2.9 NAV-2 25
2.2.10 OpenCV 27
2.2.11 Pose Estimation 27
2.2.12 Fiducial Markers for Pose Estimation 28
CHAPTER 3: METHODOLOGY 30
3.1 System Used (For Development/Implementation) 31
3.1.1 Hardware used 31
3.1.2 Software used 31
3.2 Phase I: Transport 32
3.2.1 AMR Design 32
3.2.2 AMR Chassis fabrication 37
3.2.3 ROS2 Setup 40
3.2.4 Simulation of AMR in Gazebo and Rviz 44
3.2.5 AMR Control 55
3.3 Phase II: Improvement of transportation 68
3.3.1 SLAM Deployment using LIDAR 69
3.3.2 NAV2 (AMR Navigation) 77
3.4 Phase III: Functionality to transportation 78
3.4.1 CV Setup 78
3.4.2 Material handling 82
CHAPTER 4: RESULTS AND DISCUSSION 97
4.1 Output 97
4.2 Work Completed 98
4.2.1 Phase I 98
4.2.2 Phase II 101
4.2.3 Phase III 102
4.3 Limitations 104

4.4 Problems Faced 105
4.4.1 Gazebo Lidar 105
4.4.2 Limited resources 106
4.4.3 Failure in debugging problems encountered. 106
4.4.4 Ros2_control and diff_drive arduino 106
4.4.5 Serial Communication 106
4.4.6 Camera Calibration and ArUco Marker Distance Measurement 107
4.4.7 Spin Functionality 107
4.5 Budget Analysis 108
4.6 Work Schedule (Gantt Chart) 108
CHAPTER 5: CONCLUSION AND FUTURE ENHANCEMENT 110
References 111

LIST OF FIGURES

Figure 1.1: Number of mobile robots used in industry [5] 2
Figure 2.1: Building block of Autonomous Mobile Robots [13] 8
Figure 2.2: Schematic for developing the dynamics model [5] 10
Figure 2.3: Architecture of ROS2 Control [18] 15
Figure 3.1: Methodology Flowchart 30
Figure 3.2: Transparent view of robot 32
Figure 3.3: General dimension of robot shown in third angle projection 33
Figure 3.4: Model Reference 33
Figure 3.5: Displacement plot 35
Figure 3.6: Stress plot 36
Figure 3.7: Adaptive convergence graph of displacement 36
Figure 3.8: Adaptive convergence graph of axial stress 37
Figure 3.9: Top view of the robot frame 37
Figure 3.10: AMR frame with motor and wheels placed 38
Figure 3.11: Encoder enclosure and couplings 39
Figure 3.12: Coupling of Orange Hall encoder with hub motor 39
Figure 3.13: Position of rotary encoder in the AMR frame 40
Figure 3.14 rqt_graph used to visualize nodes and topics in slamtoolbox 43
Figure 3.15: URDF model of chassis 47
Figure 3.16: Chassis with wheels 47
Figure 3.17: Chassis with caster wheels 48
Figure 3.18: Chassis with camera and LiDAR 48
Figure 3.19: Map generated via slam_toolbox of a simulated world in gazebo 53
Figure 3.20: Saving the map in RVIZ 54
Figure 3.21: Providing goal pose in RVIZ 54
Figure 3.22: Steering direction 56
Figure 3.23: Cytron Motor Driver connected to Arduino using breadboard 57
Figure 3.24: PID tuning curve K_p=1, K_o= 50 and rest 0 59
Figure 3.25: PID tuning curve K_p = 0.8, K_i = 0.0001, K_d = 4 and K_o = 50 59
Figure 3.26: PID tuning curve for Kp= 0.75, Ki= 0.0004 and Kd=2 60
Figure 3.27: Running motors using ros2 node teleop_twist_keyboard 62

Figure 3.28: Overall pipeline of AMR control (phase I) 62
Figure 3.29: Actual control circuit of the AMR (phase I) 63
Figure 3.30: ROS2 control system under action 64
Figure 3.31: Checking the accuracy of odometry 67
Figure 3.32 Joystick Teleoperation 68
Figure 3.33: LiDAR positioned on top of the plate 69
Figure 3.34: LiDAR scan in RVIZ 71
Figure 3.35: AMR Autonomous Locomotion Control Circuit 74
Figure 3.36: Mapping of the real environment using Slam Toolbox 76
Figure 3.37: Saved map from Slam Toolbox 76
Figure 3.38: Cost map generated with Nav2 77
Figure 3.39: Calibration GUI 79
Figure 3.40: 4X4_50 id:4 tag 80
Figure 3.41: CV setup and ArUco tag detection 81
Figure 3.42: Adding vertical actuation mechanism. 82
Figure 3.43: Geometry of struts 83
Figure 3.44 : Position of camera 84
Figure 3.45: Motor signal distribution Arduino UNO R3 shield. 85
Figure 3.46: Motor signal distribution schematic 85
Figure 3.47: Printed PCB prototype on copper-clad 86
Figure 3.48: Arduino Nano Shield for motor driver 87
Figure 3.49: Circuit Fabrication and Testing 87
Figure 3.50 Final circuit schematic. 88
Figure 3.51 Final actual circuit. 88
Figure 3.52: Teleop material handling using joystick 91
Figure 3.53: Aruco tag in gazebo 92
Figure 3.54: Robot moving inside the table in gazebo 92
Figure 4.1: Developed and Tested AMR 98
Figure 4.2: Robot model in Gazebo simulation 99
Figure 4.3: Visualization of lidar point clouds and camera in Rviz 100
Figure 4.4: AMR setup for teleoperation being tested 100
Figure 4.5: Creation of map using SLAM Toolbox in Robotics Club 101
Figure 4.6: Camera attachment 102

Figure 4.7: Initial remote material handling test of the AMR 103
Figure 4.8: Autonomous docking and lifting of payload by the AMR 104
Figure 4.9: Map error 105
Figure 4.10: Gantt Chart 109

LIST OF TABLES

Table 3.1: Material Properties.....34
Table 3.2: Loads.....35
Table 3.3: Cut List of AMR Frame.....38
Table 4.1: Base Cost 108
Table 4.2: Items to be pledged. 108
Table 4.3: Work Schedule.....109

ABBREVIATIONS

AGV	Autonomous Guided Vehicle
AMR	Autonomous Mobile Robot
API	Application Programming Interface
AR	Augmented Reality
CAD	Computer Aided Design
COVID-19	Corona Virus Disease of 2019
CV	Computer Vision
DDS	Data Distribution Service
DSO	Direct Sparse Odometry
DTAM	Dense Tracking and Mapping
FOV	Field Of View
GNSS	Global Navigation Satellite System
ID	Identification
IMU	Inertial Measurement Unit
LAN	Local Area Network
LIDAR	Light Detection and Ranging
LSD	Large-Scale Direct
Nav2	Navigation2
PID	Proportional Integral Derivative
PTAM	Portable Tracking and Mapping
QR	Quick Response
RGB-D	Red, Green Blue-Depth
ROS	Robot Operating System
RVIZ	Robot visualization
SLAM	Simultaneous Localization and Mapping
UAS	Unmanned Aircraft Systems
UNIX	Uniplexed Information and Computing Service
URDF	Unified Robotics Description Format

SYMBOLS

v_x	Forward Velocity
v_y	Lateral Velocity
G	Center of mass
ω	Angular Velocity
F_{rrx}	Longitudinal force on the right wheel
F_{rlx}	Lateral force on the right wheel
F_{rry}	Longitudinal force on the left wheel
F_{rly}	Lateral force on the left wheel
I_z	Moment of Inertia
q	Quaternion
ϕ	Roll angle
θ	Pitch angle
ψ	Yaw angle
γ	Parametric variable of path
θ	Angle made by AMR body frame with global frame
v^{world}	Angular velocity
v^{body}	Robot velocity with respect to body frame
v_l	Linear velocity at the left wheel contact point
v_r	Linear velocity at the right wheel contact point
ω_z	Angular velocity of robot about vertical z axis
b	Wheel separation
ω_l	Angular velocity of left wheel
ω_r	Angular velocity of right wheel
r	Radius of wheel
K_p	PID proportionality constant
K_i	PID integral constant
K_d	PID differential constant
K_o	PID overall constant factor

CHAPTER 1: INTRODUCTION

1.1 Background

In a world in which the rate of change is ever-increasing, agile corporations will be the ones to sustain and thrive. In manufacturing, being agile means being able to make many different products. To be agile, industries not only need to be able to make different products but also to adapt their products and introduce new ones in a short time [1]. It allows industries to configure, reconfigure and provide a customized product that fits customers' needs. It requires being able to reconfigure operations, processes, and business. For that robots are in the center for autonomous and adaptable production process.

The recent estimations confirm that global spending on military robotics will be $16.5 billion in 2025 [2]. 88% of businesses worldwide plan to adopt robotic automation into their infrastructure [3]. In Q2 2021, robotics orders grew with 67% [4]. For companies like Amazon with warehouses from 400,000 and up to 1,000,000 square feet, a few workers just can't manage. According to statistics, warehouses as big as that of Amazon not utilizing cobots- collaborative robots, will find that 80% of workers' time will be spent navigating through the whole warehouse. This means only 20% of workers' time will be spent on productive tasks like fulfilling orders. With cobots like "Kiva," Amazon has successfully reversed the trend, such that employees are spending 80% of their time fulfilling orders and only 20% of their time navigating through warehouses. When you combine that with an effective inventory management system you have the recipe for Amazon's success [2].

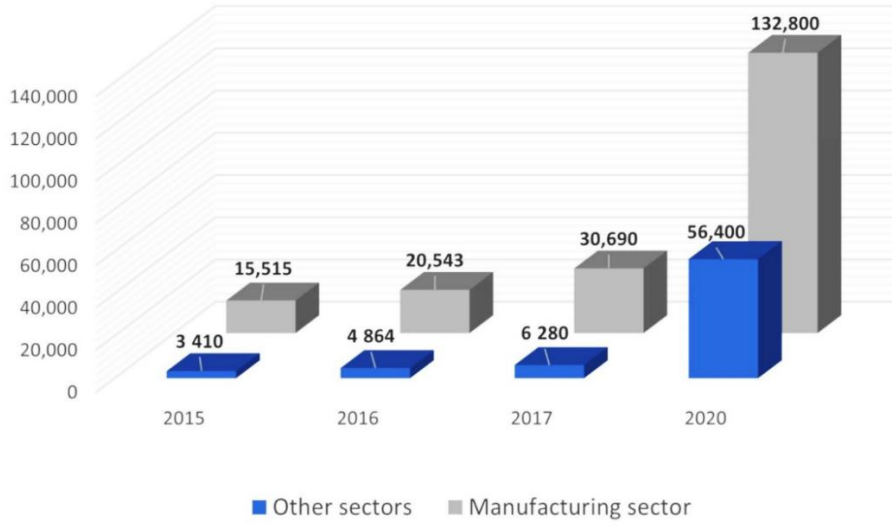


Figure 1.1: Number of mobile robots used in industry [5]

Robotics went from luxury to a necessity, coming to the point where vision-embedded robot operations allow for the following: when combined with computer vision, robotic systems can now have precise pick-and-place capacities, figure out the important assembly pieces from the storage, and set them in their correct locations. The way forward promises a greater diversification of activities performed by robots. Robot vision is much more flexible and capable of providing a broader palette of functionalities. The use of machine and computer vision algorithms to visualize, contextualize and act upon the changes in the environment can be paramount [6].

Similarly, the role of autonomous mobile robots (AMRs) in manufacturing and production is significant, especially in cellular manufacturing. In cellular manufacturing, machines or processes with similarities are grouped together to create a cell, allowing for efficient and streamlined production. Cellular manufacturing is commonly used when there is a high variety of products or when customization is required. A simple cellular manufacturing system can be enhanced by employing AMRs. It can be programmed to perform various tasks and navigate complex factory layouts. They can adapt to changing production needs, allowing for scalability and flexibility in manufacturing processes. AMRs can be easily reprogrammed or

redeployed to accommodate new products or production lines. AMRs can be programmed to respond to kanban signals, in our case ArUco markers, to retrieve empty containers or deliver replenishment materials based on the requirements. This streamlines the material replenishment process and helps maintain optimal inventory levels. This ensures that each work cell receives the necessary components precisely when they are needed, reducing waiting times and minimizing inventory levels and ensuring Just-in-Time Delivery.

The need for AMRs in the production flow and material handling was felt with changing products and processes, but now the urgency to implement automation technologies like AMRs has become increasingly evident as companies strive to navigate the challenges posed by the pandemic and build resilient supply chains for the future. By reducing labor requirements and increasing operational efficiency, AMRs can lead to cost reductions, improved productivity, and enhanced profitability in the long run and avoid disruptions such as labor shortages, health risks and safety concerns with a flexible material flow system that runs 24/7 [7].

1.2 Problem Statement

The need for agile and flexible manufacturing processes and efficient warehouse/inventory management is increasingly critical in our rapidly changing world. Industries must have the capacity to quickly produce, adapt, and introduce a wide range of products, necessitating the reconfiguration of operations, processes, and materials. By leveraging navigation and computer vision algorithms, these technologies offer versatile capabilities for diverse robotic tasks. To address these demands, Automated Guided Vehicles (AMRs) emerge as an efficient and flexible solution, ideal for customization. AMRs can be programmed to streamline material flow, maintain optimal inventory levels, and enable just-in-time delivery. The COVID-19 pandemic has underscored the urgency of implementing automation technologies like AMRs to build resilient supply chains, overcome labor shortages, ensure operational efficiency, and minimize health and safety risks.

Thus, the problem at hand is the implementation of an AMR-based automation system that enhances manufacturing agility, material handling capability, productivity, cost-effectiveness, and operational resilience while facilitating efficient material flow and ensuring just-in-time delivery.

1.3 Objectives

1.3.1 Main Objective

To develop an autonomous mobile robot using Simultaneous Localization and Mapping (SLAM) and Nav2 for mapping, localization, and navigation with computer vision (CV) for material handling

1.3.2 Specific Objectives

i) To create an AMR prototype and use conventional control for navigation
ii) To employ SLAM algorithm for mapping and Nav2 to enhance the AMR's autonomous navigation, precise localization, and effective obstacle avoidance
iii) To integrate computer vision technology to augment the AMR's material handling capabilities, enabling it to accurately identify and handle payloads

CHAPTER 2: LITERATURE REVIEW

2.1 Past Studies

This literature review highlights the historical evolution of Automated Guided Vehicle Systems (AGVS) and their transition to modern Autonomous Mobile Robots (AMRs) for material handling. Key eras in AGVS development are outlined, emphasizing advancements in technology. The shift to AMRs involves integrating Simultaneous Localization and Mapping (SLAM) algorithms and computer vision (CV) technologies. Challenges and opportunities in integrating autonomous intelligent vehicles (AIVs) into manufacturing environments are discussed. The review concludes with the introduction of the SLAM Toolbox, addressing the need for accurate mapping in dynamic spaces and setting the stage for developing an autonomous mobile robot for material handling. This provides a roadmap for understanding AMR evolution and challenges in automated material handling.

Automated Guided Vehicle Systems have evolved over a seven-decade history, originating in America around 1953. The first era, spanning nearly 20 years, saw the introduction of simple track-guided systems with basic tactical sensors like bumpers. In the early 1960s, European companies, such as Jungheinrich and Wagner, entered the AGVS scene. The second era, from the 1970s to the early 1990s, witnessed significant technological advances, with the automotive industry driving demand. High-performance electronics, microprocessors, and programmable logic controllers (PLC) emerged, allowing for more complex scenarios. Active inductive track guidance became the standard, using AC-charged conductors inducing a current in coils beneath the vehicle for steering control. The era also witnessed improved sensory technology, infrared and radio signals for data transfer, and advancements in load handling automation. However, a recession in the late 1980s temporarily affected the industry. The third era, from the mid-1990s to around 2010, introduced electronic guidance with contact-free sensors. The vehicles were controlled by standard PCs, equipped with either a smart power stage (SPS) or a microprocessor. Conductive cable guidance faded, and alternative navigation technologies like magnetic and laser navigation gained prominence. WLAN became a common means of data transfer. This era emphasized

reliability and proven technology for intralogistics, offering a diverse range of well-tested technologies for AGVS applications. [8]

The above paragraph explains the evolution of AGVs; however, their need remains still unclear. W. Grzecha mentions the issue of product modularity, which allows producing different products through combination of standard components. [9] The special structure of modular products provides challenges and opportunities for operational design of assembly lines. The paper proposed an approach for design of assembly lines for modular products. Unlike a single assembly line, Assembly line – flow shop and flow shop – assembly line structure was discussed. In such ever-changing manufacturing techniques and product evolution, `manufacturing systems are more prominent. This requires the transport of raw materials, semi-finished products and products from inventory, job stations and assembly lines to respective places, where AGVs, AMRs as well as AIVs can have a significant impact. Flexible manufacturing systems can bring tremendous economic advantages to batch manufacturers. Beyond the attraction of increased efficiency, companies must automate if they are to compete in foreign and domestic markets with companies in Japan, Germany, and other foreign countries, which are automating their manufacturing operations vigorously.

Automated guided vehicle systems (AGVSs) have received increased attention by the designers and engineers of automated manufacturing systems. AGVSs are widely used in FMSs as they provide flexibility in routing parts among elements present in the system. These systems are highly complex and expensive due to the dynamic environment in which FMS functions. Furthermore, if the AGVS is not efficient, the whole system performance may be impaired by the possible starvation of machines in the system. [10]

The traditional AGV systems rely on pre-defined routes, often marked by magnetic tapes or color lines on the floor. These routes are followed by AGVs equipped with various sensors, safety bumpers, and communication devices. A traditional AGV obstacle sensor, specifically the HOKUYOU PBS-03JN, scans the environment and communicates with a PC for real-time monitoring. The limitations of traditional AGV

systems include their dependency on fixed routes, potential delays in the presence of obstacles, and issues related to maintenance and tag mismanagement. In contrast, A modern solution using a virtual map for autonomous AGV navigation is better. The virtual map is created through laser sensors or GPS, allowing for more flexible and precise manoeuvring without the need for physical markings on the floor. The findings suggest that autonomous AGV systems, with features like virtual mapping, offer advantages in terms of flexibility, cost-effectiveness in route creation, and improved obstacle avoidance compared to traditional AGV systems. [11]

Lynch, L. conducted a comprehensive review of Automated Ground Vehicles (AGVs) and sensor technologies, focusing on their roles in manufacturing processes [12]. The study explores various types of AGVs, such as Towing Vehicles, Unit Load Vehicles, Pallet Trucks, and Forklifts, each designed for specific tasks within flexible manufacturing systems. The authors delve into the crucial aspect of AGV navigation systems, discussing Laser Guidance, Line Following Guidance, Magnetic Spot Guidance, and Barcode Guidance. A detailed comparison of these navigation systems was presented, considering factors such as cost, complexity, flexibility, efficiency, and ease of expansion. The paper also highlights the pivotal role of sensors, particularly Laser Scanners and Magnetic Guide Sensors, in enabling AGVs to navigate through their environments. The findings provide valuable insights into the selection and deployment of AGVs based on the nature of tasks and operational environments, exemplified by real-world cases such as the Kiva Robot in logistics warehouses and the Dematic AGV in manufacturing production settings. The authors emphasize the increasing importance of AGVs in improving automated processes and fostering collaboration in manufacturing spaces. [12]

The use of AGVs relied mostly on the static environment for motion control. The ability to self-navigate considering the dynamic workspace lacked in AGVs. The introduction of Autonomous Mobile Robots (AMRs) overcomes this shortcoming of AGVs. As such, AMRs possess a control system capable of determining the actions necessary to perform the required tasks. They can navigate while avoiding obstacles in a dynamic environment. They have the following building block as shown below:

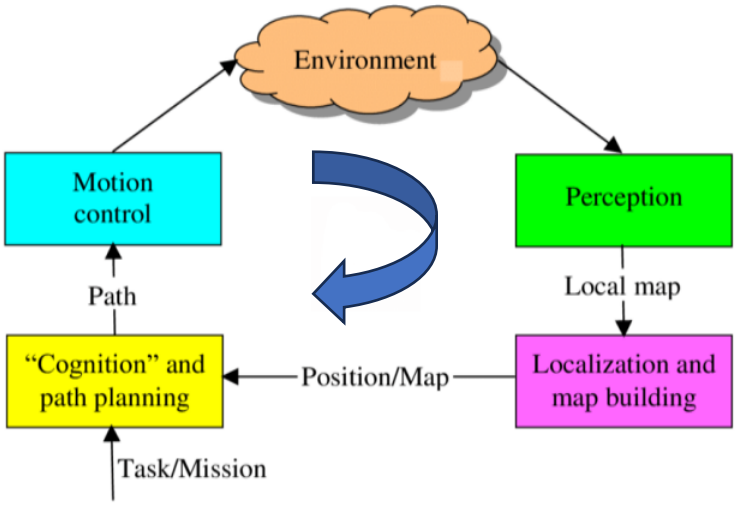


Figure 2.1: Building block of Autonomous Mobile Robots [13]

Now with the integration of machine learning and artificial intelligence, these AMRs have also been termed autonomous intelligent vehicles. The introduction of AIVs (Automated Inteligent Vehicles) in flexible manufacturing systems requires addressing issues such as software interfacing, wireless communication between AIVs and manufacturing execution system (MES), accurate identification of AIV availability, navigation, route optimization, and correct payload handling. There is importance in developing algorithms for dynamic decision-making, managing fleets of AIVs, and adapting equipment with sensors to enhance overall system visibility and efficiency. Careful planning and technological adaptations are needed to fully leverage the benefits of AIVs in smart production lines. [14]

With the advancement of AGVs and AMVs, the use of LIDAR for SLAM has been extensive. Various past studies suggest its satisfactory results and reliability. Grisette, G. et al. presented a tutorial on graph-based SLAM, which revolutionizes mobile robotics by formulating the simultaneous localization and mapping (SLAM) problem as a graph [15]. In this framework, nodes represent robot poses, and edges encode spatial constraints derived from sensor measurements. The paper emphasizes the renaissance of graph-based SLAM due to advancements in solving the associated error minimization problem. It explores the probabilistic formulation using dynamic

Bayesian networks, shedding light on the temporal structure of SLAM. The tutorial also delves into practical applications, showcasing the efficiency of graph-based SLAM in 2D and 3D laser-based mapping scenarios. Ultimately, the tutorial equips readers to implement these cutting-edge methods, underscoring the necessity of understanding linear algebra, multivariate minimization, and probability theory in this evolving field. [15]

SLAM Toolbox is an open-source ROS package designed to address the challenges of mapping dynamic and large-scale environments for mobile robotics and autonomous systems. There is a lack of existing open-source SLAM algorithms, particularly in terms of mapping large spaces in real-time using mobile processors. SLAM Toolbox builds upon the legacy of Open Karto, offering accurate mapping algorithms, various modes of mapping, and additional tools such as kinematic map merging and improved graph optimization. The package has been integrated into ROS 2 Navigation2, demonstrating its real-time positioning capabilities in dynamic environments. SLAM Toolbox is shown to effectively map spaces as large as 24,000 m^2 in real-time, surpassing the capabilities of previously available SLAM libraries. The significance of SLAM Toolbox in meeting the unmet need for accurate mapping in large and dynamic spaces, positions it as a valuable tool for both industry and research applications. [16]

The review of past studies concerned us about the need and importance of Automatic Mobile Robot and the benefits of using LIDAR based SLAM for such robots and provided us with the necessary knowledge to carry out this project.

2.2 Relevant Theories

2.2.1 Autonomous Mobile Robot (AMR)

Autonomous Mobile Robot (AMR) are also called autonomous mobile vehicles. The term automatic mobile robots refer to robots that can navigate independently in the given environment by detecting and avoiding obstacles using sensors. [17] The AMRs can play a big role in flexible production line and thus promote agile manufacturing. In any production stage, the raw materials and unfinished goods must go through a series of manufacturing processes sequentially. The use of AMRs in such production

processes can assist in making the production line more flexible by picking and placing the unfinished goods in the required workstation as per the production process.

2.2.2 AMR dynamic model

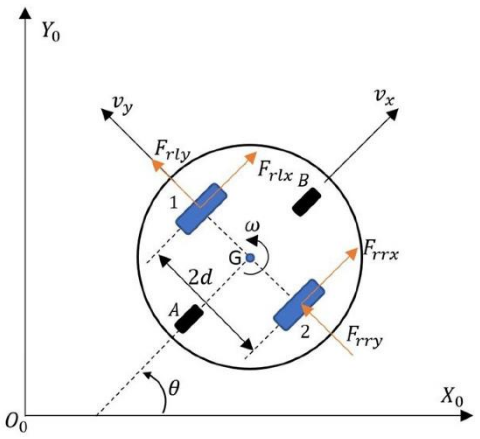


Figure 2.2: Schematic for developing the dynamics model [5]

Control signals are the linear and angular velocities of the robot for the kinematics described by equation (2.1):

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos\theta & 0 \\ sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} \tag{2.1}$$

G is the center of mass and center of rotation for the robot. v_x—forward velocity, v_y—lateral velocity, ω— angular velocity. The numbers 1 and 2 denote the drive wheels, while the letters A and B denote the supporting wheels. Then the driving forces are: F_{rrx}, F_{rry}—the longitudinal and lateral force of the right wheel and F_{rlx}, F_{rly}—the longitudinal and lateral force of the left wheel. By the law of conservation of linear and angular momentum:

$$\sum F_x = F_{rlx} + F_{rrx} = m(\dot{v}_x - v_y\omega) \tag{2.2}$$

$$\sum F_y = F_{rly} + F_{rry} = m(\dot{v}_y - v_x\omega) \tag{2.3}$$

$$\sum M_z = I_z \omega = \frac{d}{2}(F_{rrx} - F_{rlx}) \tag{2.4}$$

where:

m- mass of the robot,

I_z - moment of inertia of the robot with respect to the axis passing through the point

2.2.3 Robot Operating System (ROS)

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications [18]. It consists of various drivers, algorithms, developer tools to build a robotics project. ROS is unique in that its distributed and modular architecture is designed to facilitate the integration of various components and the sharing of information between different robots and devices.

The flexibility that ROS allows helps in providing support for both low-level hardware control and high-level task planning. Expanding on the high-level part, developers only need to focus on fine-tuning parameters, managing the nodes, their mode of communication pathway i.e., topics, services etc. Or maybe choosing an algorithm package (like for SLAM choosing slam gmapping package or slam toolbox) based on their use case without worrying about the nitty-gritty details related to its implementation.

Similarly, parts related to hardware like sensors which includes data acquisition, data processing are also managed by ROS, while the user only needs to fine-tune parameters exposed to them in the front-end [18]. These few features along with numerous other ones, enable us to focus on the robot and the task or objective at hand. In short, ROS satisfies the need for a modular, scalable, and reliable architecture; while providing features relevant to sensing, planning, mobility, autonomy, visualization, debugging etc.

ROS has two versions. The older one being ROS1 while the latest, revamped being ROS2. ROS 1 has a large ecosystem of sensor, control, and algorithmic packages made available by community contributions, enabling a small team to build complex robotics

applications and sets out to fulfill objectives as stated above. However, ROS1 isn't without its limitations. Here, security and network topology aren't prioritized. Problems like single point failure, reliability issues, struggling to deliver data over Wi-Fi also occurred. ROS2 addresses these issues, but rather than applying a band-aid solution on the existing ROS1, a re-design from the ground up was thus prompted.

The issue related to security and topology which was described above is addressed in ROS2 via the use of the DDS (Data Distribution Service) communication service which is an open-source standard. Its usage has prompted ROS2 to obtain reliable security, embedded and real-time support, multi-robot communication etc.

2.2.4 Robot Operating System 2 (ROS2)

ROS2 comprises of three categories which are Middleware, Algorithms and Developer tools. The middleware is composed of communication components, networking related APIs etc. The algorithms contain algorithm related packages like for SLAM, planning, navigation etc. This is so the end user need not concern themselves with the finer details like relevant mathematics and programmatic implementations. On the other hand, developer tools are tools, either command line or graphical for debugging, visualization, simulation, logging etc. There are also tools to create packages, build custom packages etc.

2.2.4.1 Design Principles

The following describes the design principle behind ROS2 in brief. These principles include Distribution, Abstraction, Asynchrony, Modularity etc.

Distribution: It relates to an idea of a distributed systems approach. A complex system basically comprises of components that are independent of each other in terms of functionality. However, these components or modules are connected in the sense that they communicate to each other via explicit communication pathways.

Abstraction: Abstraction means details corresponding to the insides of component (its implementation like in case of interface specification) are abstracted away or hidden.

Meanwhile, the interface provides us with tools to achieve required tasks by the user using said interface for the task it was created for. For ex: the nav2 package can be used to navigate the robot. We need not concern ourselves with the implementation details of algorithms used but only concern ourselves with including said package, change parameters like map refresh rate etc.

Asynchrony: This idea ensures that communication between components occurs in an asynchronous manner. The consequence of this idea leads to the application to work across multiple time frames/domains.

Modularity: Modularity is a UNIX philosophy. It bases itself on dividing work between modules. Now, these modules are assigned one specific task which it must fulfill to its fullest.

2.2.4.2 Design Requirements

Like the design principles stated in the previous section, ROS2 also has designated design requirements. These include security, diverse networks, real-time computing etc.

Security: Security was one of the major concerns in ROS1. ROS2 solves this issue by including features to prevent misuse (be it accidental or malicious). To add up to such measures, ROS2 comprises of authentication, encryption, and access control features.

