Misr University of Science and Technology College of Engineering and Technology Department of Mechatronics Engineering

B. Eng. Final Year Project

AUTONOMOUS NAVIGATION ROBOT USING SLAM

Group (_2_)

Prepared By:

Ziad Ahmed Fathy	80786
Lin Mazen Khubieh	80953
Moaaz Taha Ibrahim	80925
Sherif Hassan Ali	80466

Supervised By:

Dr. Bahaa Nasser	Head of Department
Prof. Dr. Mohammed Hamdy	Department professor
Prof. Dr. Mohammed Ibrahim	Menofia University Professor
Prof. Dr. Khaled Mohamed Kamal	Vice-dean of the High Institute for Engineering & Technology
Assist. Prof. Dr. Tahani Wileam	Beni-suef University Assist. Professor
Dr. Alaa Zakria Nasser	Military Technical College
Dr. Bekhet Mohamed	Japanese University

Date of final report

08/07/2023

Acknowledgment

We Want to Thank all our faculty members, for their continuous efforts and support, to improve our project, and in figuring out the missing pieces of the project by their continuous guidance, Including Prof. Bahaa Nasser, Head of the Mechatronics Department at Faculty of Engineering, Prof. Mohamed Ebrahim, Prof. Khaled, Prof. Mohamed Hamdy, and Eng. Waleed Elbadry, at Mechatronics Engineering Department.

DECLARATION

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Bachelor of Science in Mechatronics Engineering is entirely my/our own work, that I/we have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others and to the extent that such work, if any, has been cited and acknowledged within the text of my work.

Signed: _____

Date: 8, July 2023.

Signed: _____

Date: 8, July 2023.

Signed: _____

Date: 8, July 2023.

Signed: _____

Date: 8, July 2023.

ABSTRACT

This project explores the use of Simultaneous Localization and Mapping (SLAM) and Robot Operating System (ROS) in the development of autonomous vehicles with advanced mapping and path planning capabilities. Simultaneous Localization and Mapping is a technique that allows a robot to build a map of its environment while simultaneously keeping track of its location within that map. This is achieved through the use of sensors such as lidar, which gather data about the robot's surroundings. This data is then processed along with the odometry data from the wheel encoders and IMU using advanced algorithms to create a detailed map of the environment.

Robot Operating System is an open-source framework for building robot applications. By integrating Simultaneous Localization and Mapping and Robot Operating System and Navigation2 stack autonomous vehicles can navigate complex environments with greater accuracy and efficiency via different path planning algorithms using the map previously built with Simultaneous Localization and Mapping using pose graph optimization. The body of our vehicle is simple and made from laser-cut acrylic material. It has two wheels with motors and one front castor wheel. This report discusses the technical details of this integration and its potential applications in the field of autonomous navigation.

Table of Contents

Contents

List of Figures

List of Tables

1. Chapter one: Introduction and Literature review

1.1. Introduction

Imagine a world where machines can move people and goods without human intervention. A world where drones deliver packages, robots clean the streets, and trains run on their own tracks. A world where mobility is safer, greener, and more accessible for everyone. This is not a distant future, but a present that is unfolding before our eyes. Welcome to the world of autonomous vehicles, the cutting-edge of technology that is transforming the way we travel and live. and it's closer than you might think. At the heart of this technology are advanced mapping and path planning systems, powered by cutting-edge techniques like Simultaneous Localization and Mapping (SLAM) and the Robot Operating System (ROS).

1.1.1. Why autonomous vehicles?

Autonomous vehicles are machines that can move people and goods without human intervention. They can range from cars and trucks to drones and robots. There are ideal for various applications, such as environmental monitoring, disaster response, and exploration, among others. There are many reasons to use autonomous vehicles. Such as:

Safety [1]: Autonomous vehicles can avoid human mistakes that lead to many crashes and injuries. They can also handle complex and dangerous situations that humans may not be able to.

Efficiency [1][2]: Autonomous vehicles can optimize their performance, reduce their environmental impact, and increase their capacity. They can also coordinate with each other and the infrastructure to reduce traffic and congestion.

Convenience [2][3]: Autonomous vehicles can provide a more pleasant and productive travel experience for users, who can focus on other tasks or activities while the vehicle takes care of everything. They can also offer more mobility options for people who have difficulty driving or accessing transportation.

Innovation: Autonomous vehicles can enable new business models and services, such as delivery drones, robot taxis, or smart city applications. They can also use advanced technologies such as ROS (Robot Operating System) and SLAM (Simultaneous Localization and Mapping) to explore and map unknown environments.

1.1.2. Why ROS?

Modern robotic systems leverage the ecosystem and modularity of the Robot Operating System (ROS) to enhance the development environment and enable realistic simulations. ROS provides a flexible and robust framework that allows developers to create complex robotic systems with ease. This framework provides a wide range of tools and libraries that can be used to develop various robot applications.

1.2. Literature Review

1.2.1. Autonomous concept:

1.2.1.1 Early Concepts and Automata (Ancient Times to 19th Century): [4] [5]

The idea of autonomous machines can be traced back to ancient times, where myths and legends often featured mechanical beings. Ancient civilizations, such as Greece, China, and Egypt, had tales of automatons, mechanical devices capable of performing specific tasks. These early concepts laid the foundation for the development of autonomous machines **(figure 1)**.