Diverse networks: This requirement enables networks to occur via Wi-Fi, LAN etc. Along with inter communication, networking occurs within the system - intra - as well. To facilitate such a diverse array of communication networks, ROS2 guarantees to provide quality service for seamless data transfers.

Real-time computing: For concerns over safety and performance, calculations need to occur in a deterministic time frame. To aid in such critical situations, APIs are provided to developers by ROS2 system.

2.2.4.3 Communication Patterns

ROS2 provides access to several communication patterns. These are topics, services and actions which are organized under node.

a) Topics

They are used for one-way communication. They are an asynchronous message passing framework which is the publisher subscriber functionality. The publisher is the node that sends the data to the topic. Meanwhile, the subscriber is another node which if subscribed to the previously mentioned topic will have access to the data being sent by the publisher. Its architecture allows many-to-many communication.

b) Services

However, ROS2 also provides a request-response style of communication pattern, called a service. Here, a request will be sent to the service server by a service client over a service. These are used for a short range/instant of time and not continuously like topics.

c) Action

These are goal-oriented and asynchronous communication interface. It is comprised of requests, responses, and periodic feedback. Unlike the above two, action can be cancelled and is useful for long running tasks where continuous feedback relating to the progress of the task is helpful.

Now the communication patterns listed above are organized under an entity called Node which serves a singular, modular purpose. They can send or receive data via communication patterns like topics, services, actions etc. In ROS2, nodes comprise of various publishers, subscribers, services, action client and servers which can be viewed via the command line. Also, one can create their own custom node in ROS2 by utilizing client libraries like rclcpp or rclpy.

2.2.5 ROS2 Control

In ROS2, the framework used for controlling the robots and interfacing with hardware components is ros2_control. This framework consists of ros2_control, ros2_controllers, control_toolbox , realtime_tools and control_msgs GitHub repositories which assist in incorporating new hardware in the robots.

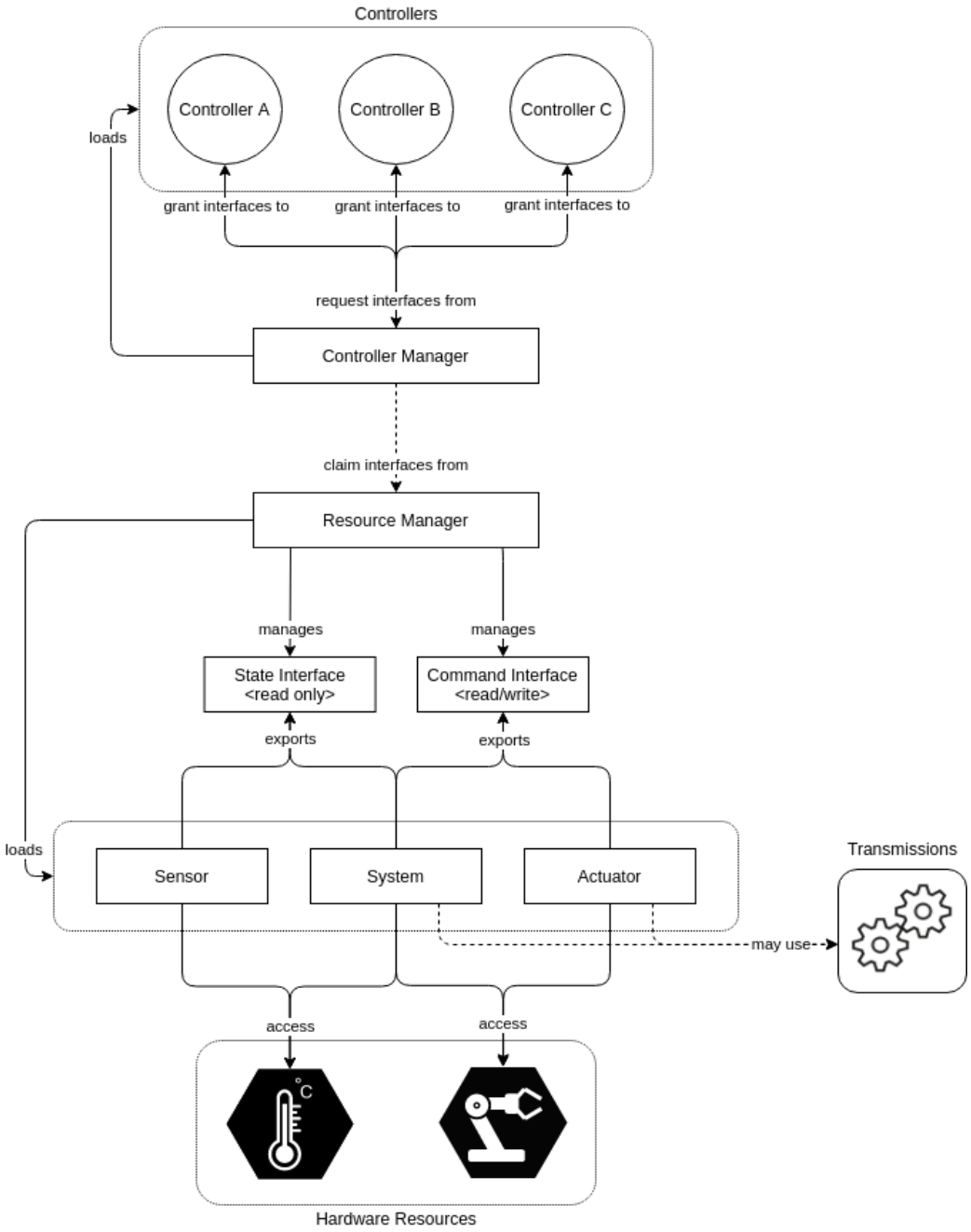


Figure 2.3: Architecture of ROS2 Control [18]

The framework consists of three main components as evidenced by figure 2.3.

i) Controller Manager (CM)

The controller manager (CM) acts as a bridge between the controllers and the hardware as shown in figure 2.3. It manages the controllers by either loading, activating, deactivating, or unloading them and the interface they need. It also grants controllers access to the hardware (through resource manager) or reports errors if it cannot do so. It also reads data from the hardware components, updates outputs of all the active controllers and transfers to the components (in a control loop) using the update () method. It employs a plugin system, meaning each component isn't a standalone executable. Instead, they are libraries loaded at runtime, containing functions that integrate into the system.

ii) Hardware Interfaces (State Interfaces and Command Interfaces)

Different types of hardware require different methods of control. For example, one robot may use serial communication for its motors, while another uses CAN bus. Additionally, they may offer different control capabilities, such as speed, position, or torque control. Each hardware setup requires a specific hardware interface, which serves as a bridge between the hardware and the ros2_control framework, standardizing how the hardware is accessed. This interface abstracts the hardware complexities, presenting them through command and state interfaces. Command interfaces allow us to control the hardware, while state interfaces provide monitoring capabilities.

iii) Resource Manager

The Resource Manager (RM) in ros2_control abstracts hardware and its drivers, known as hardware components, using the pluginlib library. It manages their lifecycle and interfaces, enabling reuse of components without reimplementing them. During the control loop, RM's read() and write() methods handle communication with hardware components.

The resource manager manages multiple hardware interfaces, each with multiple command and state interfaces, presenting them as a unified list of command and state interfaces to controllers. The resource manager accesses hardware interface information

from the URDF (Unified Robot Description Format), closely tied to the robot's hardware design using a <ros2_control> tag.

iv) Controllers

Controllers in the ros2_control framework, based on control theory, compare reference values with measured outputs to calculate system inputs. These controllers, derived from the ControllerInterface in the controller_interface package, are exported as plugins using the pluginlib library. During the control loop execution, the update() method accesses the latest hardware state to enable writing to hardware command interfaces.

Interacting with the ROS ecosystem, controllers receive control input from ROS topics and utilize algorithms to determine appropriate actuator commands. They can also publish feedback or state information on ROS topics. The controller manager matches loaded controllers with corresponding command and state interfaces provided by the resource manager, configured via a YAML file. Multiple controllers can exist within one robot, if they do not conflict over resources, though they can share read-only state interfaces.

v) Hardware Components

Hardware components in ros2_control enable communication with physical hardware and represent its abstraction within the framework. These components must be exported as plugins using the pluginlib library and are dynamically managed by the Resource Manager.

There are three types of components:

1. System: Designed for complex, multi-degree-of-freedom (DOF) robotic hardware.
2. Sensor: Used for sensing the robot's environment.
3. Actuator: Geared towards simple, single-degree-of-freedom (DOF) robotic hardware.

2.2.6 Joystick in ROS

Remote control, or teleoperation, is crucial during the initial stages of robot development. In ROS, the teleop_twist_keyboard node facilitates remote control by publishing twist messages on the /cmd_vel topic based on keyboard input. However, this method is less intuitive and requires an active terminal. A more comfortable alternative is using a joystick, also known as a gamepad or game controller. Joysticks offer a more intuitive control experience and allow for the integration of multiple robot commands to various joystick buttons, enhancing control capabilities.

To check if a joystick works in a Linux system, evtest can be installed using the command *sudo apt install joystick evtest.* Then with joystick connected to the Linux system and it can be started it in the terminal with the command *evtest*. This tool prompts the user to enter the event number and displays button presses. This test should be performed for each gamepad to ensure compatibility with the Linux system before using it in ROS.

ROS provides an inbuilt joy package, which communicates with the joystick through joy_node and publishes button presses as sensor_msgs/joy messages on the /joy topic. The messages contained in sensor_msgs/joy are timestamp-the time at which the data is received, axes measurement from a joystick and the buttons measurements form the joystick. These messages when combined are also called joystick state. These messages can be used for remote control by converting them to twist messages. For this purpose, the data coming through on /joy topic can be passed to the teleop_node located in the teleop_twist_joy package.

2.2.7 tf2

This is a subsystem of ROS2 that is used for geometric transformation. The tf2 library is used to create coordinate frames or reference axes and to define the geometric relation between those frames even when those frames are constantly changing. The tf2 keeps track of all the coordinate frames created and lets the user know the current position of

the specific frame in the map frame and even the position of that frame few seconds ago with respect to some another frame.

There are two major tasks done via tf2 in most of the robots: listening for the transforms between all coordinate frames and broadcasting relative pose of the coordinate frames to rest of the system. For this purpose, tf2 broadcaster is used to publish the coordinate frame and tf2 listener to compute differences in coordinate frames. tf2 tools provide the frames visualization tool 'view_frames' to create diagram of the frame being broadcasted by tf2 over ROS2. Similarly. tf2_echo can be used to determine the transform between the source frame and the target frame broadcast over ROSs using the command shown below:

ros2 run tf2_ros tf2_echo [source_frame] [target_frame]

The tf2 package of ROS2 also includes *static_transform_publisher* that can be used to publish the static pose of robot to tf2 which is essential in navigation. This static_transform_publisher is an executable which can be used as a command line tool or node to publish a static coordinate frame to tf2 using x, y, z offset in meters and roll, pitch, and yaw in radians. Here x, y, and z are the translation in the respective coordinates and roll, pitch and yaw are rotation about the x, y and z axes respectively. This is essential in determining let's say the pose of the camera frame with respect to the world frame and publishing it to tf2. The example of command for publishing the camera frame to tf2 is given below:

ros2 run tf2_ros static_transform_publisher --x x --y y --z z --yaw yaw --pitch pitch --roll roll --frame-id world_frame --child-frame-id camera_frame

This can also be done by using quaternions instead of roll, pitch, and yaw.

ros2 run tf2_ros static_transform_publisher --x x --y y --z z --qx qx --qy qy --qz qz --qw qw --frame-id world_frame --child-frame-id camera_frame

The concept of converting quaternion to Euler angles is essential in publishing the current orientation of different frames of the robot.

Quaternion

Quaternion is a 4-dimensional number system representation of orientation which is more accurate than the rotation matrix in describing and computing the orientation and rotation in three-dimensional space. It can also be called unit quaternion since the square of sum of all four elements of a quaternion is 1. It is computationally more efficient than other methods such as rotation matrices in describing the 3D rotation and it also avoids a lot of the numerical errors that arise in these other methods for example problem termed gimbal lock that arises when two axes align.

A quaternion can be represented as

$$q = q_x i + q_y j + q_z k + q_w \tag{2.5}$$

Here, q_x, q_y, q_z and q_w are real numbers. The q_x, q_y, and q_z forms the imaginary part of the quaternion whereas the q_w represents the real part of the quaternion. The quaternion elements can be related to the rotation around a given axis as follows.

$$q_x = \sin(\alpha/2)\cos(\beta_x) \tag{2.6}$$

$$q_y = \sin(\alpha/2)\cos(\beta_y) \tag{2.7}$$

$$q_z = \sin(\alpha/2)\cos(\beta_z) \tag{2.8}$$

$$q_w = \cos(\alpha/2) \tag{2.9}$$

For the given roll, pitch and yaw angle, ϕ, θ, and ψ respectively, the quaternion can be written below for the body undergoing roll, pitch, and yaw in sequence.

$$\begin{bmatrix} q_x \\ q_y \\ q_z \\ q_w \end{bmatrix} = \begin{bmatrix} \sin\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) - \cos\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \\ \cos\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \\ \cos\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) - \sin\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) \\ \cos\left(\frac{\phi}{2}\right)\cos\left(\frac{\theta}{2}\right)\cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right)\sin\left(\frac{\theta}{2}\right)\sin\left(\frac{\psi}{2}\right) \end{bmatrix} \tag{2.10}$$

Here α is the rotation angle and $\cos(\beta_x)$, $\cos(\beta_y)$, and $\cos(\beta_z)$ are the direction cosines angles between the three coordinate axes and the axis of rotation.

The Euler angles can be obtained from the rotation matrix. The rotation matrix is defined for the rotations about the three principal axes as given below:

A rotation of ϕ radians about x-axis also called roll is defined as

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \quad 2.11$$

Similarly, pitch of θ radians about the y-axis is defined as

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \quad 2.12$$

Finally, the yaw of ψ radians about the z-axis is defined as

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad 2.13$$

Then, the rotation matrix for the given body undergoing rotation first about the x-axis, then the y-axis and finally the z-axis can be represented as the matrix product that defines the said sequence of rotations as:

$$R = R_z(\psi).R_y(\theta).R_x(\phi) \quad 2.14$$

$$= \begin{bmatrix} \cos(\theta)\cos(\psi) & \sin(\phi)\sin(\theta) - \cos(\phi)\sin(\psi) & \cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi) \\ \cos(\theta)\sin(\psi) & \sin(\phi)\sin(\theta)\sin(\phi) + \cos(\phi)\cos(\psi) & \cos(\phi)\sin(\theta)\sin(\psi) - \sin(\phi)\cos(\psi) \\ -\sin(\theta) & \sin(\phi)\cos(\theta) & \cos(\phi)\cos(\theta) \end{bmatrix} \quad 2.15$$

Hence, the Euler angles ϕ, θ, and ψ can be computed given the rotation matrix by equating nine elements obtained in the equation 2.15 with the corresponding elements of R.

2.2.8 Simultaneous Localization and Mapping (SLAM)

SLAM is a method which is employed by autonomous vehicles, robots etc. to simultaneously build the map and localize itself in the said map by itself. This is accomplished by usage of sensors like LIDAR or camera and devices like IMU or odometry.

2.2.8.1 Working

To understand the workings of SLAM, it can be broken down into two processes: front-end and back-end. The front-end comprises of data collection while the back end is responsible for the computation.

For data collection, either VisualSLAM or Lidar SLAM can be used. Visual SLAM makes use of cameras like simple cameras, RGB-D cameras etc. The usage of cameras is a cheaper option. Additionally, they provide a large volume of information which can be used to detect landmarks (previously measured positions). Landmark detection can also be combined with graph-based optimization, achieving flexibility in SLAM implementation.

Lidar SLAM, on the other hand, uses Lidar sensor which can be either 2D or 3D. They are more precise and accurate in comparison to Visual SLAM. The Lidar output is in the form of point clouds providing high-precision distance measurements and works very effectively for map construction with SLAM.

For the backend part, Visual SLAM and Lidar SLAM vary in the algorithms they make use of. In the case of Visual SLAM algorithms, it can be broadly classified into two categories namely Sparse methods and Dense methods. Sparse method matches feature points of images and use algorithms such as PTAM and ORB-SLAM. However, Dense methods use the overall brightness of images and use algorithms like DSO, DTAM, LSD-SLAM, etc.

LiDAR point cloud matching generally requires high processing power and thus is necessary to optimize the processes to improve speed. Due to these challenges, localization for autonomous vehicles may involve fusing other measurement results such as wheel odometry, global navigation satellite system (GNSS), and IMU data. Algorithms like gmapping, hector-SLAM, cartographer etc. are available and are well integrated in ROS2 as well. The algorithms can be classified into categories like particle based and graph based.

2.2.8.2 GMapping

This algorithm employs the particle filter technique for model-based estimation. In this technique a grid-based representation of the environment is created where the map is divided into small cells ultimately forming a grid. Now, a probability distribution over the occupancy of each cell in the grid is maintained, indicating whether the cell is occupied or free.

Gmapping employs motion model to estimate the robot's position and orientation on the basis of its control inputs which is its linear and angular velocity. This model predicts the robot's position in the next step. Next, sensor measurements are incorporated either from a laser rangefinder or a depth camera, to determine whether a particular section of the environment is occupied or not. These measurements are crucial to get information about obstacles and free space.

Now a particle filter approach is employed to represent the robot's pose uncertainty. It maintains a set of particles, where each particle represents a possible location of the robot. Initially, the particles are spread across the map uniformly to represent the idea that the robot is equally probable to exist in such locations. The algorithm then updates the particles based on the motion model and sensor measurements as stated above. After the updates, the particles are assigned weights, whereby particles with poses consistent with the sensor data are assigned a higher weight value. In contrast, inconsistent particles are assigned a lower weight value.

Having performed the weight distribution, a resampling step is conducted to select a subset of particles with higher weights while discarding particles with lower weights. This implies focus to be put on the most likely poses of the robot based on the updated information. At the same time, the map also gets updated with the probabilities of occupancy derived from the sensor measurements. The above-mentioned steps are iterative in nature, where gathering more observations results in better localization of the robot and accurate creation of a map.

2.2.8.3 SLAM-toolbox

It is also a SLAM algorithm which is a Graph based technique. It uses the concept of nodes and edges, with optimization techniques often used to refine the estimates and find the most likely trajectory and map given the sensor measurements. The nodes typically represent the robots pose at different time-steps or environment. Meanwhile, the edges represent a constraint between the poses. Such constraints contain the relationships between poses or landmarks based on sensor measurements (odometry, visual information, laser scans etc.). This sub-section briefly highlights the working of graph-based SLAM algorithm at a high level.

During the graph initialization phase, the graph is empty. A single node will be added to represent the starting pose of the robot. Now, data gathering via sensor occurs where sensor measurements are associated with graph nodes. This determines which measurements correspond to which poses or landmarks in the graph.

As new poses or landmarks are observed, new nodes will be added to the graph. The edges that connect the nodes represent constraints based on the sensor measurements. These constraints are derived from odometry data (motion constraints) or sensor observations (loop closure constraints).

Optimization techniques, such as least squares or nonlinear optimization, are used to refine our pose and landmark estimates. This refinement enables us to find the most likely configuration of poses and landmarks that satisfies the constraints. This step thus adjusts the poses and landmarks to minimize the difference between the observed measurements and the predicted measurements based on the estimated poses and map.

Once the optimization step is complete, the estimated map is updated based on the refined poses and landmarks. The above-mentioned steps continue as the robot moves and collects new sensor data due to which new nodes and edges are added to the graph. Again, the optimization is carried out iteratively to improve the estimates.

The slam-toolbox SLAM algorithm is available in ROS2, default, as part of the nav2 package. The effectiveness of efficiently building large maps reliably and easily has been emphasized in the corresponding slam-toolbox paper. [16]

2.2.9 NAV-2

Nav2 is the successor of the ROS Navigation Stack used in deployment of Autonomous Vehicles. This package allows mobile robots to navigate through complex environments to complete user-defined application tasks. Not only can it move from Point A to Point B, but it can have intermediary poses, and represent other types of tasks like object following, complete coverage navigation, and more.

It provides perception, planning, control, localization, visualization, and much more to build highly reliable autonomous systems. Getting sensor data using LIDAR, camera like sensors, this will compute an environmental model, dynamically path plan, compute velocities for motors, avoid obstacles, and structure higher-level robot behaviors.

Planners: The task of a planner is to compute a path to complete a stated objective. The path can also be known as a route, depending on the nomenclature and algorithm selected. Two canonical examples of planners are computing a plan to a goal (e.g. from current position to a goal) or complete coverage (e.g. plan to cover all free space). The main task of writing a planner is to: compute the shortest path, compute complete coverage path etc. Planners can be classified into two types, mainly global and local planners.

Global Planner: The global planner is responsible for generating a general shortest path from the robot's current position to its goal position within the environment, as stated by goal pose. It typically operates on a map of the environment and considers obstacles, terrain, and other relevant factors to find an optimal or near-optimal path. The output of the global planner is a trajectory or a series of waypoints guiding the robot from its starting point to the destination. To aid this goal, global planners often utilize algorithms such as A*, Dijkstra's algorithm etc. for path planning.

Local Planner: Meanwhile, the local planner is responsible for generating a low-level trajectory that guides the robot through immediate obstacles along the path generated by the global planner. To this extent, it accounts for real-time sensor information and dynamic obstacles. It operates near the robot, reacting to environmental changes and adjusting its trajectory to avoid collisions and navigate through tight spaces.

Cost-Maps: A costmap is a regular 2D grid of cells containing a cost for unknown, free, occupied, or inflated cells. Basically, numerical representation is used to specify these cells. If looked at in Rviz, occupied regions are marked in a darker color than unoccupied ones. This costmap aids to compute a global plan or sampled to compute local control efforts.

Information from LIDAR, RADAR, sonar, depth images are important to buffer information into the costmap via layers. It is also recommended to process sensor data before inputting it into the costmap layer.

Its other application lies in the fact that it can be used to detect and track obstacles in the scene for collision avoidance using camera or depth sensors. Additionally, layers can be created to algorithmically change the underlying costmap based on some rule or heuristic.

NAV2 Commander API: The Commander API was used in this project to aid in navigating towards a goal. This goal can be supplied manually or through a topic, in the form of a goal-pose. In this API, ROS 2 and Action Server tasks are provided to help focus on building an application leveraging the capabilities of Nav2. There are various methods accompanying this API to fully utilize its capabilities. Some methods are: goToPose(), getResult(), cancelTask(),spin() etc.

- goToPose(goalpose): This method requires a ‘goalPose’ to be specified. ‘goalPose’ represents the position and orientation of robot required.
- getResult(): This method gives us the result of the navigation. IT can be success, failure or canceled.
- cancelTask(): It helps to cancel an ongoing task like ongoing navigation in case of anything that went wrong.

- spin(angle, time): “spin” helps to spin the robot through a specific angle. The angle is in radians.

2.2.10 OpenCV

OpenCV stands for Open-Source Computer Vision library developed by Intel that is written in C++ and has C++, Python and MATLAB and java interfaces. OpenCV plays a very important role in real-time operation of autonomous mobile robots. It can be used to process images and videos to identify objects; in this case ArUco tags were used for manipulating the payload. By integrating OpenCV with python libraries such as NumPy, the image pattern and its various features such as its coordinates and distance with respect to the camera can be obtained. [19]

OpenCV is also compatible with ROS. By using cv_bridge package, the image captured using camera and ROS camera module can be converted back into a form usable in OpenCV. Then, required image processing, pose-estimation can be performed.

2.2.11 Pose Estimation

Pose estimation is a computer vision technique that predicts and tracks the location of a person or object. This is done by looking at a combination of the position and the orientation of a given person/object.

Pose estimation can be distinguished into 2D pose estimation and estimates the location of key points in 2D space relative to an image or video frame. It estimates an (x, y) coordinate for each key point. Whereas 3D pose estimation transforms an object in a 2D image into a 3D object by adding a z-dimension to the prediction thus providing spatial positioning of that object.

Another line can be drawn in pose estimation with single pose and multi-pose estimation. Single pose estimation tracks and detects one person or object, while multi-pose detects and tracks multiple people or objects.

2.2.12 Fiducial Markers for Pose Estimation

Fiducial markers, in a broad sense, are objects employed within an image to establish a reference point or enable measurements. So, they are a common framework in object manipulation application tasks where markers are used for the identification, detection, and localization of different objects. Fiducial markers in their general form are objects used to provide a point of reference or a measurement in an image. These markers can include specific encoded IDs or messages. They were monochromatic mostly, but recently multicolored markers have also been introduced. They use highly distinguishable geometric patterns with strong visual characteristics to encode multiple bits of data. Barcodes and QR codes are the most well-known examples of such markers. Initially designed for Augmented Reality (AR) applications, recently these markers have been utilized in Unmanned Aircraft Systems (UAS) and robotics applications .

2.2.12.1 ARToolkit

ARToolKit, an open-source project that has gained significant popularity in the past decade, particularly within the academic community, utilizes markers composed of a wide black border enclosing an inner image. The inner image is stored in a database of valid patterns. Despite its widespread use, ARToolKit has certain limitations or disadvantages that should be noted. First, it uses a template matching approach to identify markers, obtaining high false positive and inter-marker confusion rates . Second, the system uses a fixed global threshold to detect squares, making it very sensitive to varying lighting conditions.

2.2.12.2 ARTag

ARTag is one of the most widely used packages, which is based on ARToolkit. By making the interior of the square a six-by-six-bit matrix, each marker is given a unique 36-bit long word as its ID .

2.2.12.3 ArUco

ArUco is another package based on ARTag and ARToolkit. ArUco's most significant contribution lies in its ability to enable users to create customizable libraries. Rather than including all available markers in a standard library, ArUco allows users to generate a personalized library tailored to their specific requirements. A system using a stationary ArUco marker on the ground that acts as a landing target is proposed in .

2.2.12.4 Pi-Tag

A fiducial marker with a square border that relies on round internal bits for identification is a type of marker that combines both square and round elements for its visual recognition. The square border provides a clear reference frame, while the internal round bits serve as unique identification patterns used for detection and tracking purposes.

CHAPTER 3: METHODOLOGY

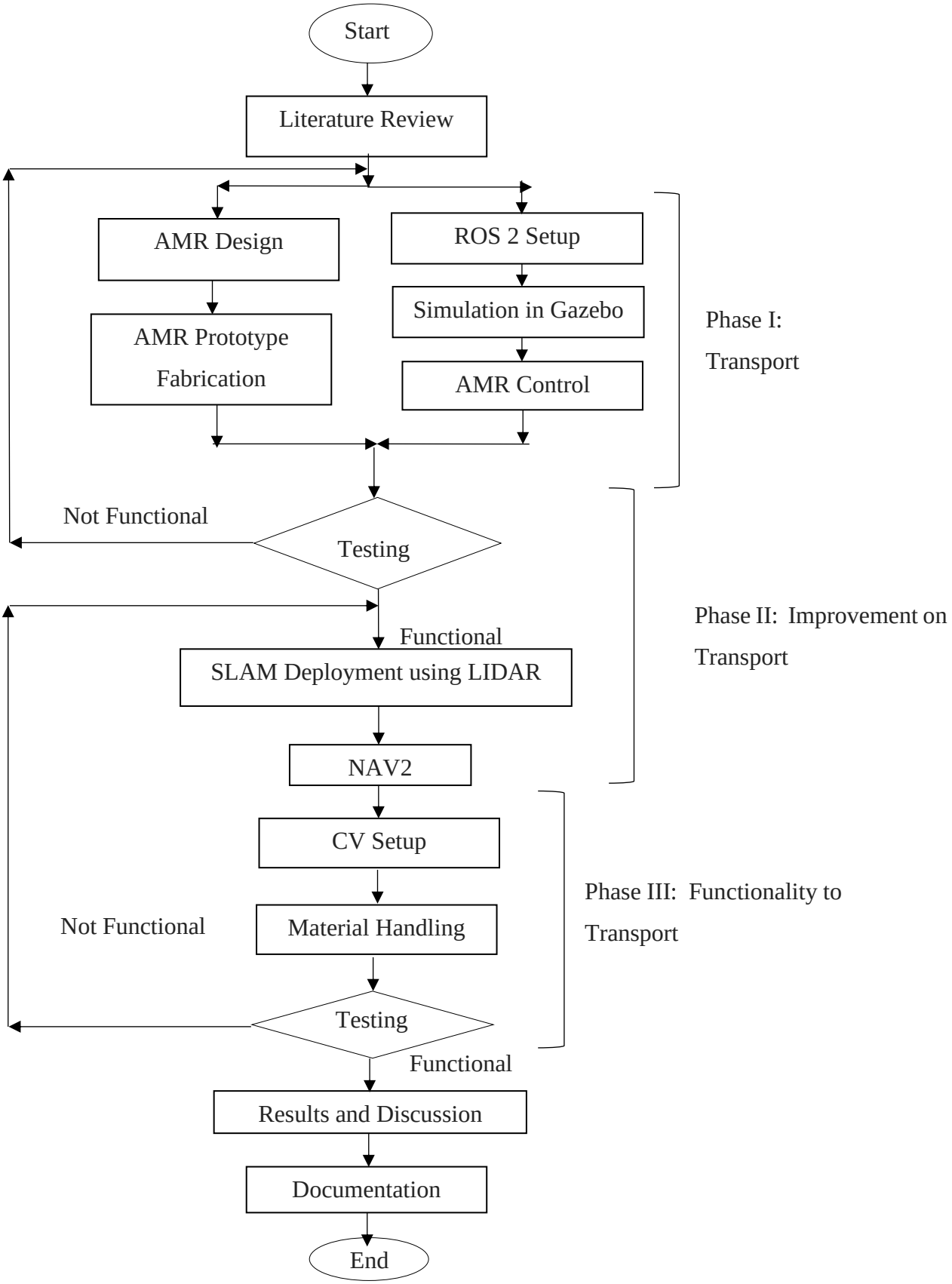


Figure 3.1: Methodology Flowchart

3.1 System Used (For Development/Implementation)

3.1.1 Hardware used

The following list of hardware represent the hardware are used in the project:

1. 2 Hub Motors: 24 volts 350 Watts (1:16) gear reduction electric motor
2. 2 Cytron Motor Driver
3. Arduino Mega
4. Raspberry Pi
5. Laptop
6. LIDAR
7. Battery 24V
8. Power Bank
9. Camera
10. Arduino Uno
11. Linear Actuators
12. Joystick
13. Emergency Switch
14. Telescopic channels

3.1.2 Software used

The major software that are used in the project are:

1. Solidworks
2. ROS 2
3. Visual Studio Code
4. Arduino Software IDE
5. Gazebo
6. RViz2
7. Creality Slicer
8. Kicad

3.2 Phase I: Transport

A CAD model of AMR's chassis was first designed and analyzed in Solidworks. Simultaneously, ROS2 humble was set up in the Linux operating system. After the completion of modelling of the vehicle, it was fabricated and the AMR control system was integrated into it, i.e., the integration of motor drivers, Arduino, raspberry pi, and joystick for conventional control. The upcoming sections further describes in detail the processes involved in phase I for the transport function of the AMR.