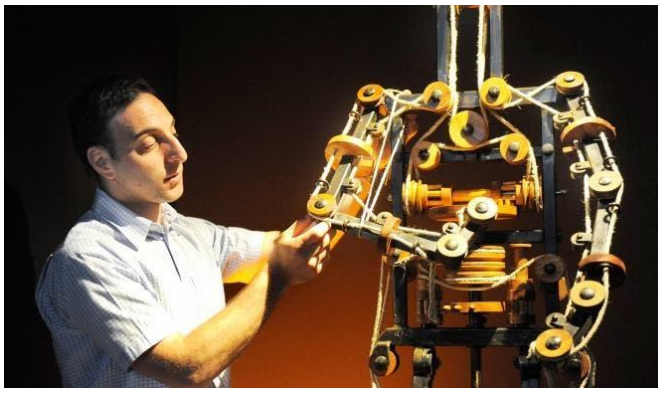


Figure 1 (1.2.1.1) LIFE-SIZED RECREATION OF DA VINCI'S ROBOT.

1.2.1.2. Industrial Revolution and Early Automation (18th to 19th Century): [6]

The Industrial Revolution marked a significant shift in the development of automation. Mechanical devices and machines were created to carry out repetitive tasks, increasing efficiency and productivity in industries such as textile manufacturing and mining. Examples include the automated spinning jenny and the Jacquard loom **(figure 2)**, which utilized punch cards to control weaving patterns.

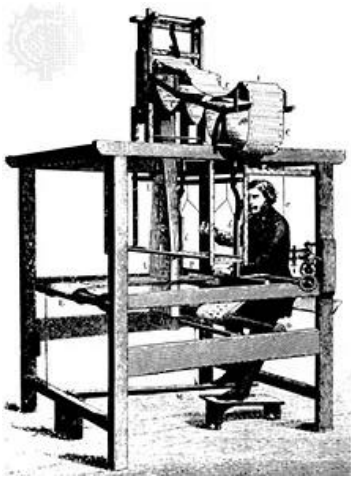


Figure 2 (1.2.1.2) Jacquard loom

1.2.1.3. Early Robotics and Feedback Control (Late 19th to Early 20th Century) [6] [7]:

In the late 19th and early 20th centuries, researchers began exploring the concept of robots as autonomous machines. Nikola Tesla **(figure 3)**, in 1898, demonstrated a remote-controlled boat, which could be considered an early example of autonomous navigation. In 1927, the first teleoperated robot, called the Televox, was introduced by Edouard Belin, which responded to voice commands. This period also saw advancements in feedback control systems, with researchers like Norbert Wiener contributing to the field of cybernetics.

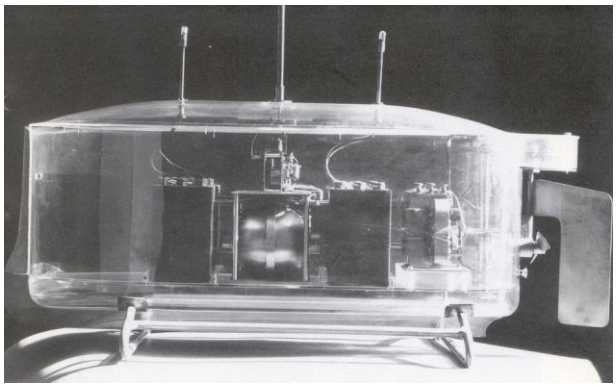


Figure 3 (1.2.1.3) Courtesy of Tesla Collection

1.2.1.4. Cybernetics and Artificial Intelligence (1940s to 1960s): [8] [9]

The development of cybernetics and the birth of artificial intelligence (AI) played crucial roles in shaping the concept of autonomous machines. Cybernetics, pioneered by Norbert Wiener and others, focused on feedback control mechanisms and communication between machines and their environment. In the 1950s, the term "artificial intelligence" was coined, and researchers began exploring the possibilities of creating intelligent machines capable of making decisions and learning.

1.2.1.5. Early Robotics and Autonomous Navigation (1960s to 1970s): [10] [11]

The 1960s and 1970s saw significant progress in the field of robotics. Researchers focused on developing robots that could navigate and interact with their environment autonomously. Notable examples include Shakey **(figure 4)**, developed at Stanford Research Institute in the late 1960s, which could navigate through a room and move objects, and the Stanford Cart, which demonstrated autonomous navigation capabilities.

Figure 4 (1.2.1.5) Shakey

1.2.1.6. Advancements in Sensing and Perception (1980s to 1990s): [12]

In the 1980s and 1990s, advancements in sensing technologies and computer vision played a crucial role in enhancing the autonomy of robots. Researchers explored techniques such as image processing, depth sensing, and laser range finders to enable robots to perceive and interpret their surroundings accurately. This period also witnessed the emergence of simultaneous localization and mapping (SLAM) algorithms **(figure 5)**, which allowed robots to create maps of their environment while simultaneously determining their own position.

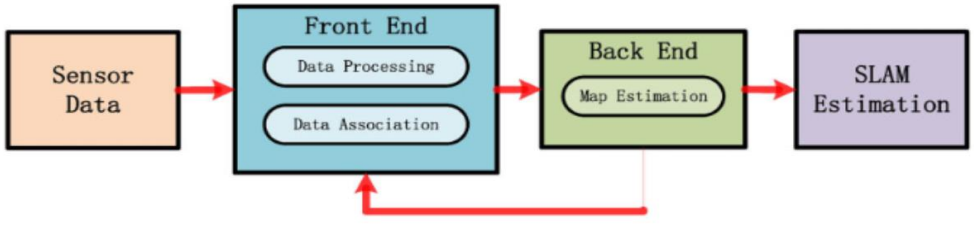


Figure 5 (1.2.1.6) The standard architecture of SLAM

1.2.1.7. Mobile Robots and Field Robotics (2000s to Present): [13] [14]

The 21st century brought significant advancements in mobile robots and autonomous systems. Researchers focused on developing robots capable of operating in unstructured and dynamic environments. Field robotics, which includes applications like autonomous underwater vehicles **(figure 6)** and planetary exploration rovers, advanced our understanding of autonomous navigation, perception, and interaction in challenging terrains and conditions.