3.2.1 AMR Design

3.2.1.1 CAD

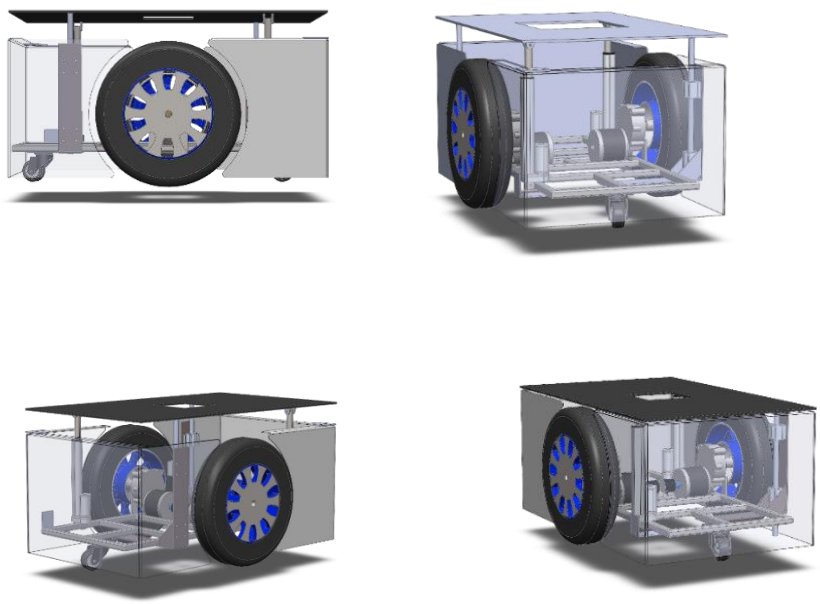


Figure 3.2: Transparent view of robot

The images above show the transparent view of the 3D model of the designed AMR. The two driving wheels are actively controlled to move the robot in various directions. Castor wheels are used to maintain balance.

The figure below shows the third angle projection of the AMR model with the major dimensions. The length of the robot is 900mm and its width is 622.80mm based on the design. The dimension of our AMR is constrained by the motor available to us.

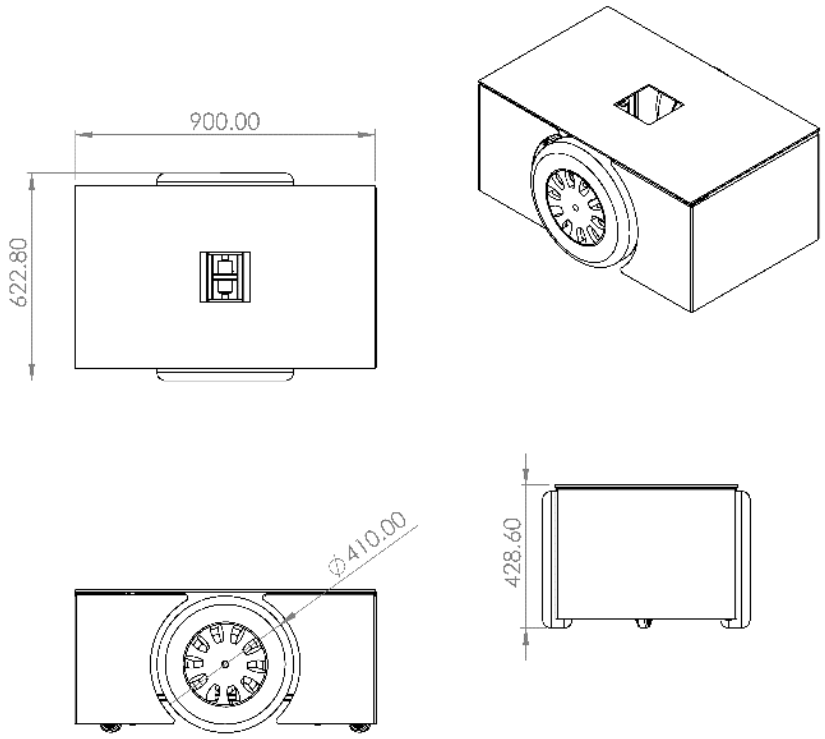


Figure 3.3: General dimension of robot shown in third angle projection

3.2.1.2 Static structural study of weldments

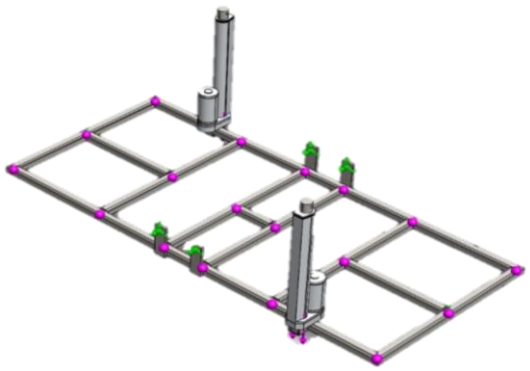


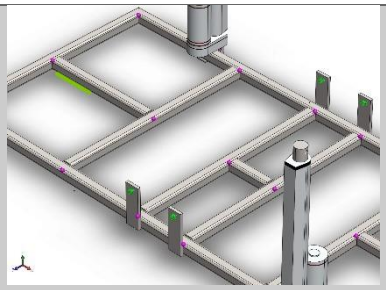
Figure 3.4: Model Reference

The isolated frame weldments of material properties as mentioned in Table 3.1 is checked against static structural analysis for von Mises Stresses, displacements and factory of safety.

Table 3.1: Material Properties

Name	Plain Carbon Steel
Model type	Linear Elastic Isotropic
Yield strength	2.20594e+08 N/m^2
Tensile strength	3.99826e+08 N/m^2
Elastic modulus	2.1e+11 N/m^2
Poisson's ratio	0.28
Mass density	7,800 kg/m^3
Shear modulus	7.9e+10 N/m^2

Table 3.2: Fixtures

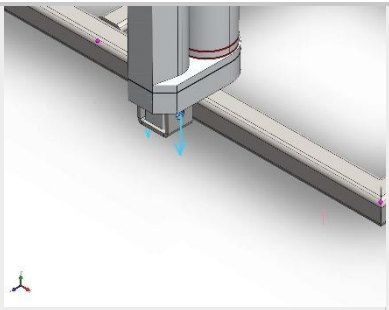
Fixtures	Fixture Image	Fixture Details
Fixed Hinge-1		**Entities:** **4 face(s)** **Type:** **Fixed Hinge**
Caster Support-1		**Entities:** **2 faces** **Type:** **Roller**

Resultant Forces on fixtures:

Components	X	Y	Z	Resultant
Reaction force(N)	**-6.53267e-05**	**810.243**	**-2.38419e-05**	**810.243**

Fixed supports listed in Table 3.2 are added on the faces of motor plate bolted to the motor, named Fixed Hinge-1, and the surface under the weldments where castor wheels are attached namely Caster Support-1. Loads of 500N each are applied on both attachments of linear actuators.

Table 3.2: Loads

Loads	Load Image	Load Details	
Force-1		**Entities:**	**2 face(s)**
		Values:	**---, ---, 500 N**
Force-2	**(On other actuator attachment)**	**Entities:**	**2 face(s)**
		Values:	**---, ---, -,500 N**

3.2.1.3 Results

Static structural simulation results show the frame deflecting under load, with maximum displacement of 0.2993 millimeters indicating potential bending on the extreme opposite corners of the frame.

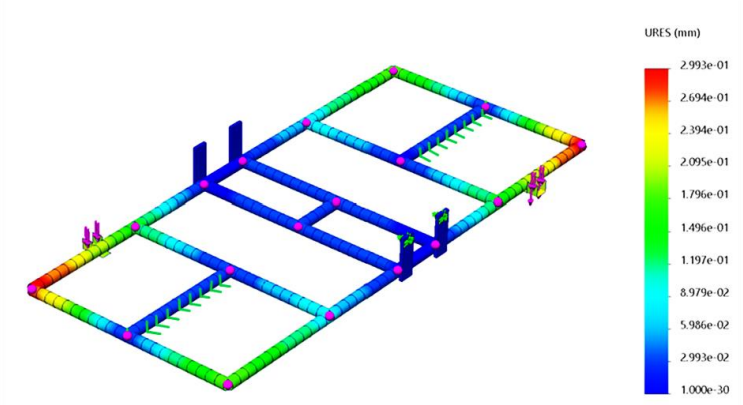


Figure 3.5: Displacement plot

The maximum von Mises stress of 115.4 MPa was found to be at the weldment of motor-plate. Max stress was found near the fixture, which is as expected. The maximum

stress is well under the yield point of the material. The minimum Factor of Safety is 1.9, which is acceptable for our use cases.

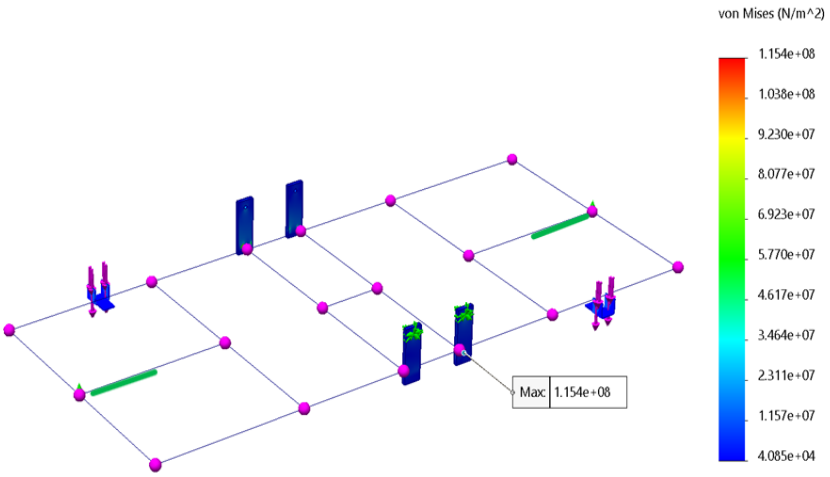


Figure 3.6: Stress plot

By varying increasing the number of elements, there was convergence of the displacement plot for 0.2993mm and axial stress for 150 kPa.

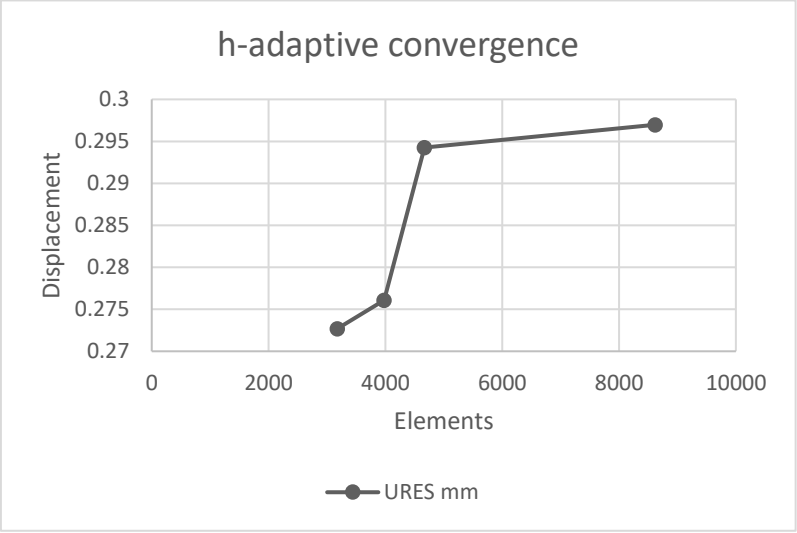


Figure 3.7: Adaptive convergence graph of displacement

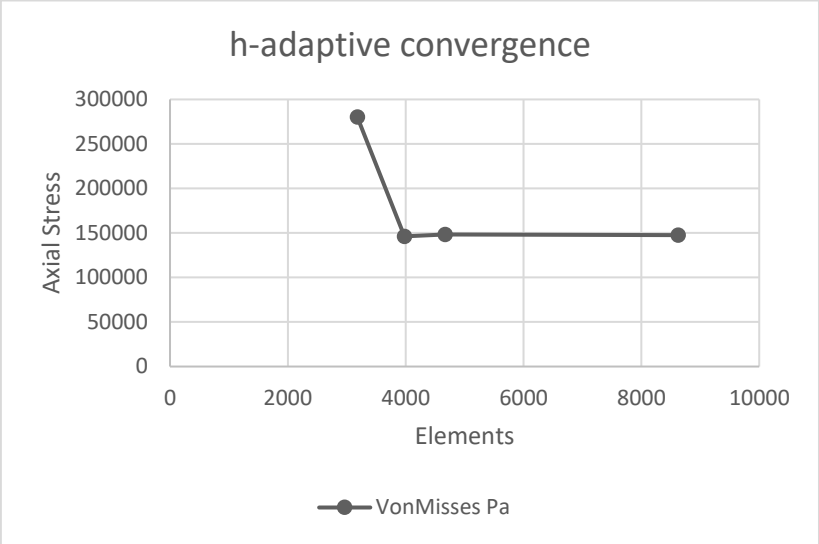


Figure 3.8: Adaptive convergence graph of axial stress

3.2.2 AMR Chassis fabrication

3.2.2.1 Frame Fabrication

First, the frame was designed in SolidWorks using weldment features. MS square tube of dimension 19.30x19.30x2.00 is used for the frame. The top view of the designed frame is shown in the figure below.

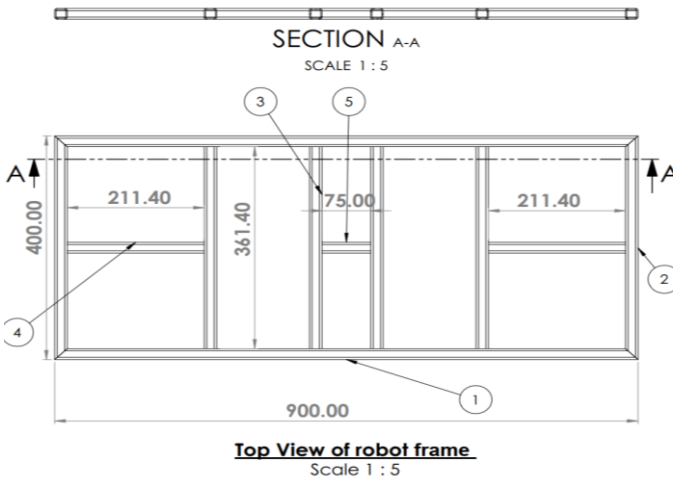


Figure 3.9: Top view of the robot frame

Table 3.3: Cut List of AMR Frame

ITEM NO.	QTY.	DESCRIPTION	LENGTH (mm)	ANGLE1	ANGLE2
1	2	TUBE, SQUARE 19.30 X 19.30 X 2.00	900	45^0	45^0
2	2	TUBE, SQUARE 19.30 X 19.30 X 2.00	400	45^0	45^0
3	4	TUBE, SQUARE 19.30 X 19.30 X 2.00	361.4	0	0
4	2	TUBE, SQUARE 19.30 X 19.30 X 2.00	211.4	0	0
5	1	TUBE, SQUARE 19.30 X 19.30 X 2.00	75	0	0

The cut list above was used to make the frame of the AMR. For the fabrication process, first a fixture to hold the tubes was created and then the above parts of the frame were welded accordingly.

3.2.2.2 Motor and Wheel Placement

After the fabrication of the frame was completed, the designed motor plates were fabricated. Then, these plates were welded to the frame. Four mild steel plates of 3 mm thickness were used to attach the hub motor with the frame. Another two mild steel plates were used to attach the caster wheel at the lengths ends. The hub motors and wheels are placed at the mid length position of the frame. The figure below shows the fabricated frame with the motor, wheels and caster wheels placed at their respective position.

Figure 3.10: AMR frame with motor and wheels placed

3.2.2.3 Encoder Placement

Initially, it was planned to use Orange OE-775 Hall Effect Two Channel Magnetic Encoder to obtain the encoder counts per revolution. So, to couple the encoder with the hub motor following parts were 3D printed. The orange hall encoder would sit inside the cylindrical holder with teeth while the circular holder with slots for the teeth would be attached at the ends of the hub motor as shown in Figure 3.12. The gears were 3D printed to offset the encoder position parallel to the axis of the hub motor.

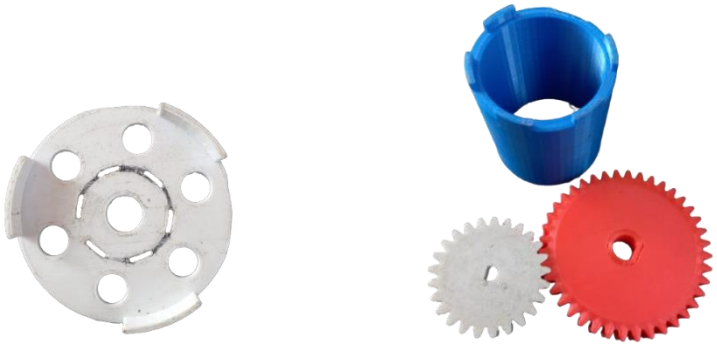


Figure 3.11: Encoder enclosure and couplings

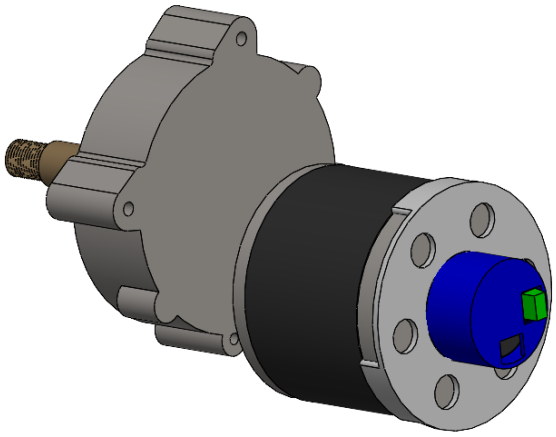


Figure 3.12: Coupling of Orange Hall encoder with hub motor

But the above model was discarded. The decision to switch from the orange hall encoder to more accurate rotary encoders was driven by concerns about accuracy and

the availability of two rotary encoders. To attach the rotary encoders to the hub motors, two L-shaped plates were fabricated and secured to the robot frame with nuts and bolts. One encoder was aligned directly with the motor shaft using a 3D printed coupler for synchronized movement. Due to space limitations caused by the narrow frame, the second encoder was positioned slightly off-center from the motor axis. This arrangement enabled successful integration of both encoders, improving accuracy without needing major frame modifications.

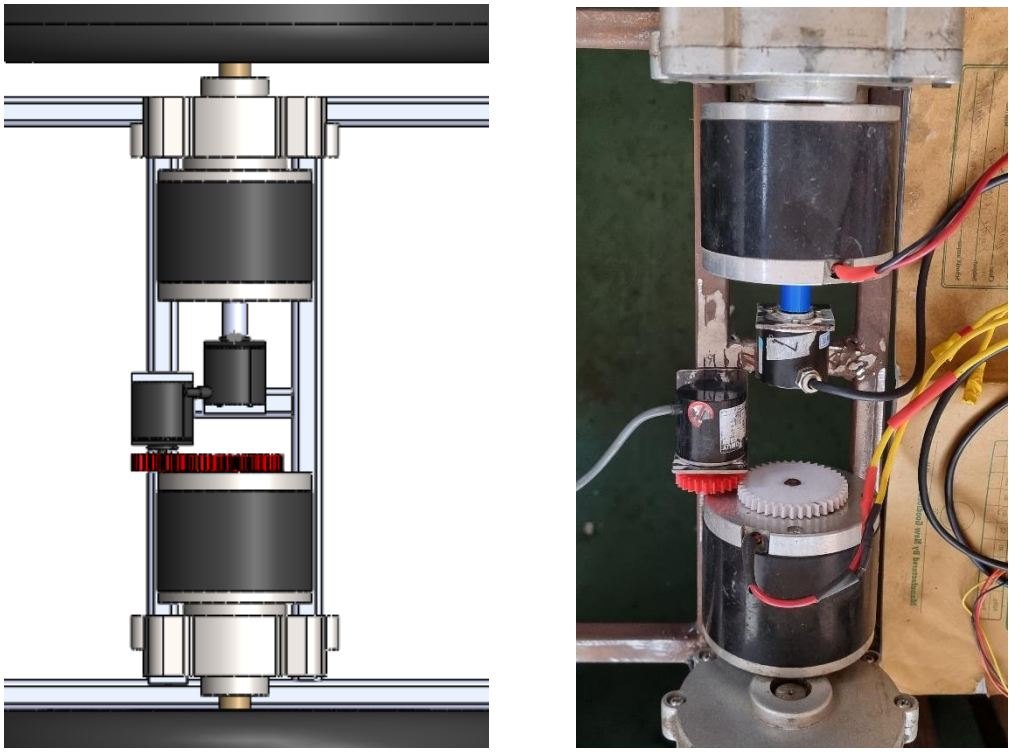


Figure 3.13: Position of rotary encoder in the AMR frame

3.2.3 ROS2 Setup

ROS2 Humble Hawksbill is the eighth release of ROS2 which is updated from its earlier version of ROS2 Galactic. ROS2 humble is supported on Windows 10 platform as well as Ubuntu 22.04 (Jammy). ROS2 humble was chosen as our robot operating system though new distributions of ROS2 have been released due to availability of considerable number of error free packages and community support on humble. Also, humble is a long-term support that will be supported till May 2027.

The following steps are followed for setting up the ROS 2 humble hawksbill in Ubuntu 22.04 Jammy:

3.2.3.1 Setting up the locale

locale # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale # verify settings

This step ensures that the local we have supports UTF-8, (Unicode Transformation Format- 8bits) which is needed for handling various symbols and languages used in ROS2 tools and packages. The *locale* displays the current locale settings including character encoding. *sudo apt update && sudo apt install locales* updates the package list and installs the locales package providing tools for managing locale setting. Then, the locale for US English with UTF encoding is generated and it is setup for the current session and for future logins. Then it is exported to various subprocesses in the current session. Finally, *locale* is run again to ensure that the required settings have been applied.

3.2.3.2 Setting sources

In this step Ubuntu Universe repository is enabled first to ensure that we have access to community maintained free and open-source applications. Then the ROS 2 apt repository is added to the source list. An apt (advanced package tool) repository contains software and packages. The ROS2 apt repository enables installing ROS packages easily on our workspace using apt commands.

3.2.3.3 Installing ROS2 humble packages

After setting up the locale and the sources, the apt repository caches are updated along with the Ubuntu systems. Then, the ROS2 is installed using the command *sudo apt*

install ros-humble-desktop. The development tools necessary for building the ROS packages is also installed by using the command *sudo apt install ros-dev-tools.*

3.2.3.4 Sourcing ROS2 humble

The environment is set up by sourcing the setup file using the command *source /opt/ros/humble/setup.bash*. This is the most important step in accessing the ROS2 package files before building and running them, and this must be sourced every time we want to use ROS packages. To ensure this, the setup command has to be included in the shell startup script which is *.bashrc* file so that whenever a new shell is opened, this setup file is sourced. This is done using the command *echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc*

3.2.3.5 Setting environment variables

After sourcing the ROS 2 setup files, the various environment variables required for the operation are set up as default. There are two main environment variables to be changed for interference less operation. The ROS_DOMAIN_ID variable is set as default to value 0. The nodes on this domain can send and receive messages to each other freely. If the nodes are on different domains, the communication between nodes is interrupted. So, to avoid the interference between different computers running ROS2 on the same network which might have domain id 0 by default, the ROS_DOMAIN_ID must be given a unique integer value. To do this, the command *export ROS_DOMAIN_ID=number* is used. Here, '*number*' is the domain id between 0 and 101 inclusive. This command is also included in the shell startup script.

Another important environment variable is ROS_LOCALHOST_ONLY which prevents the topics, services and actions used in each ROS system from being visible in other computers in the same network. The ROS_LOCALHOST_ONLY value is set to 1 for this purpose. This must be done as most of the setup is on campus, where many computers are connected to the same network at once. When a dedicated router is used to create a network, the code used to set the local host only can be commented in the shell startup script so that the other computers in the network would be able to see the topics, services, and actions.

3.2.3.6 Additional Tools

i) rqt

After setting up the ROS2 system, further packages required to visualize and run the AMR in ROS environment are also installed. 'rqt' is a GUI tool that makes it easy to understand the services, topics, and actions through the plugins available in the 'rqt'. One of the plugins is rqt_graph which is very useful in understanding what publisher is publishing messages to what subscriber node through which topic. Also, the command line tools such as *ros2 node list*, *ros2 topic list*, *ros2 service list, ros2 service find, ros2 topic echo <topic_name>* , *ros2 topic list -t*, *ros2 service list -t* etc. are used to look into the underlying action behind publishing and subscribing data to the specific nodes, only requesting for data when needed and providing the data when specifically called by the client using services. The following figure shows the use of rqt_graph to visualize the nodes and topics used while creating map using slam toolbox.

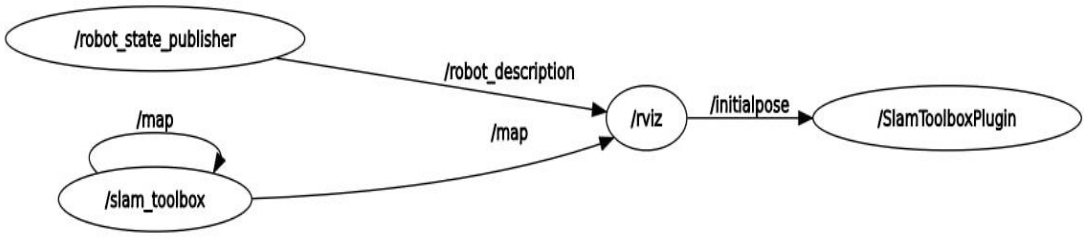


Figure 3.14 rqt_graph used to visualize nodes and topics in slamtoolbox

ii) colcon

It is quintessential for creating and building the ROS 2 workspace or the overlay required for the operation of the robot. This is installed using the command- *sudo apt install python3-colcon-common-extensions*. Both ament_cmake and ament_python build packages are supported by colcon. In this project ament_python build type is used mostly. The command colcon also supports command completion through the package colcon-argcomplete.

iii) SLAM toolbox

The Slam Toolbox package contains information from laser scanners in the form of a LaserScan message and TF transforms from odom->base link and creates a 2D map of a space. This package enables full serialization of the data and pose-graph of the SLAM

map to be reloaded to continue mapping, localize, merge, or otherwise manipulate. [16] For this purpose, slam toolbox is installed using the command *sudo apt install-ros-humble-slam-toolbox*.

iv) Nav2

Nav2 is used to move the robot from the robot's current position to goal pose by localizing the robot on the map and controlling the robot as it approaches the goal. For this Nav2 packages are installed using the command *sudo apt install ros-humble-navigation2 ros-humble-nav2-bringup*.

v) Gazebo

Gazebo simulates the motion of the robot model before building it. Gazebo incorporates ros control plugins which can be used to simulate the control of sensors, differential drive, and many more. For this, gazebo is installed using the command *sudo apt-get install ros-humble—ros-gz*.

vi) OpenCV

OpenCV is used for estimating the pose of the ArUco tag. These are binary tag fiducial markers which have their own unique IDs. First, it is necessary to calibrate the camera being used. This is done using ROS2 installing camera- calibration package. Next, a code is written in python and OpenCV to detect and thus estimate the pose of the ArUco tag. By embedding this code in ROS2 using cv_bridge, the pose can be transported as tf transform. The pose of ArUco tag is with respect to camera frame.

The additional tools used in the project are explained in the later sections when they are used.

3.2.4 Simulation of AMR in Gazebo and Rviz

Before writing the code and applying it to the physical hardware, a simulation was created to replicate the motion of the robot in the virtual world. This simulation was run on a software called Gazebo, which can run physics-based simulations. For this purpose, it was necessary to first create a robot model with sensors included. This model is also used to depict the AMR when using it in the real environment. To simulate the

actual sensors, we used the respective Gazebo plugins. The detailed implementation and path to simulating our AMR is laid out in the sections below.