Figure 6 (1.2.1.7) AUV

1.2.1.8. Machine Learning and Deep Learning (Recent Developments): [15]

Recent developments in machine learning and deep learning have revolutionized the field of autonomy. Algorithms such as deep neural networks and reinforcement learning have enabled robots to learn from data and make intelligent decisions. These techniques have contributed to advancements in perception, planning, and control, allowing robots to adapt to dynamic environments and improve their autonomy.

1.2.2. Autonomous ground robots' development:

The development of autonomous ground robots has evolved over the years, driven by advancements in sensing, computing, and robotics. Here is a brief overview of the history of autonomous ground robots:

1.2.2.1. Early Explorations (1960s-1980s): [16] [17]

In the 1960s, researchers began exploring the concept of autonomous ground robots. The Stanford Cart, developed in the late 1960s, was one of the earliest examples. It used cameras and sensors to navigate and interact with its environment. In the 1980s, the ALV (Autonomous Land Vehicle) and the Mars rovers Sojourner **(figure 7)** and Pathfinder showcased autonomous navigation and mapping capabilities.

Figure 7 (1.2.2.1) Sojourner

1.2.2.2. Industrial and Military Applications (1990s-2000s): [18][19][20]

During the 1990s and 2000s, autonomous ground robots found applications in industrial and military settings. In industries, robots like the Automated Guided Vehicles (AGVs) were employed for material handling and transportation tasks in warehouses and

Figure 8 (1.2.2.2) PackBot

manufacturing facilities. The military sector saw the development of robots like the iRobot PackBot and the Foster-Miller Talon, used for bomb disposal and reconnaissance missions.

1.2.2.3. Urban Challenges and Competitions (2000s-Present): [21]

The early 2000s marked the emergence of robotic competitions focusing on autonomous ground robots. The DARPA Grand Challenge **(figure 9)**, held in 2004 and 2005, aimed to develop autonomous vehicles capable of navigating a desert course. These competitions accelerated the development of perception, navigation, and planning algorithms for autonomous ground robots.

Figure 9 (1.2.2.3) the proposed vehicle for team

1.2.2.4. Field Robotics and Exploration (2000s-Present):[22][23]

Autonomous ground robots have played a vital role in field robotics and exploration. The exploration rovers Spirit and Opportunity, launched by NASA in 2003, autonomously navigated the Martian surface for years, collecting valuable data. Field robots like the Boston Dynamics' BigDog and Spot have been developed for rugged terrains and have been used in applications like search and rescue, agriculture, and inspection tasks.

Figure 10 (1.2.2.4) BigDog

1.2.3. Examples of Autonomous Ground Robots:

1.2.3.1. Boston Dynamics' Spot:[24] **(figure 11)** Spot is a quadruped robot designed for various applications. It can traverse challenging terrain, climb stairs, and navigate through complex environments. Spot has been used in industries such as construction, oil and gas, and public safety.

Figure 11 (1.2.3.1) Spot

1.2.3.2. Clearpath Robotics' Husky:[25] **(figure 12)** Husky is an unmanned ground vehicle (UGV) designed for research and development. It provides a rugged platform for testing perception, navigation, and planning algorithms. Husky is widely used in academia and research institutions.

Figure 12 (1.2.3.2) Husky

1.2.3.3. Autonomous Cars:[26] **(figure 13)**

Autonomous cars, such as those developed by Waymo, Tesla, and other companies, are a prime example of autonomous ground robots. They use a combination of sensors, AI algorithms, and mapping systems to navigate and drive safely on roads, with the goal of revolutionizing transportation.

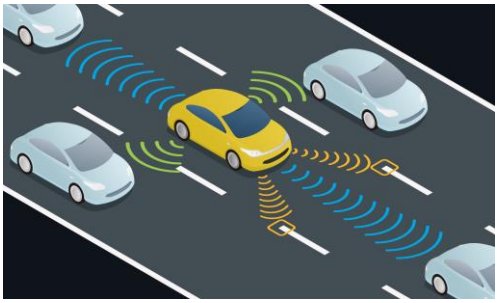


Figure 13 (1.2.3.3) autonomous cars

1.2.3.4. Starship Technologies' Delivery Robots:[27] **(figure 14)** Starship Technologies has developed autonomous ground robots for last-mile delivery. These small wheeled robots navigate sidewalks and deliver packages, groceries, and food autonomously, aiming to optimize and streamline the delivery process.

Figure 14 (1.2.3.4) Starship delivery robot

1.2.3.5. Autonomous Agricultural Robots:[28] **(figure 15)** Various companies and research organizations are developing autonomous ground robots for agriculture. These robots can perform tasks such as planting, harvesting, and crop monitoring. They leverage sensors, computer vision, and AI algorithms to optimize farming operations.

Figure 15 (1.2.3.5) autonomous agricultural robot "Rosei"

1.2.4. Conclusion:

The concept of autonomy has a long and rich history that goes back to centuries ago and has been influenced by various disciplines, including engineering, computer science, and cognitive science. These fields have contributed to the development and innovation of machines that can behave and make choices based on their own observation and understanding of the environment. Continuous advancements in technology and the integration of AI have enabled autonomous robots to improve and diversify in various aspects, such as sensing, learning, and thinking. These developments have also opened up new possibilities for the applications of autonomous machines in various industries and domains, such as manufacturing, transportation, healthcare, and entertainment.

2. Chapter Two: Project Design

2.1. Project Purpose

The purpose of autonomous ground robots is to perform tasks and operate independently without direct human intervention. These robots are designed to navigate and interact with their environment, make decisions, and execute actions based on their programming and sensor inputs. The main purpose of autonomous ground robots is to augment human capabilities, increase efficiency, improve safety, and enable automation in diverse domains. Their deployment leads to increased productivity, cost savings, and the ability to perform tasks in challenging environments, ultimately transforming industries, and improving our quality of life.