3.2.4.1 Building a package

The project_amr package is created using build type ament_python using the command *ros2 pkg create –build-type ament_python –license Apache-2.0 project_amr* inside the src folder that is located in amr_ws folder. It is best practice to have packages within the src folder of our workspace. This creates a folder named project_amr inside the src folder which contains files such as package.xml, license, and folders such as project_amr and resource. In this workspace inside the project_amr folder inside the source folder, the required launch files, description files, configuration files, and the world file for gazebo simulation are included in the respective folder. The files created in these folders for the slam and navigation are explained in the next subsection. The package is built using the command *colcon build –symlink-install* from this src directory which creates folders: build, install and log. The argument *–symlink-install* is used so that it is not required to rebuild the package whenever the python codes are changed.

3.2.4.2 Creation of Robot model

First a robot model is created. A simplified version of a robot is designed based on shapes like cuboid, sphere, and cylinder for specific parts. This modeling can be done with the help of URDF and Xacro. These are XML-based file formats and extensions used to describe the kinematic and visual properties of a robot. It defines the robot's links, joints, sensors, and other relevant information. URDF is used to represent the robot's physical structure while Xacro is a way to include macros in URDF files. This allows for more modular and reusable robot descriptions.

The code for each part is broken into separate files rather than cramming it onto one. These include camera.xacro, gazebo_control.xacro, amr_robot_core.xacro, inertial_macros.xacro, amr_robot.urdf.xacro, lidar.xacro and ros2_control.xacro. The file ‘amr_robot_core’ contains essential details on the robot's shape, geometric and

mass properties. The other files, similarly, are self-explanatory. Meanwhile, “amr_robot.urdf.xacro” handles the part of combining every single file together.

Starting from “amr_robot_core”, at first it is necessary to define a coordinate axis. The origin is based off of “base_link”. From “base_link”, joints like “chassis_joint”, “wheel_joint” etc. are based off.

```
<link name="base_link">
</link>
```

Likewise, the “chassis_joint” is also based on “base_link” which is the parent link.

```
<joint name="chassis_joint" type="fixed">
    <parent link="base_link"/>
    <child link="chassis"/>
    <origin xyz="${-wheel_offset_x} 0 ${-wheel_offset_z}"/>
</joint>
```

In the code snippet, the origin of “chassis_joint” is offset by a certain ‘x’ and ‘z’ coordinate. This ensures that the “base_link” lies at the center and not the “chassis_joint”.

The chassis length, breadth, height, weight etc are defined using variables like follows:

```
<xacro:property name="chassis_length" value="0.906"/>
<xacro:property name="chassis_width" value="0.545"/>
<xacro:property name="chassis_height" value="0.320"/>
<xacro:property name="chassis_mass" value="9.0"/>
```

Using the above variables as references, a chassis having required dimensions as above is created. The geometry and visual sections define the visual aspect of the chassis. Meanwhile, the inertial section defines the mass, moment of inertia like aspects. This section is crucial for physics-based simulation in Gazebo.

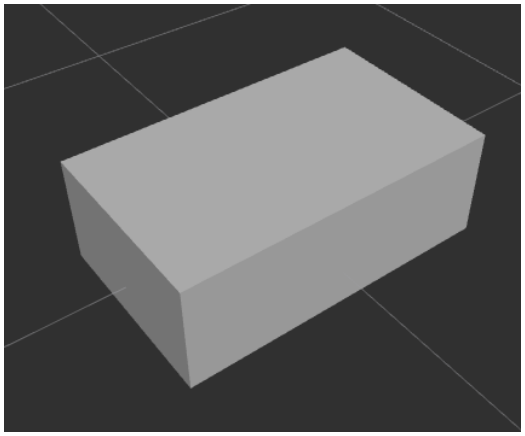


Figure 3.15: URDF model of chassis

Similarly, wheels are also added at the required positions. In this robot's case, the wheels are present along the middle of the chassis. The appendix has code for both the wheels. As can be seen in the appendix 1, the wheels are based off base_link. The wheel is offset along the 'y' axis by "wheel_offset_y" distance. The same was done for the left wheel as well, except for changing certain coordinate values.

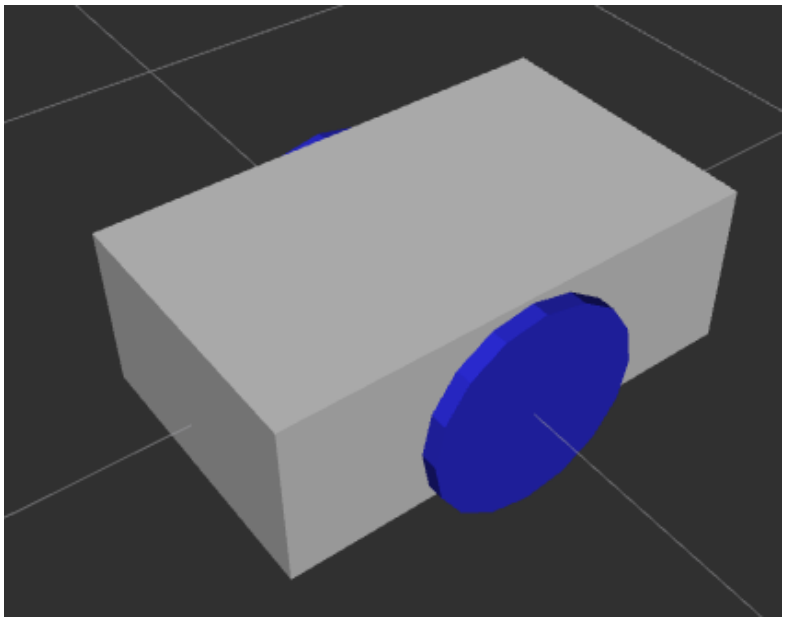


Figure 3.16: Chassis with wheels

The Robot model is a differential-drive one and hence caster wheels are also required. The physical design involved the use of two such wheels. In URDF, they were modeled using spherical shapes.

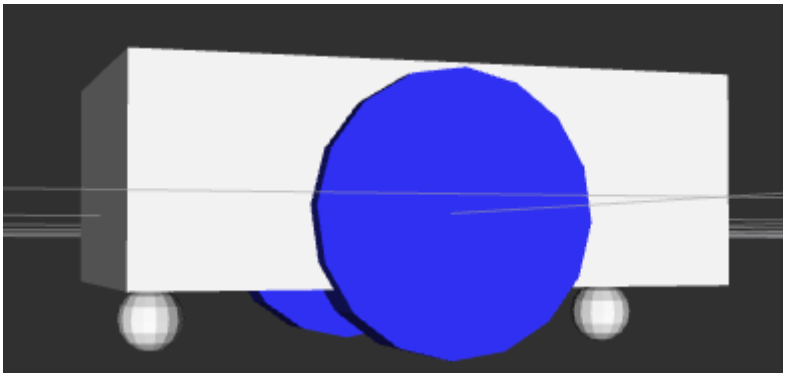


Figure 3.17: Chassis with caster wheels

In the code, the “mu values” is also entered to account for friction. The simulation would be unstable, and the movements erratic had these parameters been ignored. The caster_wheel radius is set equal to the gap present between chassis and the ground.

3.2.4.3 LIDAR and Camera

The LIDAR and Camera, URDF and xacro files are written separately as mentioned previously. Both are attached to the chassis_link and consist of cylindrical and cubical shapes. To be able to produce scan images or simply images, it is necessary to simulate these sensors like their real-world counterparts. This can be done via Gazebo plugins. The dimensions are specified in terms of geometry and mass. LIDAR is named laser_frame and has been linked to the chassis_link. The same is done for the camera as well.

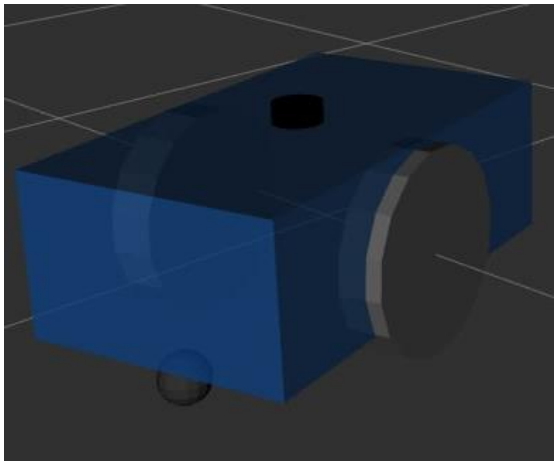


Figure 3.18: Chassis with camera and LiDAR

3.2.4.4 Gazebo Plugins

Although the geometric model of the robot was created, it needs to be operated. That is, the wheels need to rotate, the camera needs to display images and the lidar needs to send point cloud message. Because these need to be simulated as well in gazebo, Gazebo plugins are perfectly suited for this task. The plugins used are explained below:

i) Gazebo control

The below code snippet is a plugin which acts like wheel encoders in gazebo. Thus, using a ROS node like “teleop_twist_keyboard”, the wheels of robot model can be made to rotate, allowing it to move in a certain direction and speed. Information like wheel diameter and wheel separation are required. Likewise, values for maximum wheel torque and acceleration need to be input.

```
<!-- Wheel Information -->
        <wheel_separation>0.520</wheel_separation>
        <wheel_diameter>0.410</wheel_diameter>

        <max_wheel_torque>12</max_wheel_torque>
        <max_wheel_acceleration>3</max_wheel_acceleration>
```

ii) LIDAR plugin

To simulate the functionalities of a LIDAR, a gazebo plugin called lidar_controller is used. The data related to LIDAR is published to the /scan topic. Besides, info related to LIDAR range, angle is needed as well.

```
<gazebo reference="laser_frame">
        <update_rate>10</update_rate>
        <ray>
        <samples>360</samples>
              <min_angle>-3.14</min_angle>
              <max_angle>3.14</max_angle>
            <min>0.3</min>
```

```
            <max>8</max>
        </range>
        </ray>
```

As shown above, this plugin must reference to a LIDAR link, which in this case is the laser_frame. The laser_frame corresponds to the LIDAR part created for the robot model. This enables the gazebo lidar to work like a real one.

iii) Camera plugin

To simulate the physical camera, a plugin named “camera_controller" from file "libgazebo_ros_camera.so" is used. Unlike the LIDAR, the camera specification is based off default Gazebo values. This plugin also references the camera_link_optical frame off which the camera geometry is based.

3.2.4.5 ROS2 control in simulation

First dependencies for the ROS2_control: ros-humble-ros2-control, ros-humbe-ros2-controllers and ros-humble-gazebo-ros2-control were installed.
Next the URDF file was updated by creating a file named *ros2_control.xacro* and it was included in *robot.urdf.xacro.* A *<ros2_control>* tag (with details about hardware interface for controller manager), with name *GazeboSystem (type=” system”) and* a *<gazebo>* tag (for Gazebo to load necessary code) were added for simulation.

```
<ros2_control name="GazeboSystem" type="system">
  <hardware>
    <plugin>gazebo_ros2_control/GazeboSystem</plugin>
  </hardware>
  <joint name="left_wheel_joint">
    <command_interface name="velocity">
      <param name="min">-10</param>
      <param name="max">10</param>
    </command_interface>
    <state_interface name="position" />
    <state_interface name="velocity" />
```

```
    </joint>
    <joint name="right_wheel_joint">
        ……….
    </joint>
</ros2_control>
```

The <*gazebo*> tag included a <*plugin*> *with* filename"*libgazebo_ros2_control.so"*. The libgazebo_ros2_control.so plugin in Gazebo serves multiple purposes. It sets up communication with the hardware interface, manages the controller manager for control tasks, and utilizes the URDF provided by robot_state_publisher for consistency in the simulation. It also contains a <*parameter*> tag with the YAML configuration file which was created in our *config* directory and named *my_controllers.yaml*. Within the file parameters for the controller manager were added which included the *update_rate* of 30 for the rate at which the controller manager publishes, *use_sim_time* set to true for Gazebo simulation time and *diff_cont* for differential drive controller and *joint_broad* for join state broadcasting. Additional parameters required for the *diff_cont* and *joint_broad* was also added.

```
controller_manager:
  ros__parameters:
    update_rate: 30
    use_sim_time: true
    diff_cont:
      type: diff_drive_controller/DiffDriveController
    joint_broad:
      type: joint_state_broadcaster/JointStateBroadcaster
diff_cont:
  ros__parameters:
……..
# joint_broad:
#   ros__parameters:
……….
```

To launch the controllers, we used the spawner through the command line as:

ros2 run controller_manager spawner.py diff_cont

ros2 run controller_manager spawner.py joint_broad

Later our launch file was updated (including the controllers as ROS2 nodes) to automatically launch these controllers along with the rest of the code.

To drive the robot in Gazebosimulation using the *teleop_twist_keyboard*, we mapped the */cmd_vel topic* to */diff_cont/cmd_vel_unstamped* as the new controller listens to the later topic unlike the previously used Gazebo diff drive plugin.

3.2.4.6 Building and launching the simulation file

The package containing the above-mentioned files needs to be built using colcon. This allows for running or launching the respective ROS nodes. Various nodes like nodes for URDF, Gazebo, RVIZ, teleop etc. need to be run from the Linux terminal. As SLAM and Navigation are also done, their respective nodes need to be started as well. However, opening more than six or seven terminals and closing them is unfeasible as mistakes are common to be encountered in the testing phase. To avoid such hassle, launch files are available. These files are python files and contain information about the nodes that we require.

colcon build –symlink-install
source install/setup.bash
ros2 launch project_amr simulation_launcher.py world:= src/amr_robot/ worlds/ factory.world

This launches a gazebo instance where the robot can be visualized. The world tag placed above command loads a custom-made factory world, with obstacles around. To operate the robot model in the gazebo environment, a separate command to run teleop node must be started.

ros2 run teleop_twist teleop_twist_keyboard –ros-args /cmd/vel:=/diff_cont/ cmd_vel_unstamped

A terminal-based controller emerges which is used to drive the gazebo simulation model. The RVIZ tool can help to check whether the LIDAR, camera etc do their expected job.

rviz –d src/amr_robot/config/view.rviz

The above command opens up a rviz window. There the plugins needed for visualizing camera and LIDAR data can be added. LIDAR data is something akin to a series of red dots lining across the obstacle surface. Camera, on the other hand, displays the image of object ahead of the robot. Moving the robot around one can see how the LIDAR data and camera image keeps on changing with the surroundings. After the above steps, it is time for SLAM and Navigation related simulation.

3.2.4.7 SLAM and Navigation

First off, nav2 package along with slam_toolbox needs to be installed before moving with this part. Second, the parameter files concerning both these packages, which are of .yaml extension, need to be placed in config folder. The parameter file was based off nav2 tutorials.

Using SLAM toolbox plugin, a map of the surroundings is created. This map can be viewed in RVIZ itself, as the robot is being moved around its surrounding with teleop. A map is created as the robot moves. Figure 3.19 shows a map being generated of the walls, the obstacles like boxes, etc. present in the factory model we created in gazebo. The white portion of the map represents the mapped portion of the world whereas the grey area represents the unmapped area. The black lines represent the presence of obstacles. After this map is created it is saved for navigation purposes.

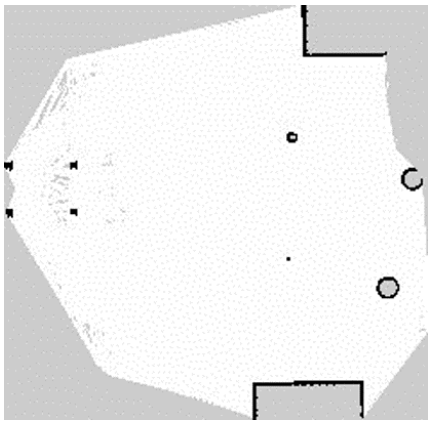


Figure 3.19: Map generated via slam_toolbox of a simulated world in gazebo

The figure above shows the use of Slam-toolbox plugin to save the map above as workspace_map. It is necessary to serialize the map as during navigation, the serialized map data is used. For this, the serial file's location must be included in the parameter file for the SLAM in the config folder.

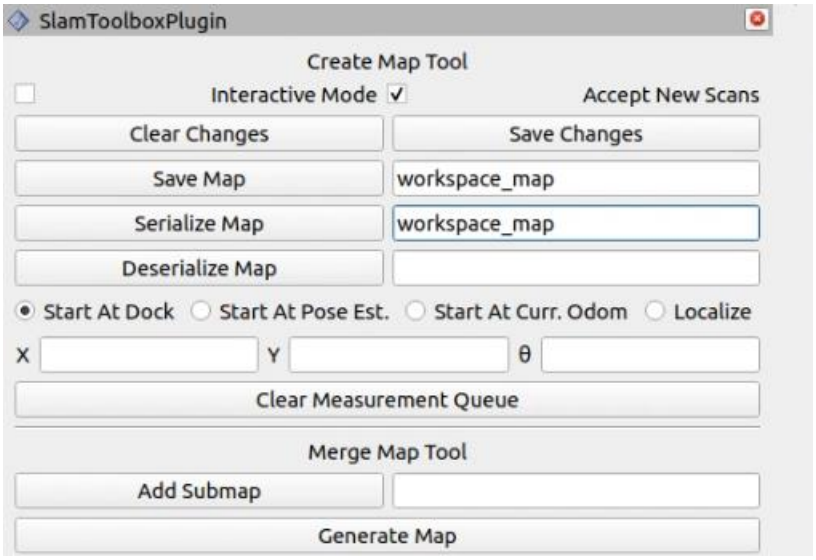


Figure 3.20: Saving the map in RVIZ

When loading the navigation launch files, the previous saved map is provided in the params file inside config is used. Now in RVIZ, a goal pose is given at any point in the above present map. The goal pose is represented by a green arrow as can be seen in the figure below. Nav2 enables the robot to create the shortest path to the goal and thus reach it without human control.

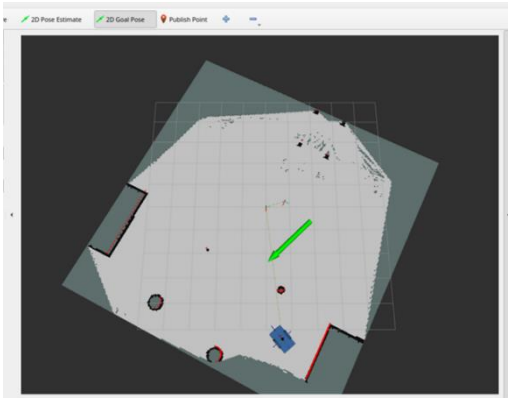


Figure 3.21: Providing goal pose in RVIZ

3.2.5 AMR Control

3.2.5.1 Robot Kinematics

For a Mobile robot, the fundamental component required is its actuators. As done in our simulation, a differential drive mechanism is selected for our robot for its simplicity and easy maneuverability. In this mechanism two drive motors are required in which two wheels are connected. These two wheels are also called power wheels. For the stability of the robot, additional wheels are added as caster wheels. However, three-wheel systems are popular as it can navigate through rough terrain with all three wheels in contact with the ground. This is a limitation in our robot which can be overcome in the future through spring/suspension attachment in casters.

The kinematics of the robot can be modelled in two dimensions as this would provide satisfactory result. So, two motion variables are required to fully define the (translation and rotation) motion of our robot. The linear x velocity and angular z velocity from robot's frame of reference is used. . However, as the actuators used for motion are two motors, those two motors' rotational speed can only be commanded. The following motion model is used for our robot :

Suppose the robot is following a path γ(s) (where s parameterizes the path). The instantaneous linear velocity expressed with respect to the body frame is given by:

$$v^{body,linear} \begin{bmatrix} v_x \\ 0 \end{bmatrix} \quad 4.1$$

Note that the velocity is tangent to the curve γ at s, and that in the body-attached frame the y-component of the velocity is zero (i.e., in the body-attached frame, ν(y)=0). The steering direction is determined by the angle θ as $[\cos\theta, \sin\theta]^T$, so that the linear velocity with respect to the world frame is given by

$$v^{world,linear} \begin{bmatrix} v_x \cos\theta \\ v_x \sin\theta \end{bmatrix} \quad 4.2$$

Because our robot moves in the plane, the z-axis of the body-attached frame is always parallel to the z-axis of the world frame. This greatly simplifies the description of angular velocity, which in this case we may define as $\omega = \dot{\theta}$, the instantaneous rate of change of the robot's orientation.

It is common to combine the angular and linear velocity into a single vector,

$$v^{body} = \begin{bmatrix} v_x \\ 0 \\ \dot{\theta} \end{bmatrix} \quad 4.3$$

$$v^{world} = \begin{bmatrix} v_x cos\theta \\ v_x sin\theta \\ \dot{\theta} \end{bmatrix} \quad 4.4$$

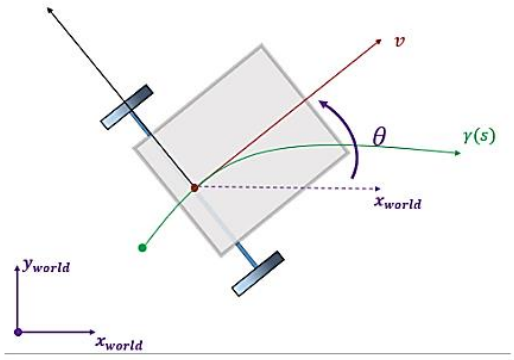


Figure 3.22: Steering direction

Given linear velocities V_x and V_y and angular velocity ω_z, we can find the left (V_l) and right (V_r) wheel linear velocities for a differential drive robot using the following equations:

$$v_l = v_x - \left(\frac{b\omega_z}{2}\right) \quad 4.5$$

$$v_r = v_x + \left(\frac{b\omega_z}{2}\right) \quad 4.6$$

Where, b is the width between two wheels

$$\omega_l = \frac{v_l}{r} \quad 4.7$$

$$\omega_r = \frac{v_r}{r} \quad 4.8$$

Where r is the radius of the wheel.

3.2.5.2 Motor Driver and Controller

Now to drive the motors, a controller that controls and directs the speed of the robot and a motor driver that provides the necessary electric power to the motors as requested by the controller are required. Arduino Uno is used as our motor controller, which runs a program to give the PWM (Pulse Width Modulation) and DIR (direction) values for the Cytron Motor Driver MDD20A. The PWM value is a value between 0 to 255 which defines the power to the motor while the DIR value is either 1 or 0 which defines the direction of motor rotation. The program in microcontroller uses the CytronMotorDriver.h library provided by the manufacturers as:

```
#include "CytronMotorDriver.h"
// Configure the motor driver.
CytronMD motor1(PWM_DIR, 5, 4);  // PWM = Pin 5, DIR = Pin 4.
CytronMD motor2(PWM_DIR, 6, 7); // PWM = Pin 6, DIR = Pin 7.
```

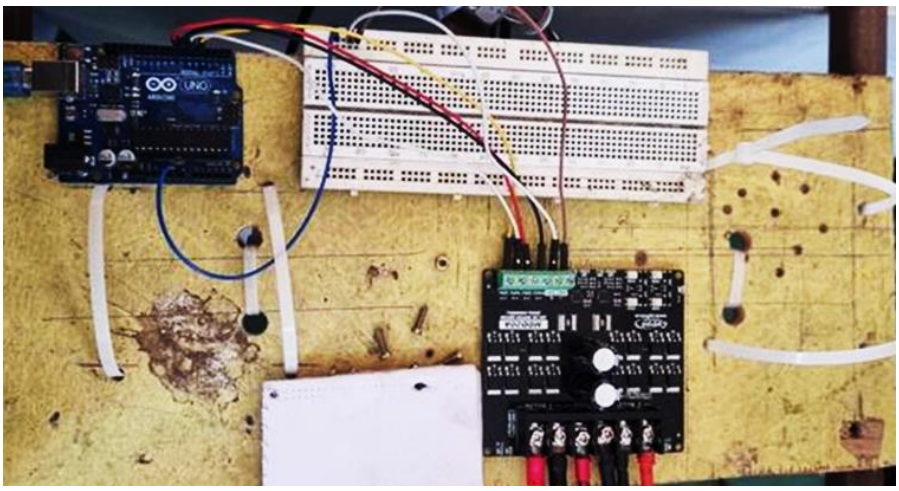


Figure 3.23: Cytron Motor Driver connected to Arduino using breadboard

This allows the open loop control of the robot. The arduino IDE serial monitor is used to command the motor PWM and DIR and observe the motor rotation.

3.2.5.3 Closed Loop Control

For autonomous robot, the robot needs to sense the environment around it as well as its motion. The robot can be sure about its motion by using odometry data. For this, rotoary encoders are used on both motors. The encoder is able to calculate the exact rotation of the wheels by sensing the counts/ticks it receives in each second. The interrupt routine is used for encoders in arudino for counting as:

```
 volatile long left_enc_pos = 0L;
 volatile long right_enc_pos = 0L;
 static const int8_t ENC_STATES [] = {0,1,-1,0,-1,0,0,1,1,0,0,-1,0,-1,1,0};
//encoder lookup table

 /* Interrupt routine for LEFT encoder, taking care of actual counting */
 ISR (PCINT2_vect){
        static uint8_t enc_last=0;
enc_last <<=2; //shift previous state two places
enc_last |= (PIND & (3 << 2)) >> 2; //read the current state into lowest 2 bits
        left_enc_pos -= ENC_STATES[(enc_last & 0x0f)];
```

Now, this odometry data is used to regulate the speed of the motors through a PID loop as shown bellow:

```
 input = p->Encoder - p->PrevEnc;
 Perror = (p->TargetTicksPerFrame - input);
 output = (Kp * Perror - Kd * (input - p->PrevInput) + p->ITerm) / Ko;
 p->PrevEnc = p->Encoder;
 output += p->output;
   p->ITerm += Ki * Perror;
 p->output = output;
 p->PrevInput = input;
```

The values of K_p, K_i, K_d and K_o are selected via hit and trail and tuned as per our motor. The K_o is not seen in other PID equations however is used here just to factor other constants. For tuning out PID variables, first K_i and K_d are kept zero and then K_p value was set to 1 which showed high fluctuation. Then, the K_p value was reduced to 0.1 keeping rest zero which again showed insignificant deflection but did not reach the target value as shown below:

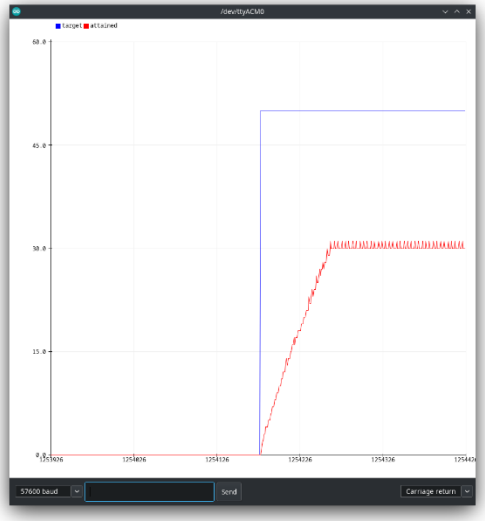


Figure 3.24: PID tuning curve K_p=1, K_o= 50 and rest 0

So, again K_p value was changed to 0.8 which gave satisfactory result, and the K_d value was set to 4 and K_i value to 0.0001 which gave the plot as below:

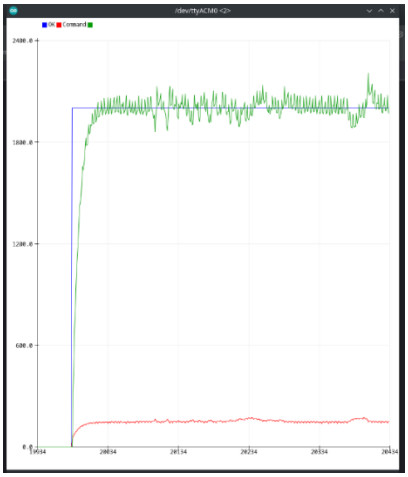


Figure 3.25: PID tuning curve K_p = 0.8, K_i = 0.0001, K_d = 4 and K_o = 50

Again, the fluctuation seemed to be significant, so the K_p value was decreased to 0.75, K_i value to 0.0004 and K_d value to 2 after numbers of iteration. And attained satisfactory PID loop result as below:

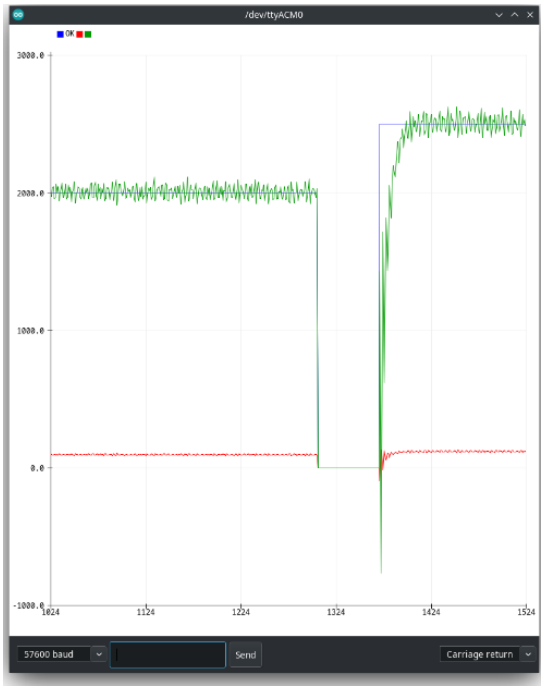


Figure 3.26: PID tuning curve for Kp= 0.75, Ki= 0.0004 and Kd=2

This allowed the closed loop control of our motors where the motor could be driven at the desired speed. The speed variable used in this code is TargetTicksPerFrame, which is the required ticks/counts the encoder should read in one single PID loop. The PID loop rate is set to 30Hz and the counts per revolution of the encoder is 2400 and the gear ratio of left and right motor (wheel) to encoder is 16*1.5 to 16 respectively. This information can be used to convert radian per second to the speed variable use in the Arduino code as;

 # Convert rotational velocities to encoder counts per loop
*mot_1_ct_per_loop = int((omega_l * self.N_PPR * self.GR) / (2 * math.pi * self.f_PID))*
*mot_2_ct_per_loop = int((omega_r * self.N_PPR * self.GR *1.5) / (2 * math.pi * self.f_PID))*

This computation is done separately and the required Ticks Per Frame is passed to Arduino which is explained in the following section.

3.2.5.4 Remote Control and Computation

As our robot is mobile, it requires wireless control. This is handled by Raspberry Pi 400 Model B 2018 model which also handles rest of the computation of our autonomous robot and thus be called the brain of our robot. ROS2 setup was done on the Raspberry Pi as well in the similar manner as done in the PC for simulation purpose. Raspberry Pi communicates with arduino using serial communication through USB (Universial Serial Port). For this we used the python3-serial package's serial library in raspberry pi and encoded the information in UTF-8 format and sent it to the port (/dev/ttyACM0) at 57600 baud rate as;

```
# Serial communication setup
print(f"Connecting to port {self.serial_port} at {self.baud_rate}.")
self.conn = serial.Serial(self.serial_port, self.baud_rate, timeout=1.0)
print(f"Connected to {self.conn}")
.....
# Send the command to Arduino
command = f"m {mot_1_ct_per_loop} {mot_2_ct_per_loop}\n"
self.send_command(command)
```

All of these were done in ROS2 humble by creating ROS2 python package named my_robot_control and an ROS2 node named robot_controller_node.py. This ROS2 node subscribes to the /cmd_vel topic, which consists of the robot velocity in geometry_msgs consisting of linear x , y and z and angular z, y and z velocity. Our ROS2 node subscribes to that topic and computes the count per loop speed variable and publishes it on the topics /mot_1_ct_per_loop and /mot2_2_ct_per_loop. In the meantime, the ROS2 node also passes the message to serial port for the Arduino to read and direct the motor driver to run the motor in the desired speed as asked by the values in /cmd_vel topic. For then, we were publishing messages on /cmd_vel topic using the ROS2 node teleop_twist_keyboard.

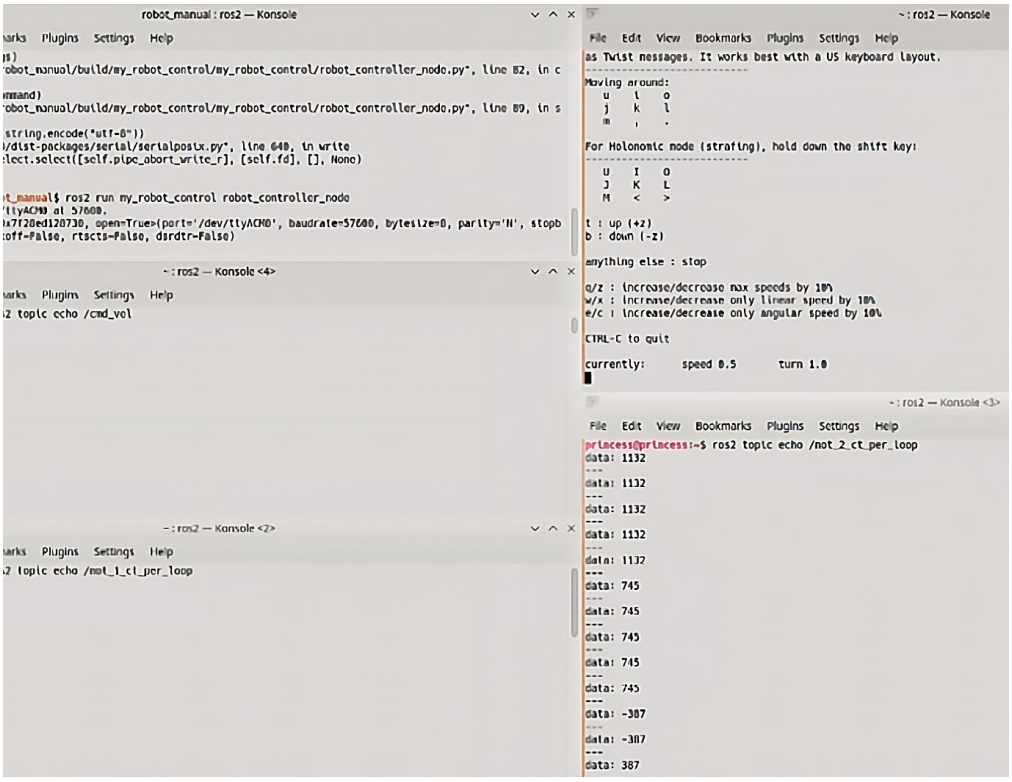


Figure 3.27: Running motors using ros2 node teleop_twist_keyboard

The overall pipeline and actual control circuit of the AMR control is shown in the figure below.

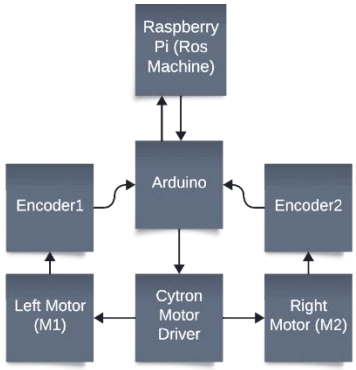


Figure 3.28: Overall pipeline of AMR control (phase I)

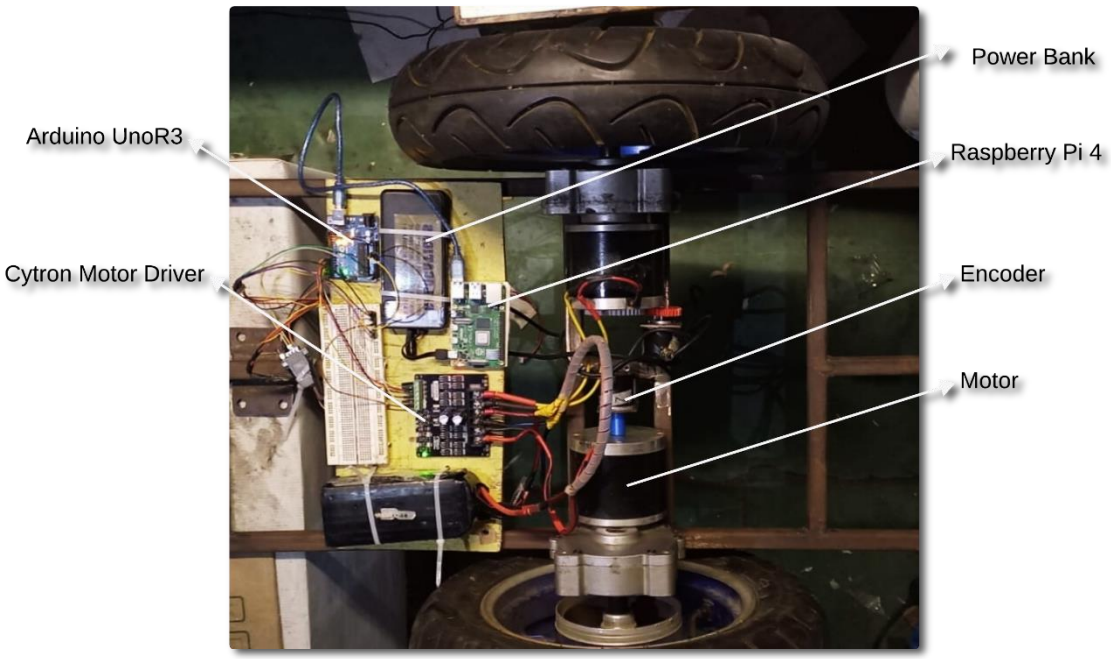


Figure 3.29: Actual control circuit of the AMR (phase I)

3.2.5.5 ROS2 control in AMR

As in simulation, required dependencies for ROS2 Control; ros-humble-ros2-control, ros-humbe-ros2-controllers were installed in Raspberry Pi. The ros2 workspace used for simulation was cloned which included all the description files (URDF), world files, launch files and config files.

The ros2 control for our AMR was implemented using the *diff_drive_controller*, hardware interface plugin, joint state broadcaster plugin, and controller manager. The *diff_drive_controller*, from *ros2_controllers*, translates command velocity into abstract wheel velocities. Our custom hardware interface plugin converts these abstract wheel velocities into signals for motor controllers. The controller manager, within the *ros2_control_node*, coordinates all components. Meanwhile, the joint state broadcaster reads motor encoder positions from the hardware interface and publishes them to the */joint_states* topic, enabling robot state publisher to generate wheel transforms.

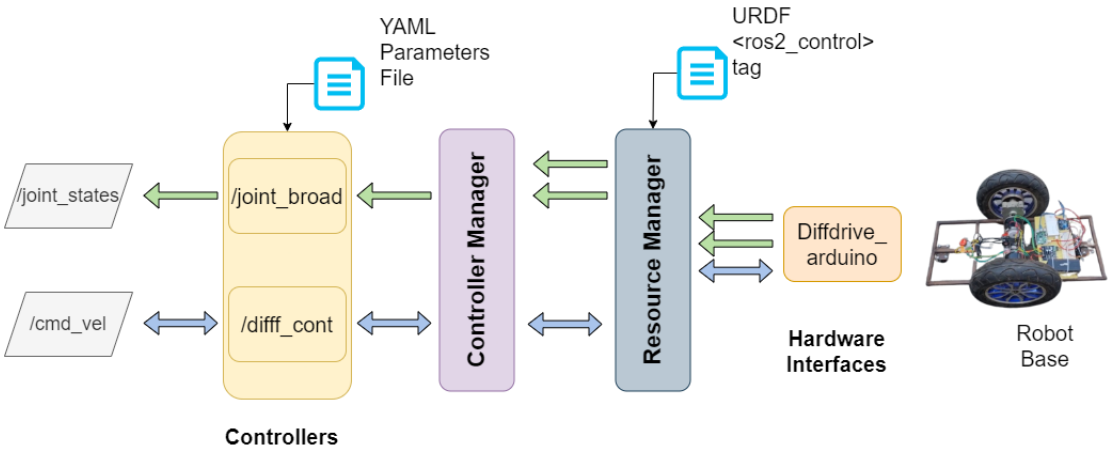


Figure 3.30: ROS2 control system under action

A hardware interface named *diffdrive_arduino* was used which exposes two velocity command interfaces (one for each motor), along with velocity and position state interfaces. Internally, it sends "m" and "e" commands over serial communication. The "m" command is sent with *mot_1_ct_per_loop and mot_2_ct_per_loop* which is taken by the Arduino program and send corresponding PWM value based on the PID loop to the motor driver. This is used to control the velocity of the wheels and is thus computed based on the velocity command interfaces' value. On the other hand, the "e" command sent to the Arduino returns the encoder positions and thus is used to compute the wheel position and velocity which is passed to the respective state interfaces.

In our URDF file, *ros2_control.xacro,* the plugin was changed to diffdrive_arduino/DiffDriveArduino and parameters for the hardware interface plugin were added as:

```
<ros2_control name="RealRobot" type="system">
  <hardware>
    <plugin>diffdrive_arduino/DiffDriveArduino</plugin>
    <param name="left_wheel_name">left_wheel_joint</param>
    <param name="right_wheel_name">right_wheel_joint</param>
    <param name="loop_rate">30</param>
    <param name="device">/dev/serial/by-path/platform-fd500000.pcie-pci-0000:01:00.0-
      usb-0:1.4:1.0 </param>
    <param name="baud_rate">57600</param>
```

```
        <param name="timeout">1000</param>
        <param name="enc_counts_per_rev">2400</param>
    </hardware>
<joint name="left_wheel_joint">
……….
    </joint>
    <joint name="right_wheel_joint">
……….....
 </joint>
</ros2_control>
```

These parameters are utilized by the hardware interface. The "loop rate" determines the frequency of the PID loop per second within the Arduino program. It computes the counts per loop to be transmitted via serial communication to the device specified in the "device" parameter. Here, the device is identified using the by-path address, which corresponds to the fixed USB port of the Raspberry Pi. Hence, the Arduino responsible for the motor driver must be connected to the 4th USB port. The "baud rate" defines the communication speed for serial communication. The "timeout" indicates the duration in milliseconds to wait for serial communication. Lastly, the "encoder counts per revolution" specifies the counts generated by the encoder for each revolution, crucial for determining speed, position, and PID counts per loop. For the encoder used in this project, the manufacturer data wasn't available, so it was calculated experimentally, by counting 15 revolutions in slow rpm on open loop and dividing the corresponding encoder counts by 15.

In our launch file, launch_robot.launch.py, the controller manager node is integrated. This file includes *rsp.launch.py* with arguments *use_sim_time* set to false and *use_ros2_control* set to true, allowing switching between simulation and real robot modes. Additionally, it launches the robot state publisher, responsible for publishing robot transforms and the URDF-provided robot model, which can be visualized using RVIZ2. The *ros2_control_node* from the *controller_manager* package is also included, with parameters *robot_description* and *controller_params_file*. The *robot_description* published by the robot state broadcaster is passed using the *Command ()* function from the *launch.substitutions* library, as the *ros2_control_node* cannot directly input it.

Furthermore, the spawner from the *controller_manager* package is used to include *diff_cont* and *joint_broad*, with a delay implemented to ensure proper launch order, as the latter depends on the former. A *TimeAction* of 3 seconds is used to delay the launch of the controller manager, while the *OnProcessStart* of *RegisterEventHandler* delays the launch of *diff_cont* and *joint_broad* until the *controller_manager* is launched.

Using the new *launch_robot.launch.py* file we could launch all the necessary nodes, controllers, and interfaces for controlling the AMR using ROS2 control. To drive the real AMR using the *teleop_twist_keyboard,* we mapped the */cmd_vel topic* to */diff_cont/cmd_vel_unstamped* as the controller listens to the later topic.

In the meantime, the real AMR motion could be visualized using the rviz2 and adding TF and robot model. The coordinate frame was selected to Odom and the topic for robot model to */robotdescription*.

3.2.5.6 Odometry Check

Odometry is never perfect, which is why we use other localization methods such as SLAM or GPS, but it is still worth getting as good as we can. To check the accuracy of our odometry, the following three steps were taken.

1. Encoder accuracy

   The robot's *base_link* was set as the fixed frame in RVIZ. After driving the robot around and returning it to the initial position, alignment of the axes in RVIZ was compared with the beginning. Satisfactory results were achieved with a count of 2400 encoder counts per revolution.

2. Wheel Radius

   The robot's initial position was marked, and another marker was placed 1 meter ahead. Driving the robot straight until it reached the second marker, we ensured that the distance traveled in RVIZ matched the actual distance of 1 meter. A wheel radius of 0.205 meters provided satisfactory results.

Figure 3.31: Checking the accuracy of odometry

3. Wheel separation

   Once again, the robot's initial position was marked, and a complete rotation was made. The resulting rotation in RVIZ was compared with the actual rotation. Satisfactory results were obtained with a wheel separation of 0.520 meters.

3.2.5.7 Remote Control of AMR with joystick

After successfully testing the Autonomous Mobile Robot (AMR) with the ROS2 node teleop_twist_keyboard, a joystick (Lanjue L3000 Sirius Usb Joystick Gamepad Controller Blue) was utilized for remote control. To enable joystick control, the joy package was installed in ROS2, which converts raw joystick data to the /joy topic and then to a Twist message via the *teleop_twist_joy* package. A launch file named *joystick.launch.py* was employed, launching the *joy_node* of the joy package and the *teleop_node* of the *teleop_twist_joy* package. The *joy_node* received raw joystick data and published button presses and their corresponding values to the */joy* topic. Parameters such as *device_id*, *deadzone*, and *autorepeat_rate* were specified in the

joystick.yaml configuration file. Using this topic, the *teleop_node* published messages to the */cmd_vel* topic based on the parameters in the configuration file.

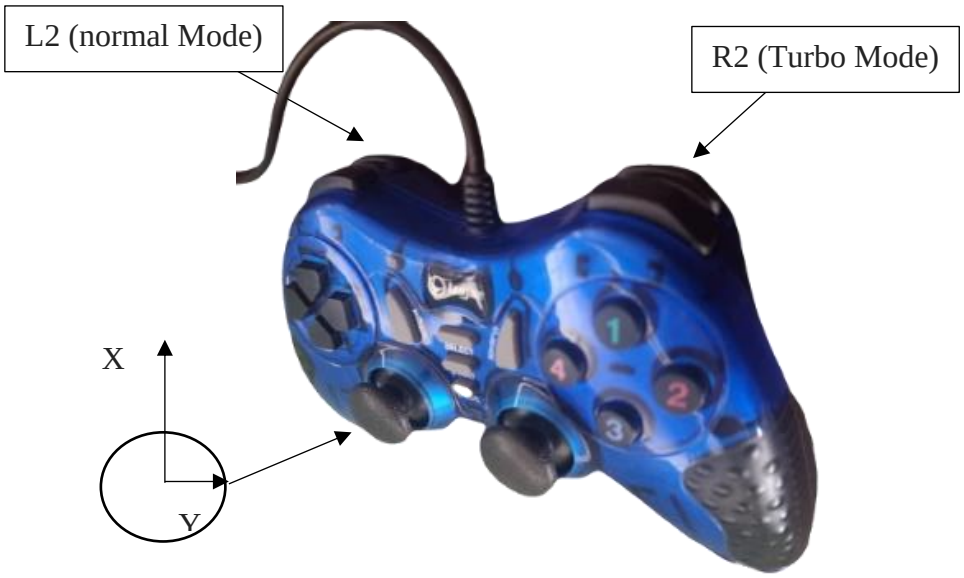


Figure 3.32 Joystick Teleoperation

Linear motion was controlled by axis 1 of the */joy* topic, with *scale_linear* and *scale_linear_turbo* set to 0.5 and 1 respectively for normal and turbo modes. Similar settings were applied for angular motion, with additional button presses of L2 for normal and R2 for turbo, as specified by *enable_button* and *enable_button_turbo* parameters for buttons 6 and 7 respectively, with *require_enable_button* set to true. The */cmd_vel* topic was then remapped to */diff_cont/cmd_vel_unstamped* as required by the controller manager.

3.3 Phase II: Improvement of transportation

In this phase, mapping, localization, navigation, and obstacle detection abilities of the vehicle are enhanced using SLAM. This is implemented using the SLAM package offered in ROS 2 utilizing LIDAR for sensor data input.

3.3.1 SLAM Deployment using LIDAR

3.3.1.1 LIDAR stand fabrication

The optimal position for the Lidar was determined to be at the center of the robot and above the level of the wheels. This configuration allows for a complete 360-degree view of the environment, enabling the Lidar to capture more 2D point clouds and generate a more accurate map. To achieve this positioning, a custom Lidar stand was fabricated. This stand comprises an angle iron of dimension 25x25x3 and 270mm length, welded to the robot frame at the base, and a 3mm thick metal plate measuring 122mm x 88mm welded at the other end. The Lidar holder, designed to attach securely to the metal plate welded to the angle iron, was 3D printed. The metal plate consists of 4 holes which are used to fix the lidar holder with the plates using nuts and bolts.

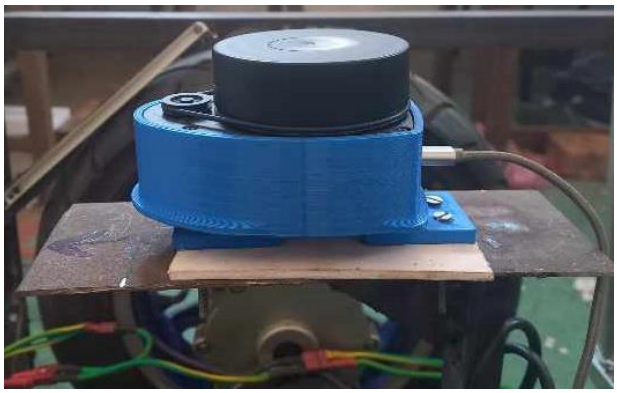


Figure 3.33: LiDAR positioned on top of the plate

3.3.1.2 Using RPLiDAR

A physical LIDAR device is used to generate scan data. RPLiDAR is used in the AMR as its hardware interface is already available in ROS2 repository. For that purpose, installing the necessary packages is necessary. The package for RPLiDar can be installed by the following command:

sudo apt install ros-humble-rplidar-ros

Now the LIDAR device needs to be plugged into the computer's port. This port needs to be registered in the RPLiDAR launch file so as to get it to scan images and display said scan in RVIZ. The port ID can be found via the command:

ls /dev/serial/by-path/

The resulting output is the port Path of LIDAR device which is: pci-0000:00:14.0-usb-0:3:1.0-port0. This path can be different depending on computers/laptops used. Next, it is necessary to write a launch file for this device. Following is a code snippet of said file:

```
Node(
        package='rplidar_ros',
        executable='rplidar_composition',
        output='screen',
        parameters=[{
        'serial_port': '/dev/serial/by-path/pci-0000:00:14.0-usb-0:3:1.0-port0',
        'serial_baudrate': 115200,
        'frame_id': 'laser_frame',
        'angle_compensate': True,
        'scan_mode': 'Standard'
        }])
```

Serial port consists of the path that is previously generated using the 'ls' command. The serial_baudrate value is specific to this LIDAR device and depending on type of RPLiDAR, it can be different. It is necessary to provide "frame" or "frame_id" in this file. Launching this file and RVIZ, the LIDAR takes images in real-time of its surroundings.

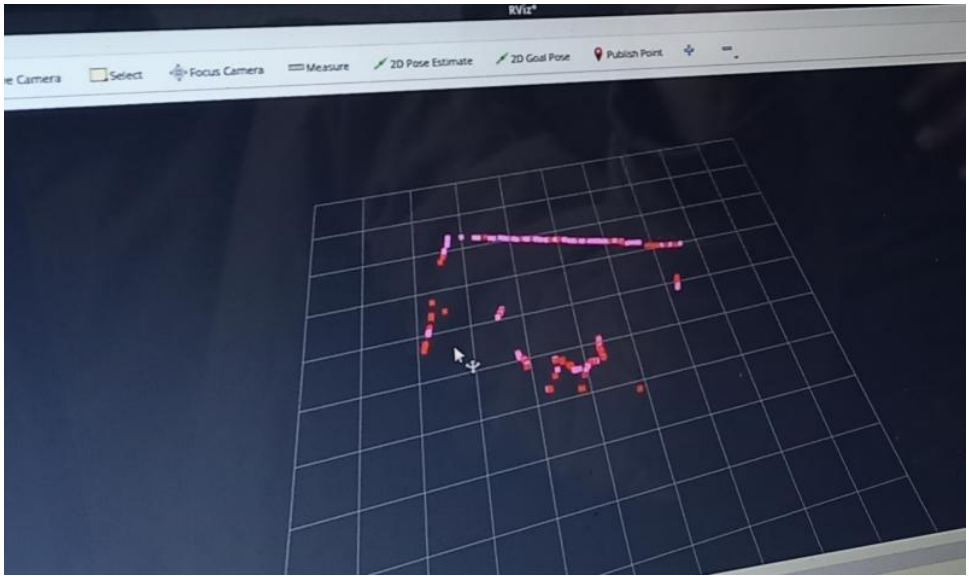


Figure 3.34: LiDAR scan in RVIZ

3.3.1.3 Laser Filter for removing in body point clouds

The AMR's material handling ability was being fabricated simultaneously when the SLAM was being deployed using LiDAR. During the test of lidar with the material handling components in the AMR, it was seen that the lidar would detect the edges of the platform when the AMR would move in an uneven surface. When mapping with SLAM, these points would be seen as obstacles that is black points in the map. As a result, it was necessary to filter these 2D point clouds within the AMR's body so that these points would not be seen as obstacles in the map. Otherwise, these obstacles would not enable the nav2 to find the path given a goal pose.

The laser_filters package is used to perform the filtering of the in-body 2D point clouds. The laser_filters package can be git cloned using the command *git clone* https://github.com/ros-perception/laser_filters.git. Then, the package has to be built and sourced to be used in filtering.

The laser_filters can also be directly installed using the command line in ROS2. The command line for doing this is *sudo apt install ros-humble-perception.* The ros-perception is the meta repository which consists of various packages such as laser_filters, robot_self_filters, etc. There are various types of filters that can be used

for filtering the data points in the laser_filters package. The box_filter best suits our requirement due to its ability to filter (remove) the point clouds within the AMR body of rectangular shape and thus it is used. The following code snippet shows how laser_filters package is used.

```
def generate_launch_description():
   return LaunchDescription([
      Node(
         package="laser_filters",
         executable="scan_to_scan_filter_chain",
         parameters=[
            PathJoinSubstitution([
               get_package_share_directory("project_amr"),
               "config", "my_laser_config.yaml",
            ])],
         ],
      )
   ])
```

Here, this python code is defined in a launch file called *my_laser_filter.launch.py* for launching a ROS 2 node using the launch system in ROS 2. The generate_launch_description function returns a LaunchDescription object, which encapsulates the launch configuration. Inside the LaunchDescription, Node entry is used to specify the package laser_filters to run the executable node scan_to_scan_filter_chain on point cloud data coming through the topic /laserscan, The parameters include a YAML configuration file *my_laser_config.yaml* for configuring the behavior of the node. The PathJoinSubstitution and get_package_share_directory functions are used to construct the absolute path to the configuration file by joining the package's share directory with the relative path to the configuration file. The configuration file includes the following data:

```
scan_to_scan_filter_chain:
  ros__parameters:
    filter1:
      name: box_filter
```

```
type: laser_filters/LaserScanBoxFilter
params:
  box_frame: laser_frame
  max_x: 0.5
  max_y: 0.3
  max_z: 0.0
  min_x: -0.5
  min_y: -0.3
  min_z: 0.0
  invert: false
```

This YAML code defines parameters for a filter node named filter1 within a ROS 2 launch configuration file. The filter type specified is laser_filters/LaserScanBoxFilter, indicating it's configured to process laser scan data. The parameters set the boundaries of a box filter applied to the laser scan data, defining a region in 3D space. The box_frame parameter specifies the frame of reference for the box filter which is *laser_frame*. The *max_x*, *max_y*, and *max_z* parameters define the maximum boundaries of the box along the X, Y, and Z axes respectively, while *min_x*, min_y, and min_z define the minimum boundaries that are required to incorporate the whole robot body. The invert is set to false so that the point inside the box is removed. If it is set to true, then the points inside the box will be kept and the rest will be removed.

3.3.1.4 Autonomous Locomotion Control Circuit

Before starting to map using SLAM, the control circuit previously used for remote control was upgraded connecting the above configured LiDAR to the Raspberry Pi using the Serial USB cable as shown in the figure below.

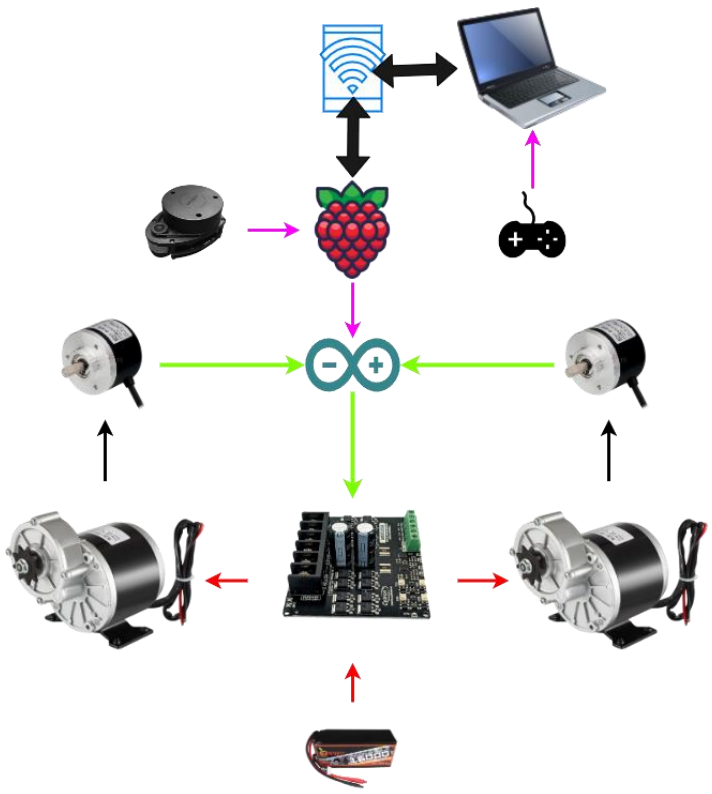


Figure 3.35: AMR Autonomous Locomotion Control Circuit

3.3.1.5 Mapping with SLAM

In the AMR project, we integrated the *slam_toolbox* by Steve Macenski. Initially, we augmented our robot's URDF with an additional link named *base_footprint*. This link has no content and is connected to the *Base_link* via a fixed joint, without any offsets.