2.2. Advantages of Ground Autonomous robots:

2.2.1. Automation and Efficiency:

[29] Autonomous ground robots are deployed to automate repetitive and labor-intensive tasks, increasing productivity and efficiency in various industries. They can perform tasks such as material handling, inspection, cleaning, and logistics, reducing human effort and improving workflow.

2.2.2. Safety and Hazardous Environments:

[30] Autonomous ground robots are well-suited for operating in hazardous or dangerous environments, where human presence may pose risks. They can navigate through areas with high temperatures, radiation, toxic substances, or unstable terrain, enabling safer operations in industries like mining, nuclear facilities, and disaster response.

2.2.3. Exploration and Mapping: [31] Autonomous ground robots are used for exploration and mapping purposes. They can navigate through uncharted terrains, collect data, and create maps of the environment. These robots have been utilized in space exploration, planetary rovers, deep-sea exploration, and mapping in rugged or inaccessible areas.

2.2.4. Surveillance and Security: [32] Autonomous ground robots serve as effective tools for surveillance and security applications. Equipped with cameras, sensors, and AI algorithms, these robots can patrol and monitor areas, detect intrusions or suspicious activities, and provide real-time data to security personnel. They are employed in facilities, borders, airports, and public spaces for enhanced security.

2.2.5. Transportation and Delivery: [33] Autonomous ground robots are being developed for transportation and delivery purposes. They can navigate roads or pedestrian pathways autonomously, delivering goods, packages, or food in a timely and efficient manner. These robots have the potential to revolutionize last-mile delivery and logistics, reducing costs and improving convenience.

2.2.6. Assisting and Supporting Humans: [34] Autonomous ground robots can assist humans in various tasks, providing support and enhancing capabilities. They can act as companions for the elderly or people with disabilities, help with household chores, provide assistance in healthcare settings, and support workers in industries by carrying heavy loads or performing repetitive tasks.

2.3. Project Constraints

2.3.1. The previous project

At first our project was about SWARM robotics, the target was 6 small scaled wheeled robots equipped with IR sensors and wheel encoders to get odometry measurements of the robots, also using the IR Sensors in the local communication between the robots, which are organized on the circumference of the robot separated by 45^{o} angle increments. Controlling the robots using Robot Operating System (ROS), to manage fleet communication.

2.3.1.1. Time constrains.

The project needed deep knowledge in ROS, in controlling a one robot first then understanding communication protocols for slicing the ROS messages. This took a lot of time in developing knowledge in practical robotics and ROS, taking several courses and applying to several small projects with SLAM and path planning. After the first semester this showed an alarm that we may face a problem in applying our theoretical knowledge via simulation to a one real robot, and this is a larger problem for several robots.

2.3.1.2. Budget Constraints

Since our project needed six robots, we were trying to achieve the goal with lower end hardware that we didn't have the needed budget for higher end hardware, but when we implemented these specifications to the simulation, the results were too inaccurate, seeing this we decided to change the project.

2.3.1.3. Technical Constraints

As we were making several robots, we faced a problem that we are forced to use lower end hardware to keep the costs feasible, mainly using simple microcontrollers and sensors. Specially, using a microcontroller instead of a regular on-board computer such as raspberry pi, makes the computing resources very limited. This made us forced not to use ROS directly on each robot but needing to be slicing the messages in the sending and receiving from the robots to the master computer, and this made the whole process very complex. Many of the swarm robot's projects use a raspberry pi to avoid this problem.

Another problem is the small mechanical bodies for the cost, which made it hard to find suitable small motors with encoders that we can use for the odometry. Along with the encoders having IR sensors instead of a proper depth camera or a lidar introduced many issues in filtering the readings to be at least usable. Another constraint was the lower number of resources to learn swarm robotics, especially that it is just having its first steps in the market after being only a research topic that is in development. This made it harder to find some examples to learn on or even the knowledge to do the basic communication tasks.

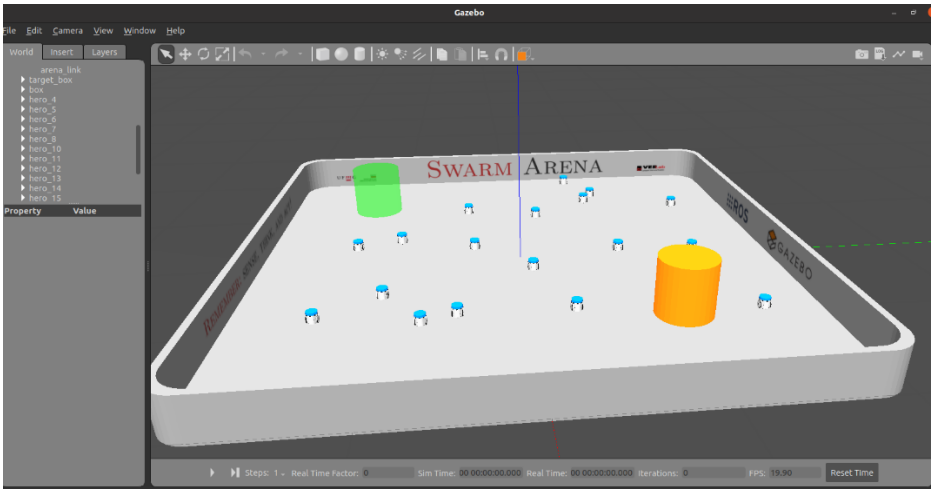


Figure 16 (2.3.1.3) Swarm Robots Representation

2.3.2. The current project

2.3.2.1. Parts availability & selection

- The available motors we found weren't strong enough to carry our chasse, so we were limited and had to lower the body weight and distribute the inside components in the best way to balance the robot.
- To balance the robot, we needed to add a castor wheel in the front. At first we chose the castor wheel in **(figure 17)**, but we faced a problem that when the robot change its direction the robot slip further than the desired change, so we couldn't control the robot in a precise way.

Figure 17 (2.3.1.1.a) Castor wheel

- We changed the first castor wheel to a ball castor wheel **(figure 18)**, we face another problem that the friction of the ball castor was a little bigger than the first one. But we didn't have another choice so we decided to stick with it.

Figure 18 (2.3.1.1.b) Ball castor wheel

2.3.2.2. Time constraints.

One of the main challenges we faced in this project was the change of topic in the second semester, which required us to start from scratch in mechanical and electrical regions in a short period of time. This also affected our testing process, as we could not follow the original plan and had to adjust to the new requirements and specifications. Despite the tight schedule and the unexpected changes, we are committed to making further improvements after graduation.

2.4. Project specifications

We are building a $40 \times 30 \times 8\ cm$ wheeled robot **(figure 19)** equipped with LIDAR sensor and wheel encoders to get odometry measurements of the robot. The chasse is laser cut acrylic. We have two wheels with motors and one castor wheel. Using a raspberry pi on-board computer along with an Arduino nano to control the motors motion and to execute basic control commands then to use Simultaneous Localization and Mapping (SLAM) Algorithm, to construct a map, then using path planning for the autonomous navigation.

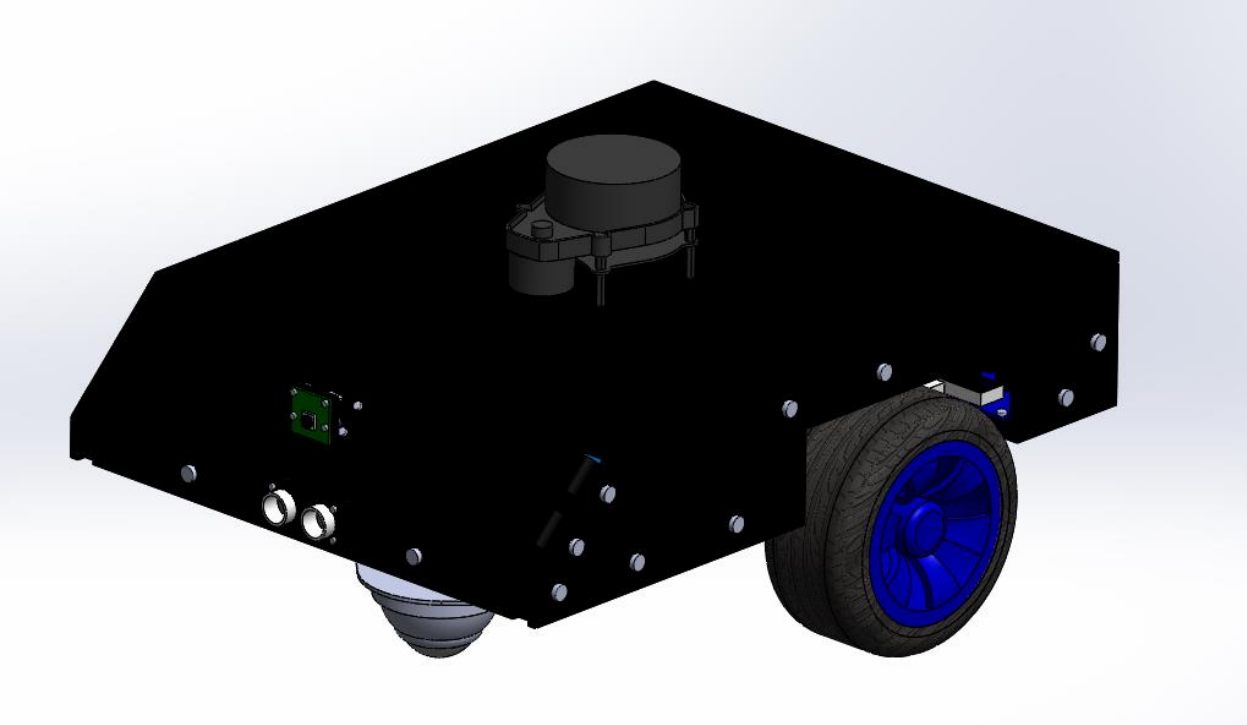


Figure 19 (2.4) Overview of the body (Isometric view)

2.5. Description of the selected design

2.5.1. Mechanical Design

For our design, we will be using a laser cut body, using acrylic as a material.

2.5.1.1. Acrylic material:

2.5.1.1.1. Advantages:[35][36]

1. Excellent optical clarity & transparency (if needed).
2. Highly resistant to variations in temperature.
3. Up to 17 times the impact resistance of ordinary glass.
4. Light weight.
5. Highly resistant to many different chemicals.
6. Easy to maintain.
7. Easy to process and shape.
8. Excellent dimensional stability.

2.5.1.1.2. Properties: [36] [37] [38]

Property	Value
Technical name	Acrylic (PMMA)
Relative Density	1.19 g/cm3
Minimum Service Temperature	-40ºC
Maximum Service Temperature	80ºC
Melt Temperature	130°C (266°F)
Chemical Formula	(C5H8O2) n
Tensile Strength	65 MPa (9400 PSI)
Shrink Rate	0.2 – 1% (.002 – .01 in/in)
Flexural Strength	90 MPa (13000 PSI)
Specific Gravity	1.18

Typical Injection Mold Temperature	79-107°C (175-225°F)
Water absorption (immersion 24 hours)	0.20%
Light Transmittance	92%
Heat deflection temperature	95°C (203°F) at 0.46 MPa (66 PSI) **