The *slam_toolbox* offers various modes, and we're opting for Online Asynchronous mode. Here, operations are live instead of relying on recorded data. Asynchronous processing means always handling the latest scan, even if it means skipping scans. We've copied the param files from the *slam_toolbox* directory to our workspace's config directory. Currently, we're running in mapping mode, utilizing the *odom* frame for odom, *map* frame for map, *base_footprint* for base frame, and */scan_filtered* topic for scan data.

cp /opt/ros/humble/share/slam_toolbox/config/mapper_params__online_async.yaml /home/prince/humbleamr_ws/scr/project_amr /config/

Before running slam, the launch_robot.launch.py and the rplidar.launch.py were launched to launch the robot controllers and the RPLidar.

We have several launch files available for the *slam_toolbox*, including *online_async_launch*, which we'll be using. To launch it from the terminal, use the following command:

ros2 launch slam_toolbox online_asysnc_launch.py params_file:= ./src/projectamr/config/mapper_params_online_async.yaml use_sim_time:=false

The map generated by *slam_toolbox* was visualized in RVIZ by adding the map to the display and setting the topic to */map*. The fixed frame was changed to "*map"* to ensure stability as the map gradually formed while the robot navigated the workspace. Upon returning to the starting point, slight drift in the *"odom"* frame was observed when checking TF frames. Various services provided by the *slam_toolbox*, such as */serialize_map* and */save_map*, were utilized for map management. Additionally, the *slam_toolbox* plugin in RVIZ, accessed via *Panels > Add New Panel > SlamToolboxPlugin*, was utilized for additional functionalities. Options for saving maps in different formats, including *robotics_map_save.pgm* and *robotics_map_save.yaml* for the old format, and *robotics_map_serial.data* and *robotics_map_serial.posegram* for the new format, were explored. Both formats were saved for potential reuse, particularly in localization mode.

Figure 3.36: Mapping of the real environment using Slam Toolbox

To leverage a saved map for localization, the current map in RVIZ was cleared and the system restarted. Subsequently, adjustments were made in the *mapper_params_online_asynce.yaml* file from mapping to localization mode. The name and location of the saved serialized map (without extension) were specified in *Map_file_name*, and *map_start_at_dock* was set to true to ensure the map started at the previous position at the beginning of map creation. Upon rerunning *slam_toolbox*, the previously saved map was loaded, allowing the robot to localize itself within the map. Later, the launch file was copied to our own directory from the ros2 slam toolbox directory.

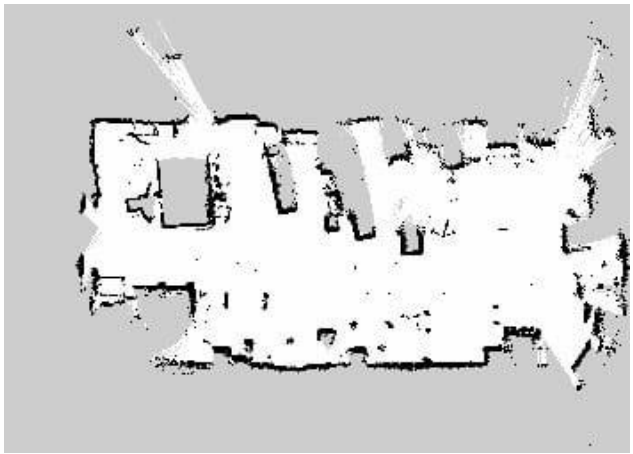


Figure 3.37: Saved map from Slam Toolbox

3.3.2 NAV2 (AMR Navigation)

Once the Slam Toolbox and RVIZ is running with the map, we launched the nav2 using the following prompt:

Ros2 launch nav2_bringup naviagtion_launch.py use_sim_time:=false

The nav2 launch file and config files were copied from the ROS directories to our own project directory.

cp /opt/ros/humble/share/nav2_bringup/params/nav2_params.yaml ./src/projectamr/config/

cp /opt/ros/humble/share/nav2_bringup/launch/navigation.launch.py ./src/projectamr/launch /

We incorporated a new map into RVIZ, setting its topic to */global_costmap/costmap*, which visually represents navigation obstacles and pathways. Redder areas indicate obstacles, while a buffered zone is defined. The navigation system avoids higher-cost regions, utilizes lower-cost areas for planning paths, and resorts to mid-range regions if necessary. By integrating global cost maps, live lidar data, and initial and final poses, the global planner determines paths. A local cost map is also generated, allowing the local planner to refine the path. The nav2 system navigates autonomously along the planned path. To specify the goal pose, we added a navigation panel > tool type > nav2 goal tool, enabling graphical input of the final coordinates and orientation on the map.

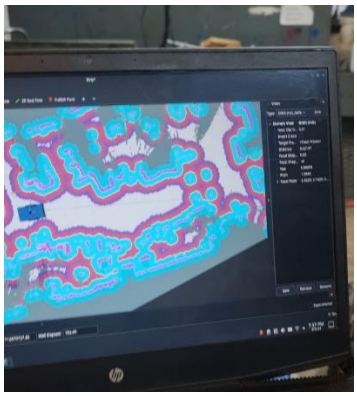


Figure 3.38: Cost map generated with Nav2

Using *twist_mux*, we gave prioritization to the joystick command over nav2 commands on the */cmd_vel* topic so we could overtake the AMR control when the nav2 fails or generates rubbish commands.

3.4 Phase III: Functionality to transportation

In this phase, the vehicle's material handling ability is enhanced through the utilization of computer vision. The required material is detected by the camera installed in the vehicle using ArUco marker. This provides the coordinates of the ArUco marker with respect to the camera frame while we know the location of camera and robot baselink in the map frame. Then the resulting coordinates are transformed to the map frame such that the destination coordinate where the robot baselink should reach with respect to ArUco marker is calculated. Then the calculated coordinate is published to the goal topic of Nav2 which causes the robot to reach the destination autonomously. Once the AMR reaches its destination then with the help of linear actuators, it can carry the materials from its current position to another goal pose.

3.4.1 CV Setup

For object detection purpose OpenCV was used. The latest version to date was 4.9.x, however, most distros ship with version 4.5.x. As ROS2 also ships with the latter version, it was necessary to work with the older version. One can use the latest builds, but there are some changes in API and in some ROS2 setups, it can prove to be a problem due to version conflicts. To install the latest version, pip can be used.

3.4.1.1 Camera Calibration

For material handling purposes, ArucoTags are used. To determine the distance to these tags, it was necessary to calibrate the camera. Calibration was done using a checkerboard pattern of specific width and height, and in this project ROS2's package was used. So, to install these packages, following steps are required:

sudo apt install ros-<ros2-distro>-camera-calibration-parsers
sudo apt install ros-<ros2-distro>-camera-info-manager

sudo apt install ros-<ros2-distro>-launch-testing-ament-cmake

where <ros2-distro> can be humble, galactic depending on installed ROS2 software. Then, a checkerboard pattern was printed of width 11 and height 8 squares. Other patterns can also be used. Camera was opened using ROS2 package with the following command:

ros2 run v4l2_camera v4l2_camera_node

Calibration process can be started using following command in a new terminal. The size of the board has to be reduced by one in the arguments as shown in the command below:

ros2 run camera_calibration cameracalibrator --size 7x10 --square 0.02 --ros-args -r image:=/my_camera/image_raw -p camera:=/my_camera

By displaying the board and moving it in X,Y direction, skewing and bending the board, the color of bars at the right of GUI window changed to green. At this point, the calibration was successful.

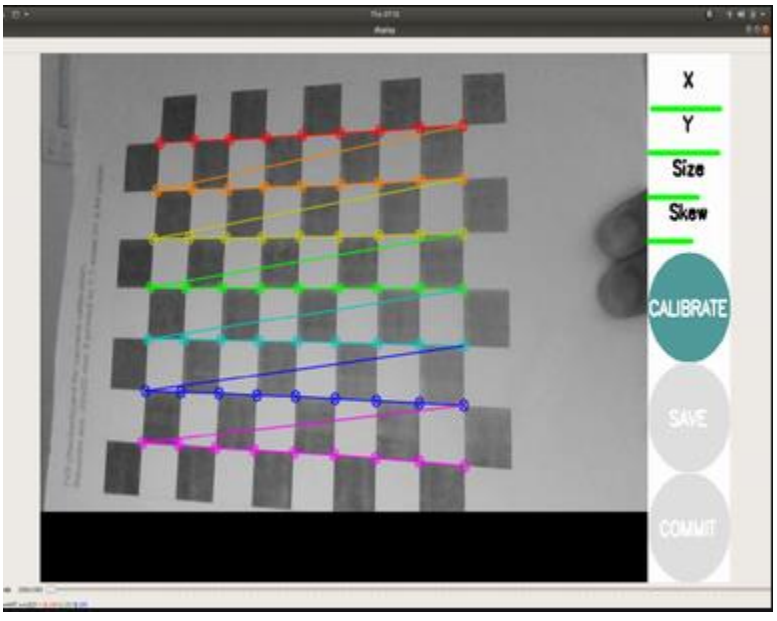


Figure 3.39: Calibration GUI

The terminal also contained information related to camera matrix and distortion coefficients which are important for gauging the distance to ArucoTag.

3.4.1.2 ArUco Marker Generation

This project uses Aruco markers of 4X4-50 type. 4X4 means a tag comprise of 4 by 4 square while 50, means there are 50 unique combinations of such tags. Each of these tags are identified by id number from 0 to 49. It can be generated either from a code or website called arucogen. In the code below, ar_dict specifies the type of aruco tag as 4X4_50. "marker_id" represents id number within 0 to 49, while "marker_size" represents pixel size.

```
aruco_dict=cv2.aruco.DICT_4X4_50
marker_image=cv2.aruco.drawMarker(aruco_dict,marker_id,marker_size)
cv2.imwrite(file_name,marker_image)
```

Finally, the image tag was generated by giving a file name and referencing "marker_image" variable. This was then printed.

Figure 3.40: 4X4_50 id:4 tag

3.4.1.3 ArUco Marker Detection

For ArUco marker detection and its pose estimation, ROS2 node was created. This node was subscribed to an image topic called /image_raw. The following command can be used for detection:

corners, ids, _= cv2.aruco.detectMarkers(image, aruco_dict, parameters=parameters)

Here, “corners” represents coordinates of 4 corners in matrix format, meanwhile “ids” represents the id of the tag. If the tag is not detected, these variables will be null. To calculate the pose of the tag, the following command was used:

rvec, tvec, _ = cv2.aruco.estimatePoseSingleMarkers(corners[i], 0.05, camera_matrix, distortion_coefficients)

For “camera_matrix” and “distortion_coefficients”, separate numpy arrays need to be created and its entries should include the values that were found after calibration. So, “rvec” and “tvec” give rotation and translation vectors respectively. Using these vectors, we can determine the pose of the said aruco tag.

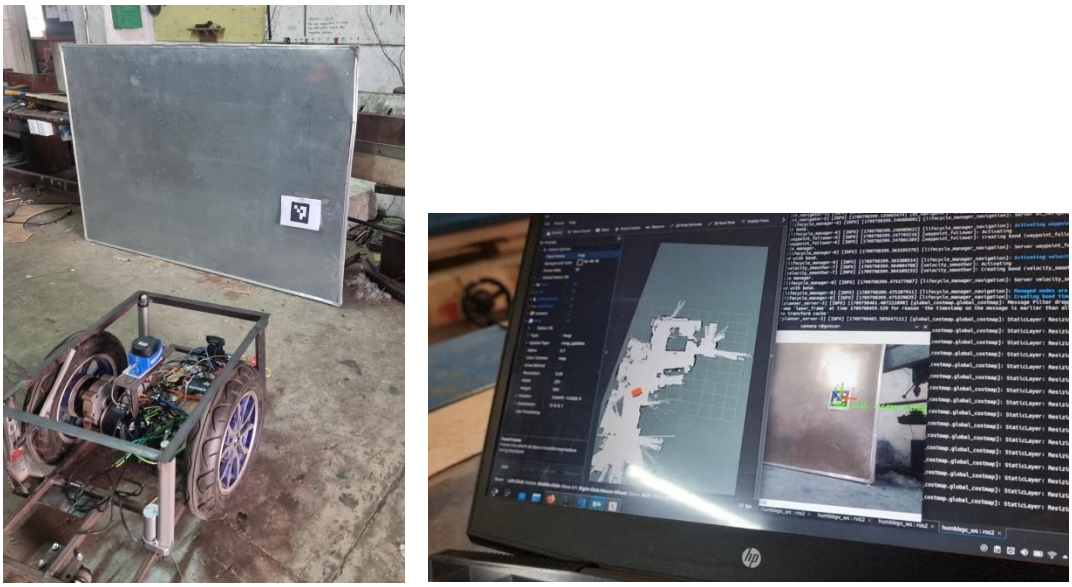


Figure 3.41: CV setup and ArUco tag detection

3.4.2 Material handling

3.4.2.1 Material Handling Design and Fabrication

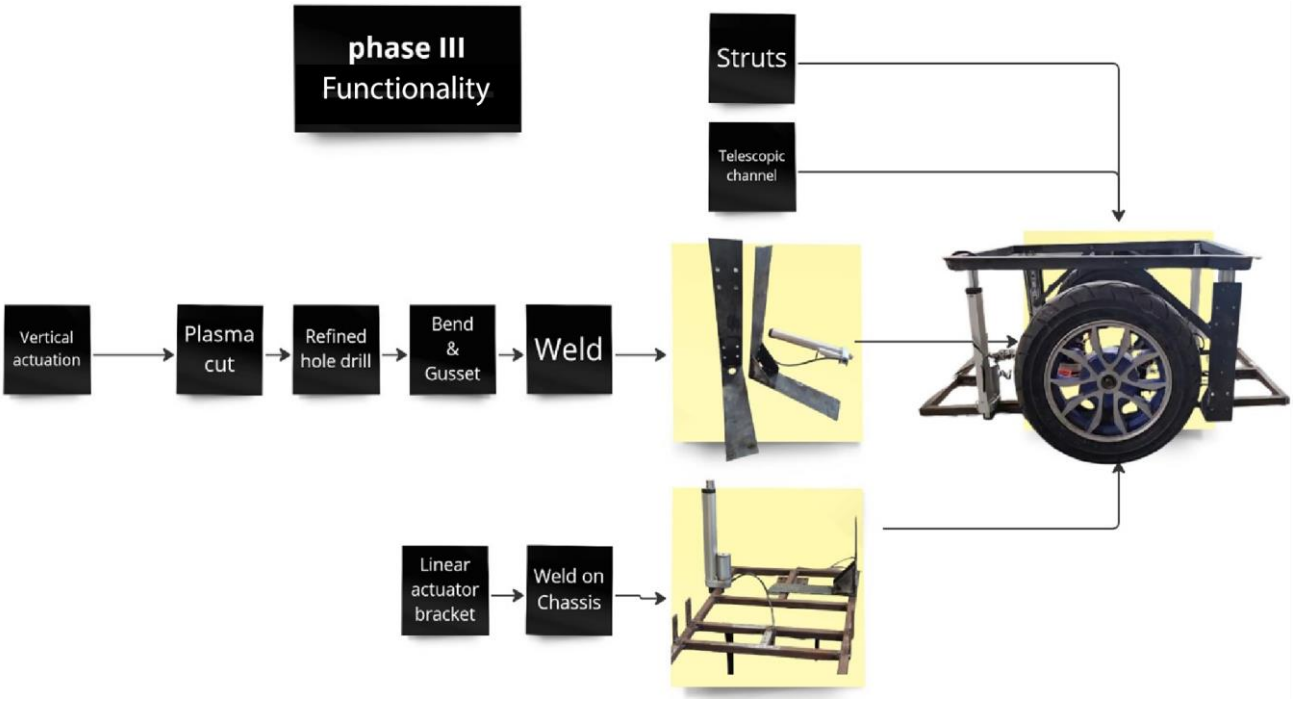


Figure 3.42: Adding vertical actuation mechanism.

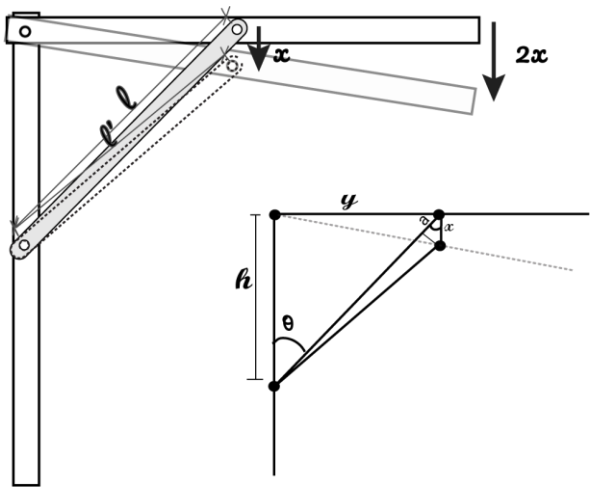
Fabrication of phase 3 involved vertical actuation setup on the AMR. Vertical actuation is crucial in material handling but is challenging with just two linear actuators instead of at least three or four of them. But due to high cost and lack of easy availability, the AMR handles material with just two linear actuators. But it raises challenge when the top platform is unsymmetrically loaded as the linear actuators have hinge joints only and allows rotation of platform about its pin joint axis. Thus, we have telescopic channel at two remaining corners of the platform. Theoretically, assuming all rigid body and telescopic channel allows only one degree of movement, i.e. along vertical axis, the platform should not swivel about the hinge joint at linear actuator heads. But, in practice we do encounter the following:

Pin joints and slider joints allow additional degree of movement due to deflections of pins, deformations of parts in the mechanism due to material's elastic behavior.

These small deformations accumulated allow a gross larger displacement on unsymmetrical loading, i.e. load at corners of the platform that are not directly supported by the linear actuator head.

Due to the lack of tight tolerances on joints of parts, the slider also allows more than one degree of movement.

To solve this displacement of the two corners, struts are installed, attaching onto two points, on linear slider and on platform.



$$a = x \cdot cos\theta$$

$$for\ \theta = 45^o, \qquad \boldsymbol{a = 0.707x}$$

Figure 3.43: Geometry of struts

Let's assume the strut is deformable, and the length can change due to axial force exerted onto it. When the platform rotates about the hinge joint of a linear actuator, the strut squashes smaller by 0.707 times the displacement of the pin joint of the strut, which is close to the actual displacement of the pin joint. It is better to increase this value by reducing θ. There are two ways to reduce θ, one is by increasing h and another is by reducing point of pin joint from hinge, i.e. y. However, due to lack of attachment point on telescopic channel we could not increase h further down. Also reducing y, will increase leverage of load on the deflection, requiring strut of higher strength. Now, the resistance force is provided by the strut that is directly proportional to the displacement of strut.

Another major goal of phase III was to integrate computer vision for docking into the material rack by recognizing the ID of ARuco tags attached on the racks. A 3D printed camera attachment is used. The camera is positioned as shown in Figure 3.36, where:

z = 65mm y = 510mm β (camera inclination) = 75º θ (camera FOV) = 90^o

Figure 3.44 : Position of camera

3.4.2.2 Circuit Design

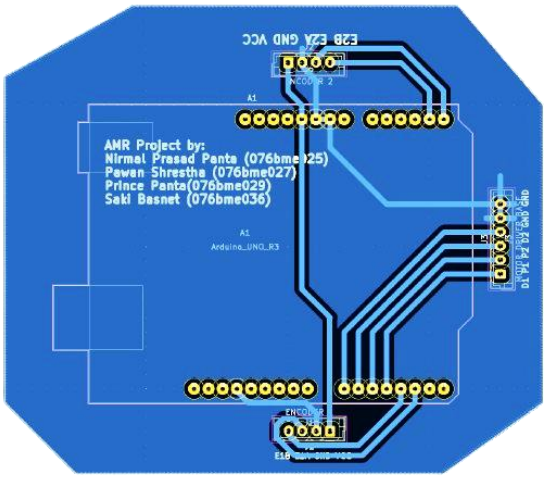


Figure 3.45: Motor signal distribution Arduino UNO R3 shield.

The PCB was designed in KiCAD 7.0 for distribution of PWM signal from the Arduino UNO R3 to Cytron Motor Drivers. It connects the encoders coupled to hub motors shaft to serial input ports of Arduino Uno R3 as shown in Figure 41.

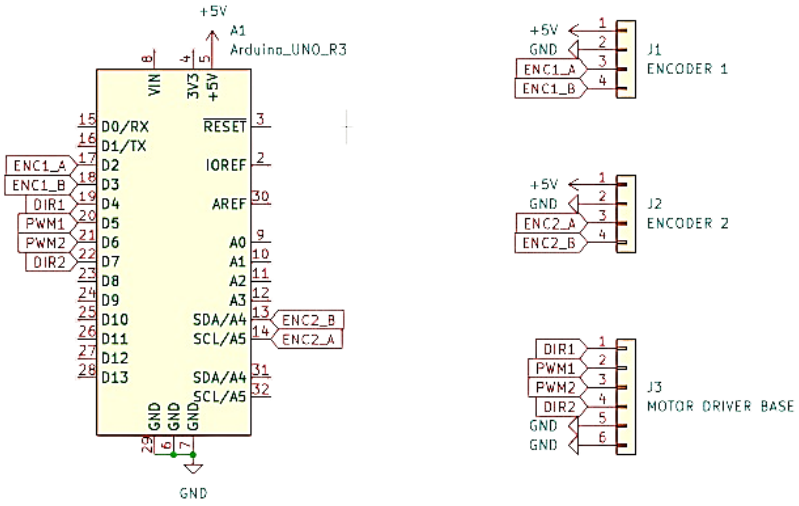


Figure 3.46: Motor signal distribution schematic

The encoder pins of left encoder ENC1_A and ENC1_B are connected to digital pins D2 and D3 respectively, while for right encoder, ENC2_A and ENC2_B are connected to analog pins A4 and A5 respectively.

3.4.2.3 Circuit Board Fabrication and Wiring

In this phase, we transitioned from using breadboards and jumper wires to develop our circuit to fabricating one PCB and one Matrix Broad circuit. We utilized signal wires connected with JST connectors at the end.

After finalizing the PCB layout for the chassis circuit, the design was printed onto glossy paper and transferred onto a clean copper-clad board using a hot lamination machine. An etching solution was prepared by mixing hydrogen peroxide and hydrochloric acid in a 3:1 ratio. The board was immersed in the solution to remove unprotected copper, creating the circuit pattern. Acetone was used as a solvent to remove any remaining toner from the board. Holes for component leads and vias were drilled according to the design to create a prototype of the PCB, as depicted in Figure 43. Finally, components such as 4-pin JST connectors, 6-pin JST connectors, and male pin headers were soldered onto the PCB.

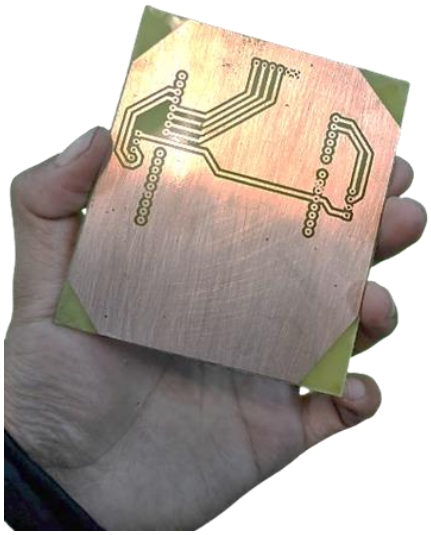


Figure 3.47: Printed PCB prototype on copper-clad

For the Material Handling section, a simple circuit board was created using the matrix board to control two linear actuators. The Arduino Mega's four pins were linked to the four signal pins of the Cytron motor driver, and two other pins served as ground, all connected through a 6-pin JST connector. Components were inserted into the matrix

board, soldered, and interconnected using signal wires and soldering, following the schematic diagram shown in the figure.

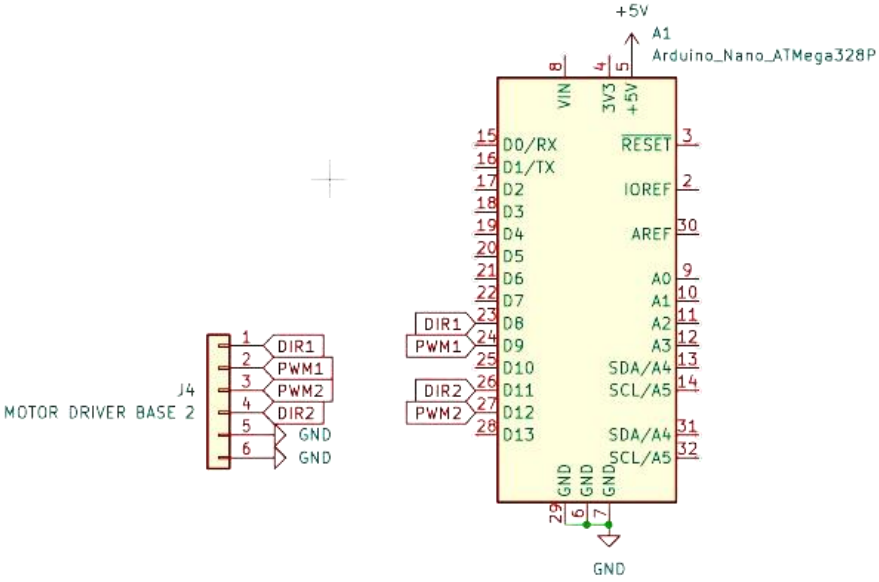


Figure 3.48: Arduino Nano Shield for motor driver

No separate power circuit was needed for the microcontrollers, as they were powered and communicated with the Raspberry Pi via USB cables. Wires from the motor drivers and encoders were connected to a 6-pin JST connector and a 4-pin JST connector, respectively, and later linked to their corresponding connectors on the circuit boards.

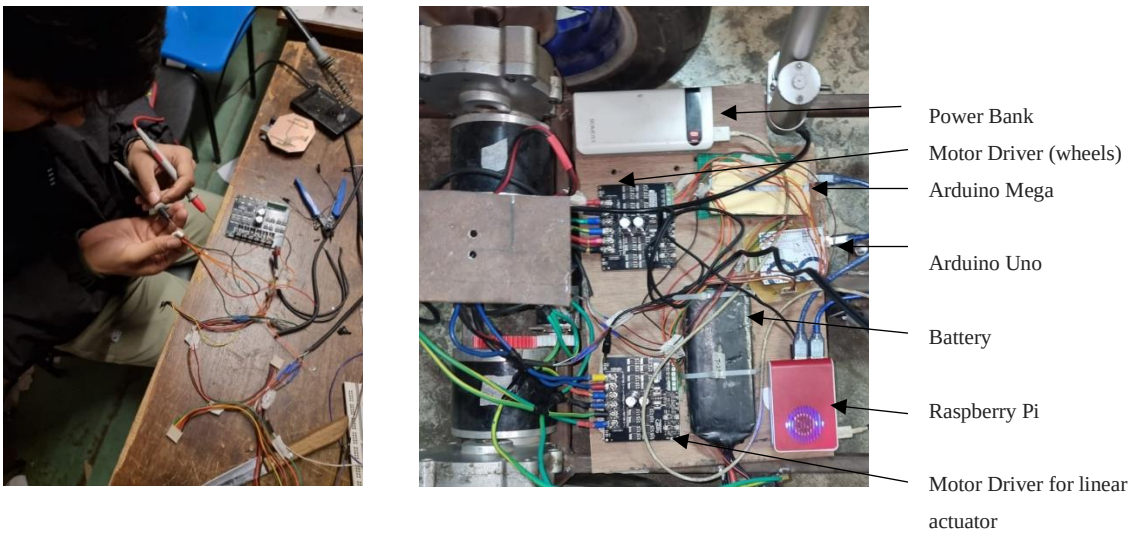


Figure 3.49: Circuit Fabrication and Testing

The motor drivers were connected in parallel to the battery via a power cable passing through an emergency switch. Subsequently, the hub motors and linear actuators were connected to the respective outputs of the motor drivers. For power, the Raspberry Pi was connected to a power bank using a USB Type-C cable.

The final circuit can be represented as shown in the schematic figure below.

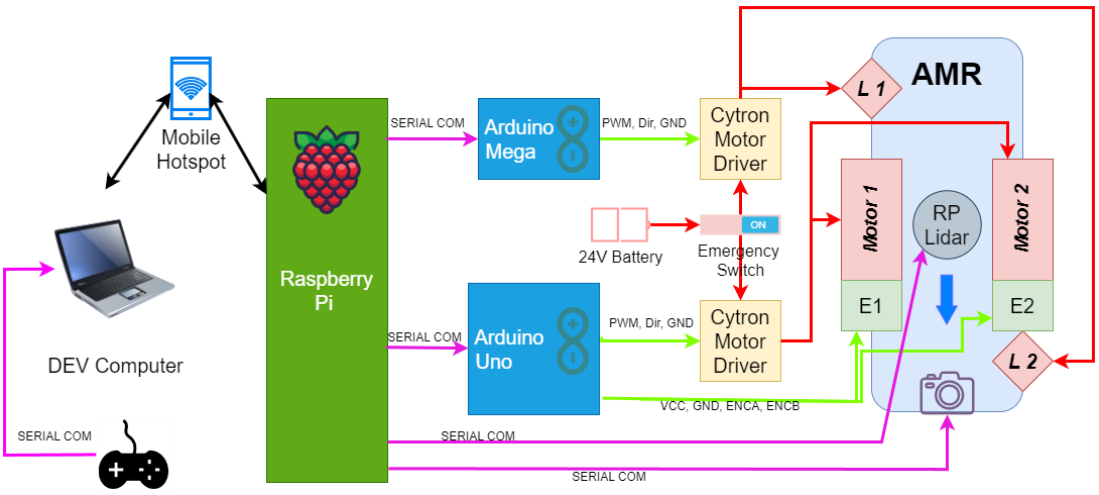


Figure 3.50 Final circuit schematic.