Table 1 (2.5.1.1.2) Properties of acrylic

2.5.1.2. CAD Model

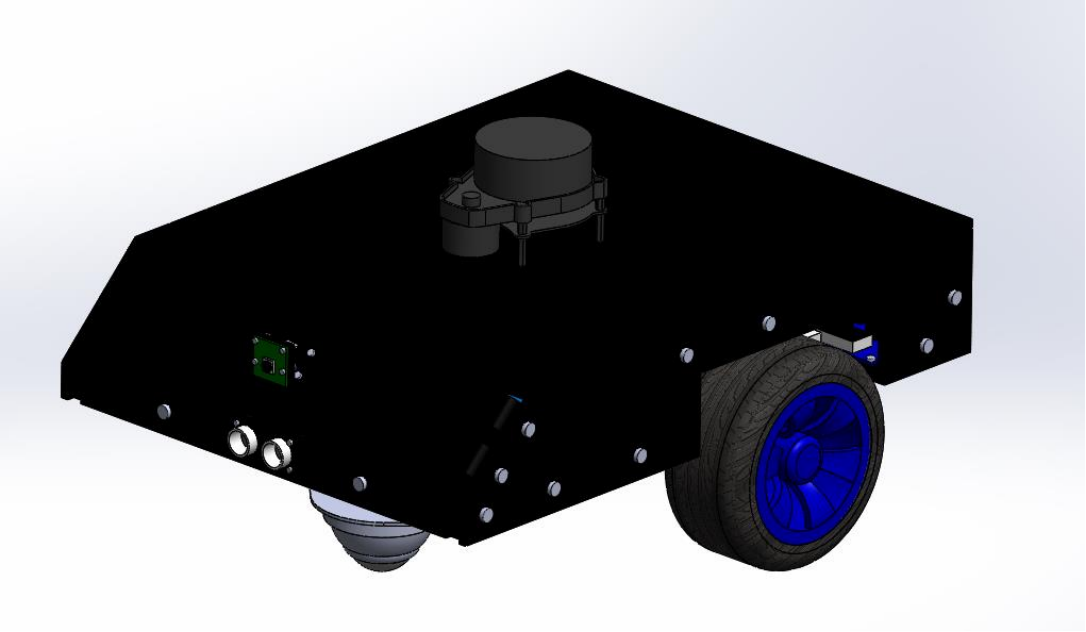


Figure 20 (2.5.1.2.a) Isometric View of The Robot

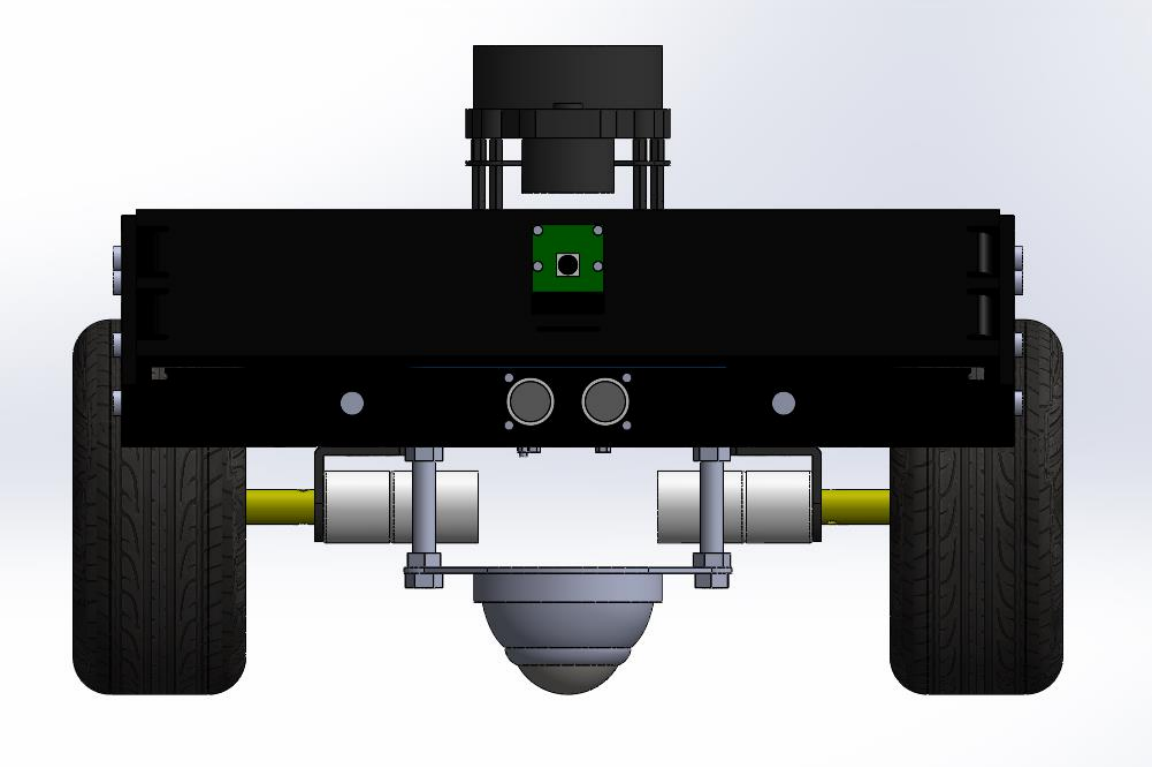


Figure 21 (2.5.1.2.b): Front view of the Robot

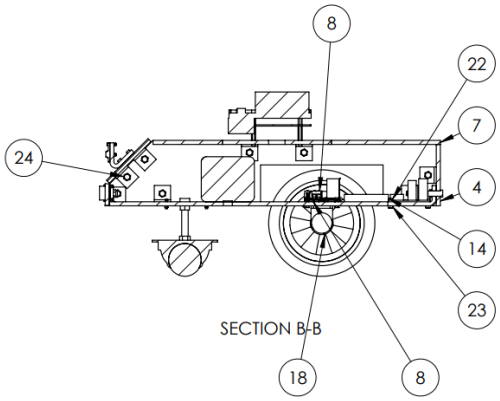


Figure 23 (2.5.1.2.d): Side View with components number

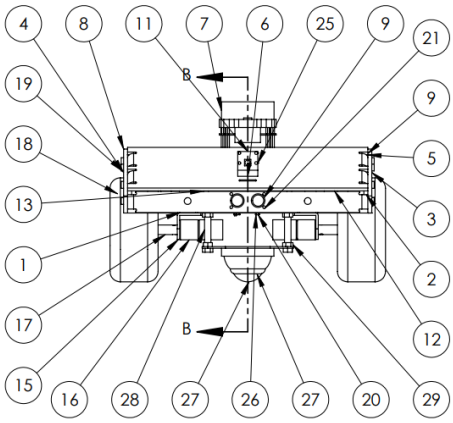


Figure 22 (2.5.1.2.c): Front View with components number

ITEM NO.	PART NUMBER	QTY.
1	Bottom chassis	1
2	Front chassis	1
3	Side chassis	2
4	Back chassis	1
5	Top chassis	4
6	Camera holder	1
7	LIDAR	1
8	L298N Stepper Motor Driver	1
9	Ultrasonic	1
10	Raspberrypi	1
11	Camera	1
12	Converter	2
13	Battery	1
14	breadboard	2

Table 2 (2.5.1.2.b) Parts of the robot that are addressed in figure (2.5.1.2.c) & figure (2.5.1.2.d)

15	Motor_Mount	2
16	motor	2
17	Coupler	2
18	Wheel	2
19	Screw6mm	23
20	Screw3mm	16
21	Screw2mm	12
22	Screw4mm	6
23	Bolt4mm	6
24	Bolt6mm	23
25	Bolt2mm	10
26	Bolt3mm	8
27	CastorWheel	1
28	CastorScrew	2
29	CastorBolt	6

Table 3 (2.5.1.2.a) Parts of the robot that are addressed in figure (2.5.1.2.c) & figure (2.5.1.2.d)

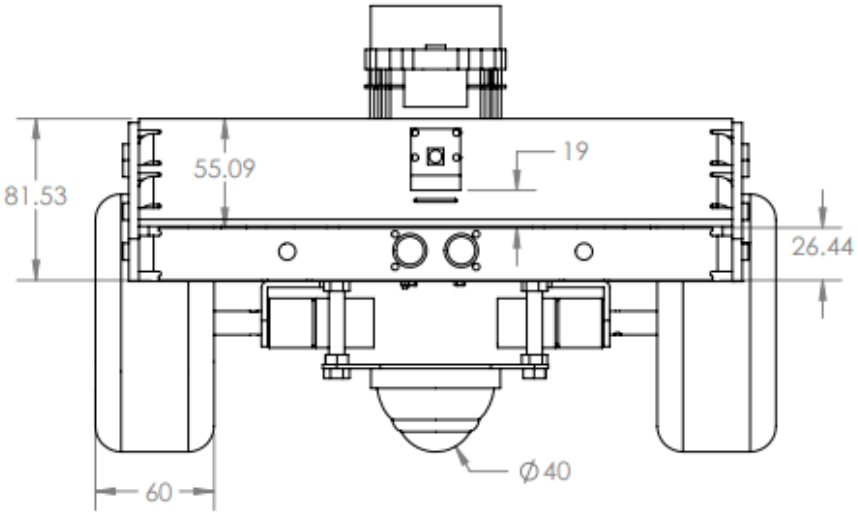
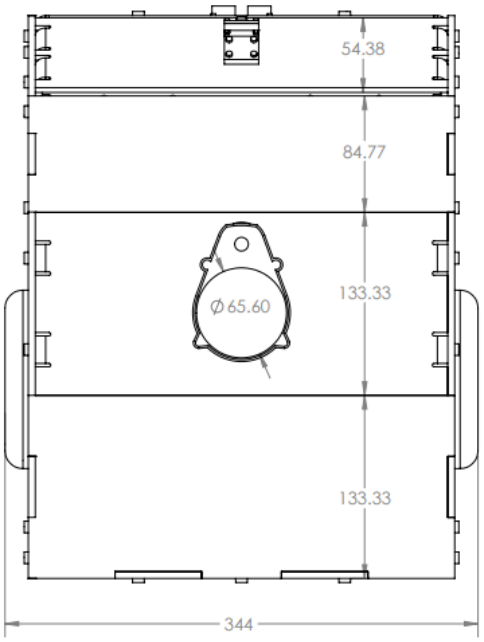