The actual circuit looks as shown in figure below.

Figure 3.51 Final actual circuit.

3.4.2.4 Teleop Material Handling

We utilized base load material handling with two linear actuators, both intended to move simultaneously over specific distances. An Arduino Mega 2560 controls this process in an open-loop manner. We provide a certain PWM signal for a set duration based on experimental findings. Ideally, under no load, applying the same PWM value for the same duration should result in equal displacement for both actuators. However, our design includes constraints at the remaining corners of the frame to prevent bending, resulting in initial loads on the actuators. Due to imprecise prototyping, these resisting forces are not uniform. Consequently, separate PWM values were empirically determined for the two actuators to achieve approximately 9 cm displacement each during assembly.

After experimentation, PWM values of 150/255 and 140/255 were chosen, with a lifting duration of 10.5 seconds and a lowering duration of 10.7 seconds. More time was allocated for the lowering motion to ensure complete descent, especially considering potential battery depletion after transportation, which might affect the effectiveness of the PWM signal. Therefore, a longer duration was set for lowering, although the difference is minimal.

This functionality was implemented within the Arduino program using an if statement. Upon receiving a serial message of '0', the program would provide a positive PWM value to lift the actuators, and it would lower them upon receiving '1'. To prevent repeated actuation from the same command value, a logic was employed: the previous command was stored, and the loop was entered only when the new command differed from the previous one. This was to prevent over-actuation since the maximum displacement of the linear actuators exceeded 9 cm. Time was tracked using the Millis() function, and while the time interval remained below the selected duration, the PWM value was passed through the loop. Once the time interval exceeded the set duration, the loop exited, and no further PWM value was sent, thus stopping the actuators at their current position.

```
// Check if value changed and avoid repeated actions:
if (value != previous_value) {
```

```
  previous_value = value;
  // Control motor direction based on value:
  if (value == 0) {
    motor1.setSpeed(motor_current_forward);
    motor2.setSpeed(motor_current_forward2);
    delay(action_duration);
    motor1.setSpeed(0);
    motor2.setSpeed(0);
  } else if (value == 1) {
    motor1.setSpeed(motor_current_reverse);
    motor2.setSpeed(motor_current_reverse2);
    delay(action_duration2);
    motor1.setSpeed(0);
    motor2.setSpeed(0);
  }
}
```

Raspberry Pi functioning as the brain of the AMR sent the command signals to the Pi through serial communication. It runs a ROS node for this purpose which subscribes to a ros topic /joy topic and reads the button message with index 0 and 2 which are button 1 and 3 of the controller on the right sides. It will first check if the message is received or not based on the availability of the given button index on the message. This code will perform action if only one of the buttons is pressed, so that if both the buttons 1 and 3 of joysticks are simultaneously pressed there won't occur any error. If button index 0 is pressed and button index 2 is not pressed then, it will set the logic value to 0 and if vice versa it will set the logic value to 1.

```
if msg.buttons[button1_index] == 1 and msg.buttons[button2_index] == 0:
        logic_value = 0  # Button 8 pressed, send 0
        self.send_message(logic_value)
    elif msg.buttons[button1_index] == 0 and msg.buttons[button2_index] == 1:
        logic_value = 1  # Button 9 pressed, send 1
        self.send_message(logic_value)
```

The logic value is published to a topic named logic_message of Int8 type. Additionally, it is sent to the serial port as a string representation of the logic value, followed by a newline character, with a baud rate of 115200 and a timeout of 1 second.

```
serial_data = str(logic_value) + "\n"
self.serial_port.write(serial_data.encode())
self.get_logger().info("Message sent: {}".format(logic_value))
```

The joystick is connected in the PC and the previously used launched file is used to run joystick which publishes the message to the /joy topic.

Figure 3.52: Teleop material handling using joystick

3.4.2.5 Simulation of Autonomous Material Handling

Before testing on the real robot, the material handling capability was tested on Gazebo. For this, a workplace was created there with various objects as obstacles. One of the objects was a table with an aruco tag hung in front of it.

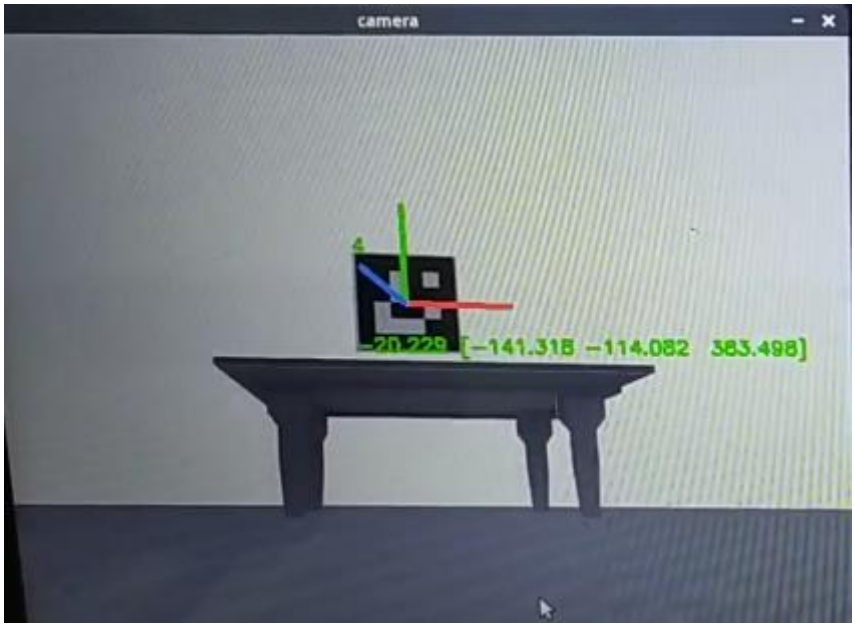


Figure 3.53: Aruco tag in gazebo

This is what the tag looks like from the camera positioned in the robot at gazebo. Here, the coordinate of the tag is being published, which is done by the node subscriber.py. Likewise, with the help of map_aruco.py, the coordinate to the above tag with respect to map frame will be published as well. Finally, nav_aruco.py subscribes to this topic and with the help of commander API, goes towards the table and finally underneath it.

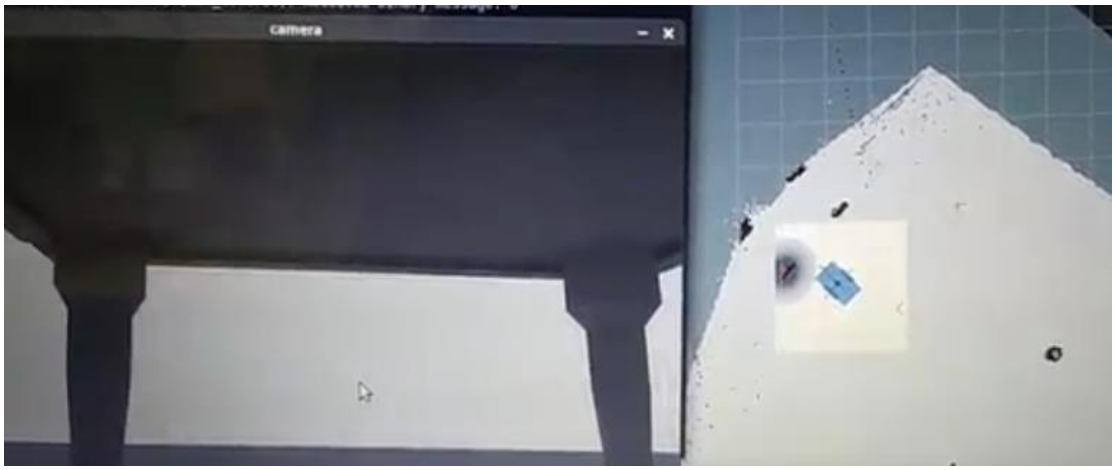


Figure 3.54: Robot moving inside the table in gazebo

3.4.2.6 Automated Goal Pose (docking)

The goal pose is the position and orientation a robot should take regarding the map frame. This goal pose needs to be specified so that the robot goes to said position and orients itself as required. One method of specifying it is through Rviz, which is a GUI method. Here, by clicking the “2D goal pose” button, which is marked by a green arrow, one can set the goal pose.

The other way is through writing a navigation program in either script form or ROS2 node form. The pose of the robot can be sent via a topic, which was done through transform broadcast. Here, the final pose is found through pose estimation done with help of Aruco tag and transformed with respect to the map frame.

navigator = BasicNavigator()

The code above instantiates a navigator instance to help with navigation. Meanwhile, the following code helps to set up the goal pose.

```
goal_pose = PoseStamped()
goal_pose.header.frame_id = 'map'
goal_pose.header.stamp = navigator.get_clock().now().to_msg()
goal_pose.pose.position.x = x
goal_pose.pose.position.y = y
goal_pose.pose.orientation.w = w
```

Likewise, the received pose (x,y,w) was set up to navigate the robot to its destination. Then, the navigator aids the navigation process with the help of Nav2 after writing the following line:

navigator.goToPose(goal_pose)

The code snippet below is to ensure the Nav2 action server works continuously so the goal is achieved.

while not navigator.isTaskComplete()

```
    feedback = navigator.getFeedback()
    if feedback:
        print('Estimated time of arrival:',
Duration.from_msg(feedback.estimated_time_remaining).nanoseconds / 1e9, 'seconds.
        if Duration.from_msg(feedback.navigation_time) > Duration(seconds=600.0):
            navigator.cancelTask()
```

Likewise, as the robot progresses towards its goal, continuous feedback is displayed. This process yields three types of output: Success, Failure and Cancellation. To know which kind of output was recieved, which also helps in debugging case, BasicNavigator() class comes with getResult() method i.e.

```
result = navigator.getResult()
```

It has three cases as stated: TaskResult.SUCCEEDED, TaskResult.FAILED and TaskResult.CANCELED. As it was necessary to align the robot towards the aruco tag, this was accomplished using the spin() method.

```
navigator.spin(spin_dist=0.26, time_allowance=10)
navigator.cancelTask()
```

Basically, the robot spins about 0.26 radians and this process should start within 10ms as stated by time_allowance parameter. Then the task is cancelled. These two lines are supposed to run until and unless the tag was within the field of view of the robot camera. To facilitate this spin maneuver, the navigation node subscribes to a topic which publishes either 1 or 0, depending on whether the tag was in the field of view or not.

```
self.subscription = self.create_subscription(Int16, 'binary', self.navigation, 10)
num = msg.data
```

“self.subscription” allows this node to subscribe to binary topic of type ‘Int16’. The data that received was called ‘msg.data’.

The above program comprises the first stage of automated goal pose. The other stage begins after the robot reaches its goal and does the steps mentioned above. In this phase,

another seperate program should maneuver the robot towards the table containing the correct Aruco tag. To ensure this other program runs, it subscribes to a topic called “nav_aruco”. It publishes value ‘1’, after the robot completes the first phase.

```
#if completed
msg.data=1
self.pub.publish(msg)
```

The second program, after getting this value, calls a function which listens to the transform broadcast between map to aruco or “map_aruco”.

```
transform = self.tf_buffer.lookup_transform(map, aruco, rclpy.time.Time())
```

It looks for transform between map and aruco frame. Once this happens it is possible to get the respective translational and rotational components.

```
t=transform.transform.translation
r=transform.transform.rotation
tvec = np.array([t.x,t.y,t.z])
rvec = np.array([r.x,r.y,r.z,r.w])
```

Where the final pose is representes as: x=t.x, y=t.y, w=r.w. This can be set to goal pose variables as such:

```
goal_pose = PoseStamped()
        goal_pose.header.frame_id = 'map'
        goal_pose.header.stamp = navigator.get_clock().now().to_msg()
        goal_pose.pose.position.x = x
        goal_pose.pose.position.y = y
        goal_pose.pose.orientation.w = w
```

Lastly, as the robot will only be halfway through the rack, it is necessary to navigate the robot, a bit further, by about 0.45m. The following code enables that:

```
robot.driveOnHeading(dist=0.45, speed=0.025, time_allowance=10)
```

3.4.2.7 Autonomous Material Handling

After docking completion, confirmed by obtaining the result with navigator.getResult() returning "TaskResult.SUCCEEDED", two nodes, namely joy_to_msg and msg_to_arduino, are utilized. The joy_to_msg node operates on the PC, similar to the previously used joy_to_Arduino node, but it solely publishes to the logic message without conducting any serial communication. Conversely, the msg_to_arduino node runs on the Raspberry Pi, subscribing to this message and performing the corresponding serial communication as done previously.

After docking, the program autonomously sends a logic message of 0 to actuate the actuators. Subsequently, a TimeAction of 12 seconds is initiated, pausing the program for that duration. Finally, the final goal pose is established using the goToPose function, setting the desired location to (x, y, w) = (2, 0, 1) on the map. Then again, the node was programmed to set logic message to “1” once the “TaskResult.SUCCEEDED” is confirmed, which will lower the payload to its destination.

CHAPTER 4: RESULTS AND DISCUSSION

4.1 Output

The image given presents our developed and tested Autonomous Mobile Robot. ROS2 humble for ROS control, SLAM, Nav2 and OpenCV have been integrated in the robot for autonomous navigation and material handling. Initially, the simulation of the robot in a virtual workspace was completed by creating the robot model of the robot, a virtual world and by integrating gazebo ROS control which is a plugin of ROS control, with SLAM and Nav2. In simulation, the creation of map using SLAM toolbox plugin was done and it was used to navigate the robot from point A to point B by providing goal pose in RVIz.

Simultaneously, the chassis of the AMR was fabricated and by assembling motors and encoders, the basis of AMR control was established and was tested. The motor was PID tuned. While testing the ROS control with AMR, there were some setbacks due to errors encountered in the early stages. Once they were resolved, the AMR was fully equipped with ROS2 control for remote navigation. After, remote control of AMR was done, the SLAM and SLAM toolbox was used to create the map of our workspace at Robotics Club using the point cloud data from RPLiDAR. Then the Nav2 was successfully used to give a goal pose so that the AMR maneuvered accurately avoiding obstacles while reaching the goal pose given using RViz. In this way, phase II of our project was also completed.

For phase III, we integrated the camera with our AMR and calibrated it using ROS2 camera calibration package. This allows us to place the ArUco marker in the cabinet to be transported from its current position to the destination using Nav2’s commander API.

Figure 4.1: Developed and Tested AMR

4.2 Work Completed

4.2.1 Phase I

4.2.1.1 AMR chassis design and fabrication

In Phase I, the CAD model was initially designed in SolidWorks. Subsequently, the frame was fabricated, along with the necessary attachment plates for connecting the hub motor to the frame and other 3D printed components for coupling the rotary encoder with the motor. Consequently, the chassis of the mobile robot required for remote testing was created. Further design and fabrication could then be incorporated into the chassis while testing of the mobile robot was underway. **Error! Reference source not found.** depicts the completed state of the Autonomous Mobile Robot (AMR) following the fabrication of the chassis and the completion of the necessary setup for remote testing.

4.2.1.2 Simulation of AMR in Gazebo and RViz

The ROS2 setup was done simultaneously with the AMR design and fabrication. The project_amr package necessary for the robot simulation in early stage and for overall operation in the completed stage was created. The AMR model required for the simulation was created inside this package using Xacro files. Rather than creating whole model in a single file, various files such as camera.xacro, gazebo_control.xacro, robot_core.xacro, inertial_macros.xacro, robot.urdf.xacro, lidar.xacro, etc were created for ease of handling the files for modification. The plugins necessary for simulation in gazebo such as gazebo control, lidar plugin, camera plugin, etc. were also included in the Xacro files. Similarly, the model of a workspace was created in the Gazebo to replicate the real environment with obstacles. Then, SLAM was used to create the map of the virtual environment and Nav2 was used to give the goal pose to the AMR using the created map. The figure below shows the robot model in the simulated gazebo environment.

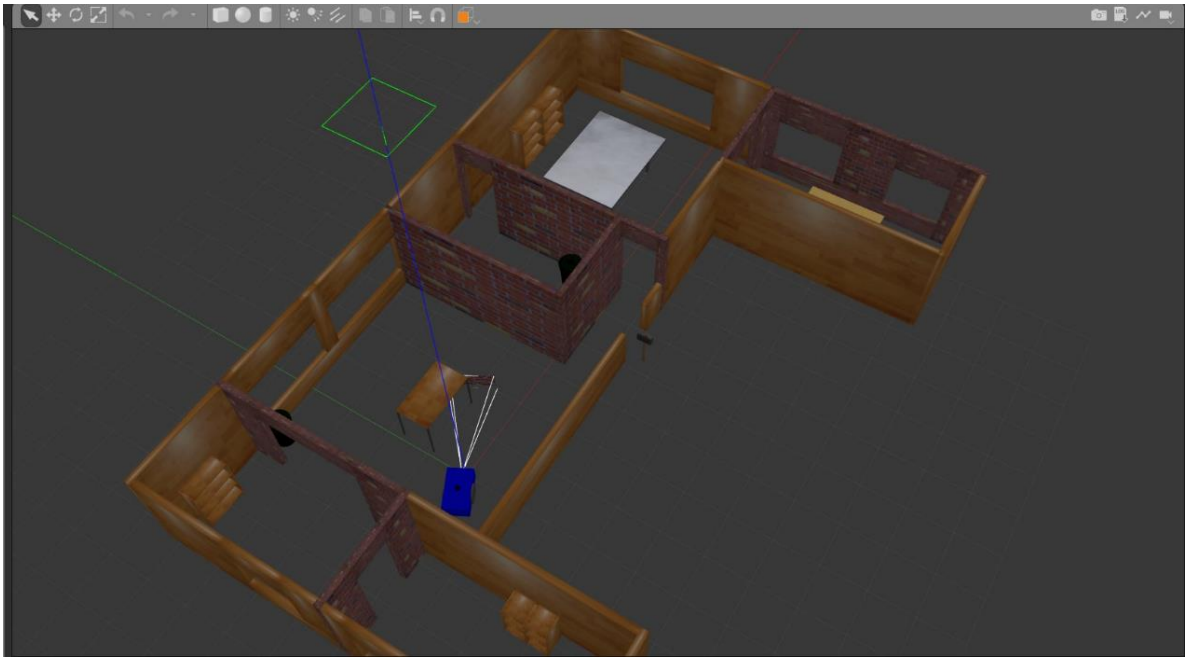


Figure 4.2: Robot model in Gazebo simulation

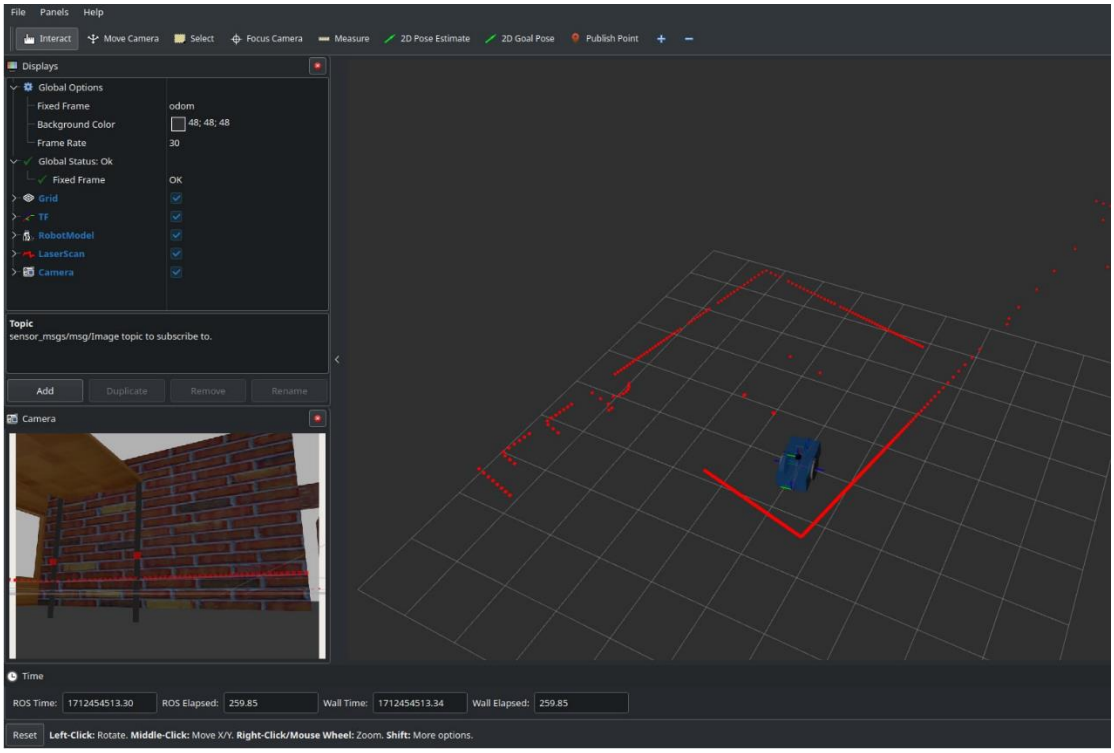


Figure 4.3: Visualization of lidar point clouds and camera in Rviz

4.2.1.3 ROS2 control in AMR and teleoperation

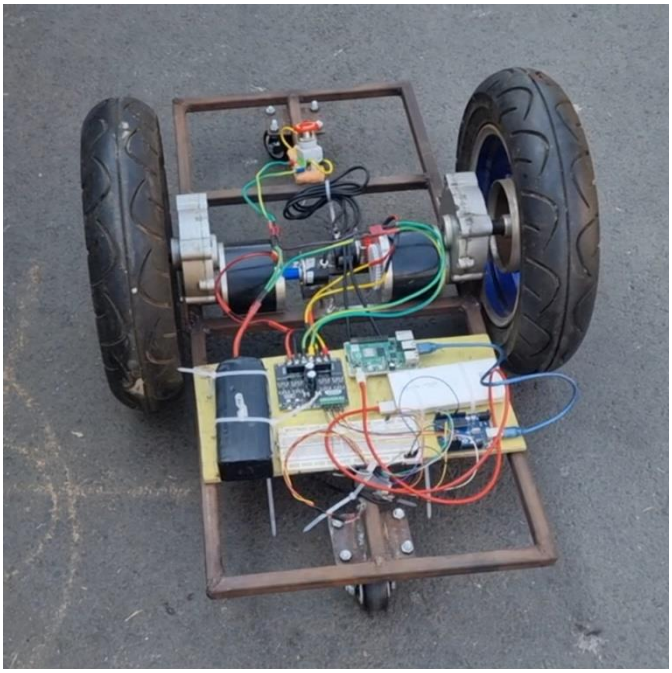


Figure 4.4: AMR setup for teleoperation being tested

Error! Reference source not found. shows the AMR setup done for testing of teleoperation using ros2 control The AMR control was achieved first with open loop control using cyclon motor driver and arduino uno 3. Then the PID tuning was done by using rotary encoders for closed loop control. Once the closed loop control was achieved, ros2 control for our AMR was implemented using the *diff_drive_controller*, hardware interface plugin, joint state broadcaster plugin, and controller manager. Then the AMR was remotely controlled using teleop_twist_keyboard node which requires input from the keyboard of the computer for remote control of the robot. Finally, teleoperation of the AMR was done with joystick using the joy package inbuilt in ROS2.

4.2.2 Phase II

4.2.2.1 Autonomous Navigation using Slam Toolbox and Nav2

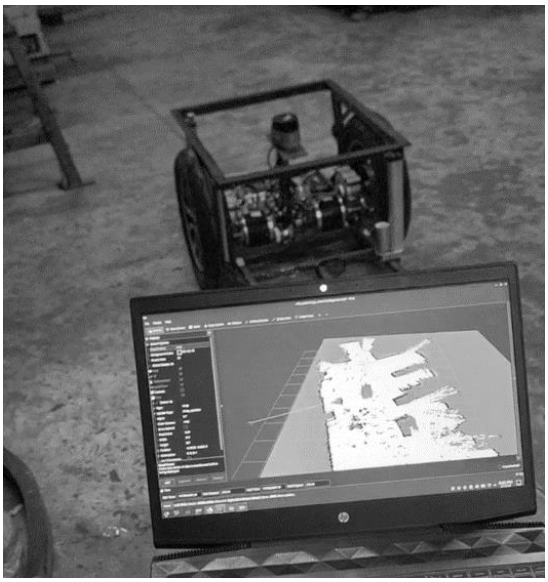


Figure 4.5: Creation of map using SLAM Toolbox in Robotics Club

The testing of autonomous navigation was done in Robotics Club. For the autonomous navigation of the AMR, first the map of the workspace of the club was done using the SLAM Toolbox. In figure 4.5, the white region in the laptop screen represents the mapped areas of the club and the black lines in the map represents the obstacles detected. Then the generated map was used for navigation by using nav2.

4.2.3 Phase III

4.2.3.1 Material Handling design and fabrication

For material handling, a system for raising and lowering the platform using two linear actuators was implemented.

Platform stability: Addressed tilting of the platform under uneven loads by installing struts that compress to counteract the tilting motion. This solution addressed limitations caused by pin joint deflections and loose tolerances.

Camera Positioning: Camera position was determined and attached to the frame for required angle of inclination with a 3D printed snap fit.

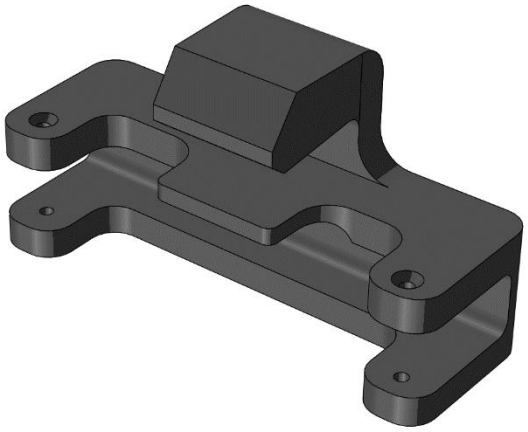


Figure 4.6: Camera attachment

4.2.3.2 Material Handling control and teleoperation

After the incorporation of material handling elements in the AMR, their remote control was done first by using joystick. The figure below shows the AMR in material handling action. Here, the AMR was teleoperated to go inside the table and lift the actuators to transport the whole table to another point. Once the AMR reaches the destination, the linear actuators are set down and the table is placed on the destination point.

Figure 4.7: Initial remote material handling test of the AMR

4.2.3.3 Autonomous Material Handling using ROS2 Control ,Slam Toolbox, Nav2, and ArUco Markers

Autonomous Docking

The navigation part is multi-stage. In the first stage, the robot makes its way towards a site near to a cart or tray. It spins around and stops only when the tag attached to the cart is detected. Once this occurs, a message is published to a topic “nav_aruco” as:

msg.flag=1
self.aruco.publish(msg)

This topic, when published, activates a subscriber node, whose job was to orient the robot under the table that needs to be transported.

However, only half of the robot gets under the table. This occurs as robot orients itself to stated goal pose with respect to the “base_link” frame which is located at its midst. The robot is given a push by 0.45 meters using a simple commander method:

robot.driveOnHeading(dist=0.45, speed=0.025, time_allowance=10)

which will enable it to drive straight ahead with a speed of 0.025 m/s. Finally, the robot gets under the table/rack.

Figure 4.8: Autonomous docking and lifting of payload by the AMR

Autonomous Material Handling

When the logic message of 0 is published by the ROS node, the linear actuators lifts the pay load and after the lifting period of 12 second the final goal pose as set in the program i.e. (2, 0, 1) and the robot reaches the final destination using Slam toolbox and Nav2. Then, the program sends the logic value of '1' which will lower the pay load completing the autonomous material handling.

4.3 Limitations

The following are the limitations of this project:

a) This project aims to implement the already developed tools and software, especially ROS 2 tools and packages for developing and testing AMR. It does not involve understanding the underlying algorithms and codes of the packages such as Nav2, SLAM, ros control, etc.
b) The effect of caster wheel on the robot motion is not yet checked. The robot does not yet possess a suspension system, so all the wheels touch the ground at the same time.

c) The coupling of the encoder is made after designing and fabricating the chassis. This has resulted in a different gear ratio in the encoder which is not the appropriate way for reading data from the encoder.
d) The Lidar may not function appropriately or behave differently in the environment with different lighting.
e) The size of our robot is constrained due to the motors available to us.
f) The AMR is not capable of detecting obstacles below the Lidar level.
g) The linear actuators of the AMR operate in open loop and varying load could cause varying actuation displacement. Also, varying load displacement could cause the two actuators to displace in different amounts causing tilting of the platforms.
h) The accuracy of the coordinate of the ArUco tag with respect to the camera is limited due to the error in calibration. The distance measured using the scale and the distance obtained using the camera were not equal during the calibration and linear regression was used to counter the deviation.

4.4 Problems Faced

4.4.1 Gazebo Lidar

When working with LIDAR plugin, the scans that were produced would change rapidly as the robot model rotated or turned to its sides. This wouldn't occur in the case of robot moving straight ahead. Further down the line, this seems to have affected the map generation using SLAM.

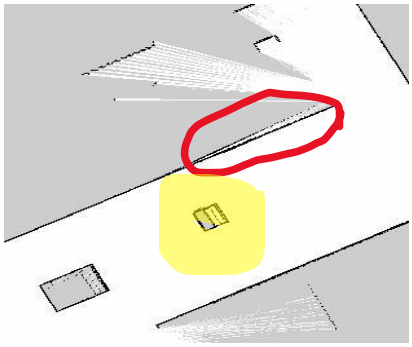


Figure 4.9: Map error

In the above figure, the part highlighted in red and yellow represents areas where map generation is affected. This occurred as the robot rotated about its axis and so did the environment's map.

4.4.2 Limited resources

Due to the lack of backup hardware resources, when we encountered problems with the hardware, we couldn't continue our work unless we managed to replace the hardware or fix it. That had suspended our work for some days.

4.4.3 Failure in debugging problems encountered.

When we were following documentation, and along the way due to some insufficient information and problems with the libraries used, we had to restart with an alternate solution.

4.4.4 Ros2_control and diff_drive arduino

The workspace initially ran using gazebo_control. Using gazebo_control, the robot model was made to run in a Gazebo environment. However, to work with physical hardware, it was necessary to integrate the ros2_control with the overall system. The integration was successful in that, the simulation ran as expected in Gazebo. But, when integrating the hardware interface code written in Arduino with ros2_control, the overall package didn't build successfully. The error message hinted at state_interface for being the problem.

4.4.5 Serial Communication

(to Arduino Mega for Material Handling control)

The previously sent serial data or garbage values would accumulate and cause the actuation of the linear actuators, so a while loop was created to discard such garbage values.

```
// Read the byte:
int value = Serial.parseInt();
```

```
while (Serial.available() > 0) {
  char garbage = Serial.read();
```

4.4.6 Camera Calibration and ArUco Marker Distance Measurement

The camera was calibrated with 50 images input and the obtained camera matrix and distortion coefficient was used to obtain coordinates of ArUco Markers using Open CV libraries. The obtained "z" coordinates were examined with the actual measurements which showed great variation. So, calibration was done again using Nav2 Camera Calibration with 125 images which gave higher accuracy. Still there was difference of 5-10 centimeters depending on the distance, so we developed an empirical relationship between the obtained distance from CV and the actual distance. The linear regression line actual = 1/1.14 fitted the relation perfectly with R^2=1.

4.4.7 Spin Functionality

One of the problems faced was the robot continuously spinning even after detecting the Aruco tag. This problem occured due to the presence of a while loop whose primary job was to make the robot rotate. This loop ran when msg.data = 0. However, it was observed via experimentation that 'msg.data' within loop condition didn't operate, i.e:

```
While(msg.data==0):
#code to stop spinning.
```

As shown above, the "msg.data" didn't get updated even after the tag was within the view of camera. To mitigate this problem, an if-else statement was used instead.

```
num=msg.data
if num == 0:
#code to spin the robot
else:
#stop spinning
#exit
```

This ensured that “num”, was continuously updated, prompting the robot to stop spinning after detecting the tag. To avoid the navigator.goToPose() from executing, flags were used, so that after reaching the goal, the robot only did its spinning task.

4.5 Budget Analysis

The tables below present the cost incurred in the project and the estimated cost of all the items that we have borrowed from different parties.

Table 4.1: Base Cost

S.N.	Description	Estimation in NRs
1	Manufacturing Cost	20,000
2	Logitech C270 Camera	3,895
3	Raspberry Pi 4	20,000
4	Documentation	5,000
5	Miscellaneous	2,000
Total		**50,895**

Table 4.2: Items to be pledged.

S.N	Description	Estimation in NRs
1	Motor Drivers	3,000
2	Hub Motor	22,000
3	Arduino	2,380
4	Linear Actuator	8,000
5	LIDAR	15,000
Total		**50,380**

4.6 Work Schedule (Gantt Chart)

The following table shows the tasks, their starting date (expected starting date) and the end date (expected end date) along with a Gantt Chart.

Table 4.3: Work Schedule

S.N.	Task	Start Date	End Date
1	Literature Review	20-May	15-Feb
2	Proposal	1-Jun	12-Jun
3	AMR Design	15-Jun	25-Dec
4	AMR Fabrication	25-Jun	20-Jan
5	ROS2 setup and Simulation	15-Jun	15-Dec
6	AMR Control	22-Nov	15-Feb
7	SLAM and Nav2 Setup	22-Nov	3-Jan
8	AMR test with SLAM and Nav2	25-Nov	15-Feb
9	CV setup	3-Jan	1-Mar
10	AMR test with CV	5-Jan	5-Mar
11	Documentation	1-Dec	6-Apr

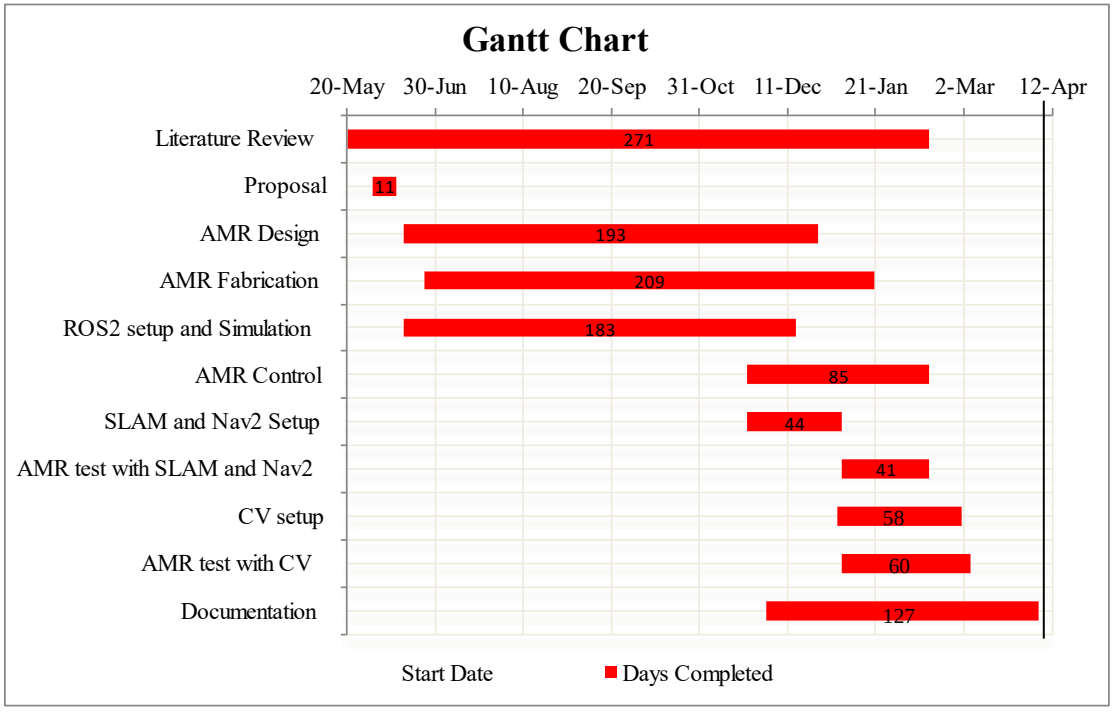


Figure 4.10: Gantt Chart

CHAPTER 5: CONCLUSION AND FUTURE ENHANCEMENT

The development of an autonomous mobile robot (AMR) equipped with Simultaneous Localization and Mapping (SLAM), Nav2 navigation, and computer vision (CV) for material handling marks a significant advancement towards agile and efficient manufacturing processes. The project successfully addresses the pressing need for flexible and adaptable automation solutions in rapidly evolving industries.

Through the integration of ROS2 tools and packages, including ROS2 Control, SLAM, and Nav2, and OpenCV, the AMR demonstrated the capability to autonomously navigate, map its environment, and handle materials with precision. The implementation of SLAM and Nav2 facilitated accurate mapping, localization, and obstacle avoidance, while computer vision technology enhanced the robot's material handling capabilities, allowing it to identify and manipulate payloads effectively.
Despite encountering challenges such as limitations in hardware resources, debugging issues, and calibration discrepancies, the project achieved notable milestones across its phases. From chassis design and fabrication to simulation in Gazebo and RViz, and from teleoperation to autonomous navigation and material handling, each phase contributed to the overall success of the project.

For further refinement and optimization of AMR’s functionality, the following points will be crucial.

- For detecting and avoiding low level obstacles, the height of the robot can be minimized, which is the norm in the industry. For this using a low radius wheel and high torque motors is required. If we are to use the same wheel and motor setup, use of ultrasonic sensors through sensor fusion can be done.
- System incorporating fleets of robot can be developed and path and task planning optimization such as Dijkstra's algorithm can be implemented.
- Advance control algorithms such as Model Predictive Control can be implemented.
- Used of additional sensors like Inertial Measuring Unit (IMU) can be incorporated to enhance the odometry data.

References

[1] A. Hormozi, "Agile manufacturing: The next logical step," *Benchmarking: An International Journal,* vol. 8, pp. 132-143, 05 2001.

[2] J. Bulao, "27+ Astonishing Robotics Industry Statistics You Should Know in 2023," 27 01 2023. [Online]. Available: https://techjury.net/blog/robotics-industry-statistics/.

[3] M. Teulieres, J. Tilley, L. Bolz, P. M. Ludwig-Dehm and S. Wägner, "Industrial robotics: Insights into the sector's future growth dynamics," McKinsey & Company, 2019.

[4] Association for Advancing Automation (A3), "Robot Orders Increase 67% in Q2 2021 Over Same Period in 2020, Showing Return to Pre-Pandemic Demand for Automation," 02 11 2021. [Online]. Available: https://www.automate.org/news/robot-orders-increase-67-in-q2-2021-over-same-period-in-2020-showing-return-to-pre-pandemic-demand-for-automation.

[5] M. Siwek, J. Panasiuk, L. Baranowski, W. Kaczmarek, P. Prusaczyk and S. Borys, "Identification of Differential Drive Robot Dynamic Model Parameters," *Materials,* vol. 16, no. 2, p. 683, 2023.

[6] "Computer vision applications in robotics," 3 03 2023. [Online]. Available: https://www.superannotate.com/blog/computer-vision-robotics.

[7] M. Urwin, "21 Robotics Companies and Startups on the Forefront of Innovation," 25 May 2023. [Online]. Available: https://builtin.com/robotics/robotics-companies-roundup.

[8] G. Ullrich, "The History of Automated Guided Vehicle Systems," in *Automated Guided Vehicle Systems: A Primer with Practical Applications*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2015, pp. 1--14.

[9] W. Grzechca, "Manufacturing in Flow Shop and Assembly Line Structure," *International Journal of Materials, Mechanics and Manufacturing*, vol. 4, no. 1, pp. 25-30, 2016.

[10] M. a. V. K. Zhou, Modeling, simulation, and control of flexible manufacturing systems: a Petri net approach, World Scientific, 1999, pp. 15-37.

[11] G. Fedorko, S. Honus and R. Salai, "Comparison of the Traditional and Autonomous AGV Systems," *MATEC Web Conf.*, vol. 134, p. 00013, 2017.

[12] L. Lynch, T. Newe, J. Clifford, J. Coleman, J. Walsh and D. Toal, "Automated Ground Vehicle (AGV) and Sensor Technologies- A Review," in *2018 12th International Conference on Sensing Technology (ICST)*, IEEE, 2018, pp. 347-352.

[13] N. Zghair, A. Khaleq and A. S. Al-Araji, "A one decade survey of autonomous mobile robot systems," *International Journal of Electrical and Computer Engineering*, vol. 11, no. 6, p. 4891, 2021.

[14] L. Lynch, F. McGuinness, J. Clifford, M. Rao, J. Walsh, D. Toal and T. Newe, "Integration of autonomous intelligent vehicles into manufacturing environments: Challenges," vol. 38, pp. 1683-1690, 2020.

[15] G. Grisetti, R. Kümmerle, C. Stachniss and W. Burgard, "A Tutorial on Graph-Based SLAM," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31-43, 2010.

[16] S. Macenski and I. Jambrecic, "SLAM Toolbox: SLAM for the dynamic world," *Journal of Open Source Software,* vol. 6, no. 61, p. 2783, 2021.

[17] KNAPP, "AMRs and AGVs: Two Automated Guided Vehicle Systems, Compared," 15 5 2023. [Online]. Available: https://www.knapp.com/en/insights/blog/differences-between-agv-amr/. [Accessed 22 12 2023].

[18] S. Macenski, T. Foote, B. Gerkey, C. Lalancette and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," *Science Robotics,* vol. 7, no. 66, May 2022.

[19] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools,* 2000.

[20] M. Kalaitzakis, B. Cain, S. Carroll, A. Ambrosi, C. Whitehead and N. Vitzilaios, "Fiducial Markers for Pose Estimation," *Journal of Intelligent & Robotic Systems,* vol. 101, no. 4, p. 71, 2021.

[21] M. Fiala, "Comparing ARTag and ARToolkit Plus fiducial marker systems," in *IEEE International Workshop on Haptic Audio Visual Environments and their Applications,* 2005.

[22] "ARTag, a fiducial marker system using digital techniques," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05),* 2005.

[23] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas and M. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition,* vol. 47, no. 6, pp. 2280-2292, 2014.

[24] J. Wubben, F. Fabra, C. T. Calafate, T. Krzeszowski, J. M. Marquez-Barja, J.-C. Cano and P. Manzoni, "Accurate Landing of Unmanned Aerial Vehicles Using Ground Pattern Recognition," *Electronics,* vol. 8, 2019.

[25] F. Bergamasco, A. Albarelli and A. Torsello, "Pi-Tag: a fast image-space marker design based on projective invariants," *Machine Vision and Applications,* vol. 24, no. 6, pp. 1295-1310, 2013.

[26] S. Macenski, F. Martin, R. White and J. Ginés Clavero, The Marathon 2: A Navigation System, 2020.

[27] T. Hellström, "Kinematics Equations for Differential Drive and Articulated Steering," 2011.

[28] "Motion Model for the Differential Drive Robot — Introduction to Robotics and Perception," [Online]. Available: https://www.roboticsbook.org/S52_diffdrive_actions.html. [Accessed 21 12 2023].

Development and Testing of Autonomous Mobile Robot for Material Handling

ORIGINALITY REPORT

15%

SIMILARITY INDEX

PRIMARY SOURCES

1	articulatedrobotics.xyz Internet	279 words — 1%
2	navigation.ros.org Internet	252 words — 1%
3	elibrary.tucl.edu.np Internet	221 words — 1%
4	www.roboticsbook.org Internet	154 words — 1%
5	gisresources.com Internet	143 words — 1%
6	digitalcommons.njit.edu Internet	130 words — < 1%
7	techjury.net Internet	128 words — < 1%
8	docs.ros.org Internet	123 words — < 1%
9	forum.arduino.cc Internet	110 words — < 1%
10	control.ros.org Internet	

		87 words — < 1%
11	www.superannotate.com Internet	81 words — < 1%
12	Gang Peng, Tin Lun LAM, Chunxu Hu, Yu Yao, Jintao Liu, Fan Yang. "Introduction to Intelligent Robot System Design", Springer Science and Business Media LLC, 2023 Crossref	77 words — < 1%
13	Michail Kalaitzakis, Brennan Cain, Sabrina Carroll, Anand Ambrosi, Camden Whitehead, Nikolaos Vitzilaios. "Fiducial Markers for Pose Estimation", Journal of Intelligent & Robotic Systems, 2021 Crossref	77 words — < 1%
14	www.me.sc.edu Internet	70 words — < 1%
15	answers.ros.org Internet	67 words — < 1%
16	Automated Guided Vehicle Systems, 2015. Crossref	54 words — < 1%
17	mirror.umd.edu Internet	54 words — < 1%
18	medium.com Internet	49 words — < 1%
19	www.mdpi.com Internet	49 words — < 1%
20	www.science.org Internet	

		49 words — < 1%
21	"Robot Operating System (ROS)", Springer Nature, 2016 Crossref	48 words — < 1%
22	dl.lib.uom.lk Internet	47 words — < 1%
23	github.com Internet	46 words — < 1%
24	docplayer.net Internet	45 words — < 1%
25	Ye Xie, Yunfeng Cao, Biao Wang, Meng Ding. "Disturbance Observer Based Control of Multirotor Helicopters Based on a Universal Model with Unstructured Uncertainties", Journal of Robotics, 2015 Crossref	44 words — < 1%
26	Qin, Zhuan. "An investigation into the effect of the S2 domain of smooth muscle myosin II on its interactions with F-actin.", Proquest, 2015. ProQuest	43 words — < 1%
27	stackoverflow.com Internet	39 words — < 1%
28	fritz.ai Internet	38 words — < 1%
29	www.uco.es Internet	38 words — < 1%

30	"Research into Design for a Connected World", Springer Science and Business Media LLC, 2019 Crossref	35 words — < 1%
31	vigir.missouri.edu Internet	35 words — < 1%
32	S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, M.J. Marín-Jiménez. "Automatic generation and detection of highly reliable fiducial markers under occlusion", Pattern Recognition, 2014 Crossref	31 words — < 1%
33	ebin.pub Internet	30 words — < 1%
34	schutzen-missar.biz Internet	30 words — < 1%
35	Michał Siwek, Jarosław Panasiuk, Leszek Baranowski, Wojciech Kaczmarek, Piotr Prusaczyk, Szymon Borys. "Identification of Differential Drive Robot Dynamic Model Parameters", Materials, 2023 Crossref	28 words — < 1%
36	espace.etsmtl.ca Internet	28 words — < 1%
37	"Proceedings of the 12th International Conference on Robotics, Vision, Signal Processing and Power Applications", Springer Science and Business Media LLC, 2024 Crossref	24 words — < 1%
38	nemertes.library.upatras.gr Internet	24 words — < 1%

39	d-nb.info Internet	23 words — < 1%
40	"Robot Operating System (ROS)", Springer Science and Business Media LLC, 2021 Crossref	22 words — < 1%
41	www.researchgate.net Internet	22 words — < 1%
42	Rafael Herrejon, Shingo Kagami. "Effects of camera calibration errors on trajectory estimation of a flying object using RLS", 2009 4th International Conference on Autonomous Robots and Agents, 2009 Crossref	21 words — < 1%
43	www.cambridge.org Internet	21 words — < 1%
44	www.patentsencyclopedia.com Internet	21 words — < 1%
45	Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, William Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild", Science Robotics, 2022 Crossref	20 words — < 1%
46	ucf.digital.flvc.org Internet	20 words — < 1%
47	Ferreira, Miguel André Gaspar Cordeiro. "Development of an Autonomous Navigation System for a Wheeled Mobile Robot", Universidade do Porto (Portugal), 2024 ProQuest	19 words — < 1%

48	en.wikipedia.org Internet	19 words — < 1%
49	Mahdami, Mohammed Salem Saleh. "Biological Inspired Adaptive Network Controller of Multiple Nonholonomic Unmanned Aerial Vehicle", King Fahd University of Petroleum and Minerals (Saudi Arabia), 2023 ProQuest	18 words — < 1%
50	prism.ucalgary.ca Internet	18 words — < 1%
51	research.library.mun.ca Internet	18 words — < 1%
52	community.openhab.org Internet	17 words — < 1%
53	Eckenhoff, Kevin. "Towards Robust Visual-Inertial Estimation.", University of Delaware, 2020 ProQuest	16 words — < 1%
54	dokumen.pub Internet	16 words — < 1%
55	qspace.library.queensu.ca Internet	16 words — < 1%
56	www.circuitsathome.com Internet	16 words — < 1%
57	www.fritz.ai Internet	16 words — < 1%
58	hdl.handle.net Internet	15 words — < 1%

59	www.hindawi.com Internet	15 words — < 1%
60	windeng.t.u-tokyo.ac.jp Internet	14 words — < 1%
61	Kybernetes, Volume 31, Issue 5 (2006-09-19) Publications	12 words — < 1%
62	adudspace.adu.edu.tr:8080 Internet	12 words — < 1%
63	digbib.ubka.uni-karlsruhe.de Internet	12 words — < 1%
64	pdfs.semanticscholar.org Internet	12 words — < 1%
65	www.diva-portal.org Internet	12 words — < 1%
66	www.fho-emden.de Internet	12 words — < 1%
67	coek.info Internet	11 words — < 1%
68	official.satbayev.university Internet	11 words — < 1%
69	pastebin.com Internet	11 words — < 1%
70	Cavazos, Arely. "Designing for Health and Wellness in Affordable Urban Housing.", The Florida State University, 2020 ProQuest	10 words — < 1%

71	Gonzalez Bellido, Eduardo Andre. "Real-time Quantitative Sonoelastography in an Ultrasound Research System", Pontificia Universidad Catolica del Peru - CENTRUM Catolica (Peru), 2021 ProQuest	10 words — < 1%
72	Joshua G. Mangelson, Ryan W. Wolcott, Paul Ozog, Ryan M. Eustice. "Robust visual fiducials for skin-to-skin relative ship pose estimation", OCEANS 2016 MTS/IEEE Monterey, 2016 Crossref	10 words — < 1%
73	Rocha, Ryan Alexander. "Toward Autonomous In-Flight Docking of Unmanned Multi-Rotor Aerial Vehicles.", University of California, Davis, 2020 ProQuest	10 words — < 1%
74	Tomasz Steclik, Rafal Cupek, Marek Drewniak. "Automatic grouping of production data in Industry 4.0: The use case of internal logistics systems based on Automated Guided Vehicles", Journal of Computational Science, 2022 Crossref	10 words — < 1%
75	ca-tehachapi2.civicplus.com Internet	10 words — < 1%
76	commons.erau.edu Internet	10 words — < 1%
77	eprints.usm.my Internet	10 words — < 1%
78	ntnuopen.ntnu.no Internet	10 words — < 1%
	ourspace.uregina.ca	

79	Internet	10 words — < 1%
80	repository.eafit.edu.co Internet	10 words — < 1%
81	roboticsbackend.com Internet	10 words — < 1%
82	transcience.ru Internet	10 words — < 1%
83	Al Qattan, Khaled Mageed Ahmad A E. "Assessment of the Physicochemical Properties of Drinking Water of Selected Schools in Kuwait", University of Malaya (Malaysia), 2023 ProQuest	9 words — < 1%
84	Casaday, Brian P.. "Numerical Simulation of Atmospheric Internal Waves with Time-Dependent Critical Levels and Turning Points", Brigham Young University, 2021 ProQuest	9 words — < 1%
85	Guo Zhenglong, Fu Qiang, Quan Quan. "Pose Estimation for Multicopters Based on Monocular Vision and AprilTag", 2018 37th Chinese Control Conference (CCC), 2018 Crossref	9 words — < 1%
86	Heydari, David. "Electric-Field Induced Nonlinear Optics in CMOS Silicon Nanophotonic Waveguides", Stanford University, 2023 ProQuest	9 words — < 1%
87	Juan C. Tejada, Alejandro Toro-Ossaba, Santiago Muñoz Montoya, Santiago Rúa. "A Systems	9 words — < 1%

Engineering Approach for the Design of an Omnidirectional Autonomous Guided Vehicle (AGV) Testing Prototype", Journal of Robotics, 2022
Crossref

88 Timber, Luke C.. "Live Load Distribution and Evaluation of Bridges Using Digital Image Measurements", University of Delaware, 2022
ProQuest
9 words — < 1%

89 Wong, Fai L. "Novel Microwave Passive Devices For Dual-Band Applications", Proquest, 2013.
ProQuest
9 words — < 1%

90 Yazdansepas, Iman. "Room Categorization Using Simultaneous Localization and Mapping and Convolutional Neural Network", Purdue University, 2023
ProQuest
9 words — < 1%

91 Zhang, Tong. "Robust Visual Observer and Controller Design for System Modeled on SE(3) With Camera Measurements", University of Windsor (Canada), 2023
ProQuest
9 words — < 1%

92 community.blynk.cc
Internet
9 words — < 1%

93 dspace.cvut.cz
Internet
9 words — < 1%

94 eci.intel.com
Internet
9 words — < 1%

95 fim.uni-pr.edu
Internet
9 words — < 1%

96 findresearcher.sdu.dk

Internet

9 words — < 1%

97 st.robu.in
Internet

9 words — < 1%

98 st5drone.metz.centralesupelec.fr
Internet

9 words — < 1%

99 thesis.library.caltech.edu
Internet

9 words — < 1%

100 vdoc.pub
Internet

9 words — < 1%

101 www.iieta.org
Internet

9 words — < 1%

102 "Mobile Radio Communications and 5G Networks", Springer Science and Business Media LLC, 2024
Crossref

8 words — < 1%

103 Amato-Yarbrough, Matthew. "Facilitating Data Communication Between Supervised Modules in a Distributed Unmanned Aerial Vehicle Software Architecture", Northern Arizona University, 2024
ProQuest

8 words — < 1%

104 Jianqi Zhang, Xu Yang, Wei Wang, Jinchao Guan, Ling Ding, Vincent C.S. Lee. "Automated guided vehicles and autonomous mobile robots for recognition and tracking in civil engineering", Automation in Construction, 2023
Crossref

8 words — < 1%

105 M. Manoj Kumar, Bhuvaneshwari Hegde, S. P. Veda Murthy, M. K. Akhila, A. S. Bhoomika. "Chapter 9 Human Activity Recognition in Construction Industry Using

8 words — < 1%

Machine Learning Pose Estimation Technique", Springer Science and Business Media LLC, 2024
Crossref

106 Pannir Sivajothi, Sindhana Selvi. "Studies on Molecular Polaritons: Condensates, Topology, and Metrology", University of California, San Diego, 2023 — 8 words — < 1%
ProQuest

107 Sudhakar, Ashok E. M.. "The Behavior of sUASs under Explosive Loading Conditions and Implications for Safe Operating Procedures", Missouri University of Science and Technology, 2020 — 8 words — < 1%
ProQuest

108 Vieira, Bruno Carvalho. "Reconversão da Plataforma Robuter num AGV com Guiamento Visual", Universidade de Aveiro (Portugal), 2024 — 8 words — < 1%
ProQuest

109 arxiv.org — 8 words — < 1%
Internet

110 da Silva Vieira do Coito, Francisco Miguel. "Study on the Development of an Autonomous Mobile Robot", Universidade NOVA de Lisboa (Portugal), 2024 — 8 words — < 1%
ProQuest

111 de Sousa, César Miguel Rodrigues. "Visual Servoing of a Human Head Using Fixed Targets", Universidade de Aveiro (Portugal), 2024 — 8 words — < 1%
ProQuest

112 docs.px4.io — 8 words — < 1%
Internet

113 epdf.tips — 8 words — < 1%
Internet

114	fenix.tecnico.ulisboa.pt Internet	8 words — < 1%
115	gist.github.com Internet	8 words — < 1%
116	hashnode.com Internet	8 words — < 1%
117	irp-cdn.multiscreensite.com Internet	8 words — < 1%
118	liu.diva-portal.org Internet	8 words — < 1%
119	odr.chalmers.se Internet	8 words — < 1%
120	plagiarismcheckerturnitinaccount.blogspot.com Internet	8 words — < 1%
121	romeo.univ-reims.fr Internet	8 words — < 1%
122	tuxdb.com Internet	8 words — < 1%
123	webthesis.biblio.polito.it Internet	8 words — < 1%
124	www.elastic.co Internet	8 words — < 1%
125	www.emeraldinsight.com Internet	8 words — < 1%
	www.expotec.ufrpe.br	

126	Internet	8 words — < 1%
127	www.iaarc.org Internet	8 words — < 1%
128	www.intel.com Internet	8 words — < 1%
129	www.sase.com.ar Internet	8 words — < 1%
130	"New Advances in Mechanisms, Transmissions and Applications", Springer Science and Business Media LLC, 2023 Crossref	7 words — < 1%
131	Attia, Mohamed. "Study of Design and Thermal Treatment of Automotive Control Arm Fabricated from A357 Semi-Solid Alloy", Universite du Quebec a Chicoutimi (Canada), 2020 ProQuest	7 words — < 1%
132	Hung-Hsing Lin, Ching-Chih Tsai. "Improved global localization of an indoor mobile robot via fuzzy extended information filtering", Robotica, 2008 Crossref	7 words — < 1%
133	Muhammad Aizat, Ahmad Azmin, Wan Rahiman. "A Survey on Navigation Approaches for Automated Guided Vehicle Robots in Dynamic Surrounding", IEEE Access, 2023 Crossref	7 words — < 1%
134	Victor Mayoral-Vilches, Ruffin White, Gianluca Caiazza, Mikael Arguedas. "SROS2: Usable Cyber	7 words — < 1%

Security Tools for ROS 2", 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2022
Crossref

135 Wayne Durham, Kenneth A. Bordignon, Roger Beck. "Aircraft Control Allocation", Wiley, 2016 — 7 words — < 1%
Crossref

136 Weiyue Chen, Wuxing Jing. "Dynamics Equations of Relative Motion around an Oblate Earth with Air Drag", Journal of Aerospace Engineering, 2012 — 7 words — < 1%
Crossref

137 Isanka Diddeniya, Indika Wanniarachchi, Hansi Gunasinghe, Chinthaka Premachandra, Hiroharu Kawanaka. "Human–Robot Communication System for an Isolated Environment", IEEE Access, 2022 — 6 words — < 1%
Crossref

138 Stathakis, Alexan. "Vision-Based Localization using Reliable Fiducial Markers", Proquest, 2014. — 6 words — < 1%
ProQuest

139 amslaurea.unibo.it — 6 words — < 1%
Internet

140 inohira.mns.kyutech.ac.jp — 4 words — < 1%
Internet

EXCLUDE QUOTES	ON	EXCLUDE SOURCES	OFF
EXCLUDE BIBLIOGRAPHY	ON	EXCLUDE MATCHES	OFF