Figure 24(2.5.1.2.e): Front view of the Robot with Dimensions



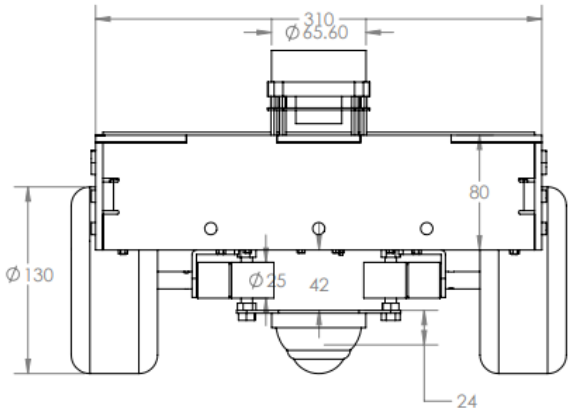


Figure 25 (2.5.1.2.f): Robot Back view

Figure 26(2.5.1.2.g): Top view of the Robot with Dimensions

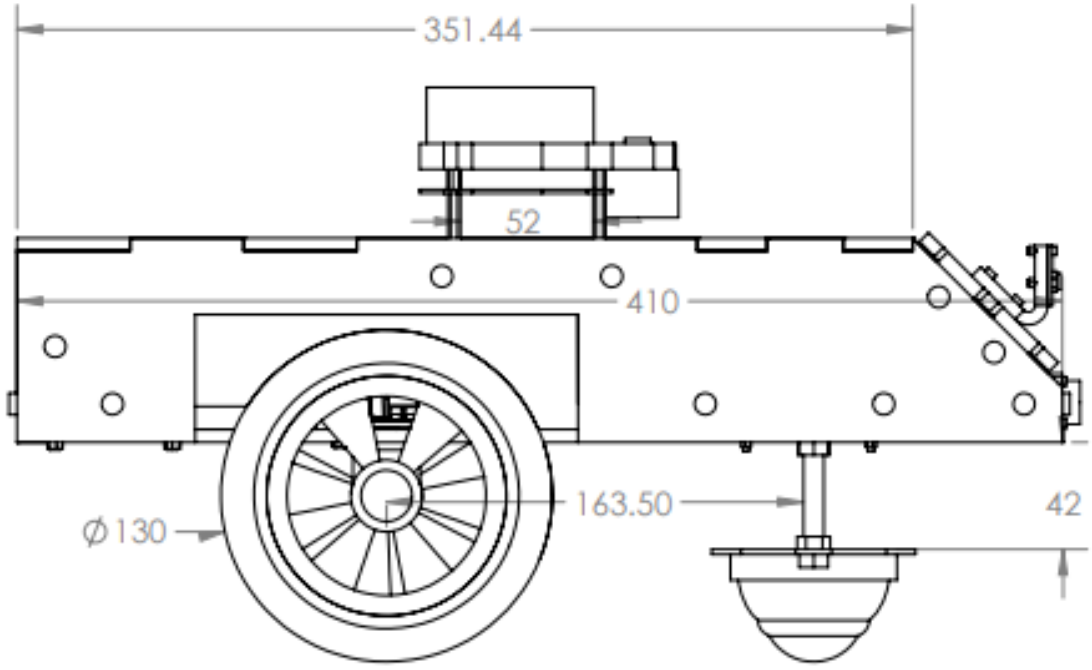


Figure 27 (2.5.1.2.): Robot side view

2.5.1.3. Kinematics

2.5.1.3.1. Generals

The calculations determining a robot's position using odometry are an example of kinematics, the study of motion without considering the forces causing it. Kinematics ignores concepts like torque, force, mass, energy, and inertia. It is a simplified form of physics focusing only on time and geometry. Kinematics is important in robotics, especially for robot arms and industrial robots. There are two kinds of kinematics: forward kinematics and inverse kinematics.

Forward kinematics calculates the position and orientation of a robot's end-effector given the angles of its joints. Inverse kinematics does the opposite: it calculates the joint angles needed to achieve a desired position and orientation of the end-effector. Robot arms use inverse kinematics to determine the joint angles required to move the end-effector to a target location and orientation.

2.5.1.3.2. Our robot

Our robot type is differential-driven model. In a nutshell, a differential wheeled robot is a mobile robot whose movement is based on two separately driven wheels placed on either side of the robot body. The robot changes its direction by varying the relative speeds of its wheels, and therefore it does not require an additional steering motor. The robot has two motor-wheel and one castor wheel.

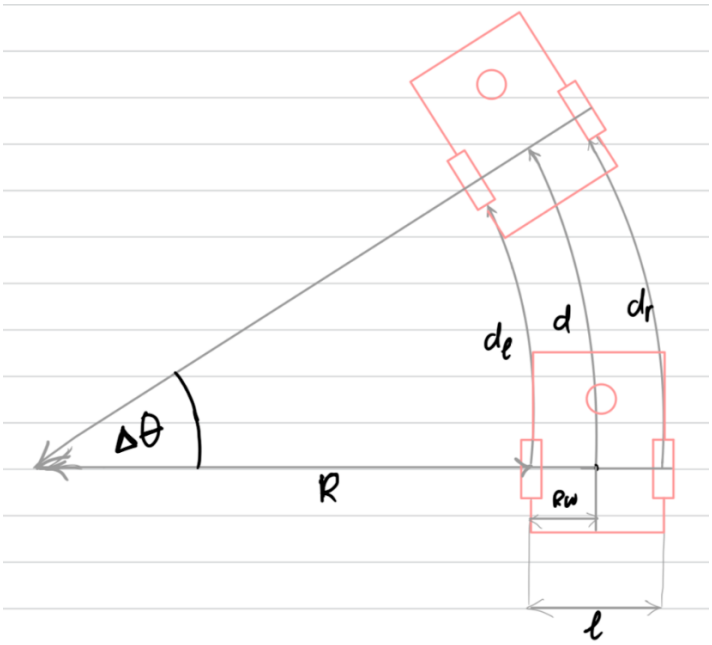


Figure 28 (2.5.1.3.a) Motion

From **figure 28**, by using circular arc formula we know that:

$$d_l = R\Delta\theta \text{ ---(1)}$$

$$d = (R + R_W)\Delta\theta$$

$$d_r = (R + 2R_W)\Delta\theta \text{ ---(2)}$$

$$\therefore \boldsymbol{from\ 1\ \&\ 2}:$$

$$\Delta\theta R_W = \frac{d_r - R\Delta\theta}{2} \text{ ---(3)}$$

$$\therefore \Delta\theta = \frac{d_r - d_l}{2R_W}$$

By the previous equations, we know that the distance traveled by the robot is computed by the average of how much each wheel has turned and is presented in:

$$d = (R + R_W)\Delta\theta = R\Delta\theta + \Delta\theta R_W$$

$$\therefore \boldsymbol{from\ 1\ \&\ 3}:$$

$$d = d_l + \frac{d_r \_ d_l}{2}$$

$$\therefore\ d = \frac{d_r + d_l}{2}$$

Now, after we understand the motion equations, let's calculate the next position.

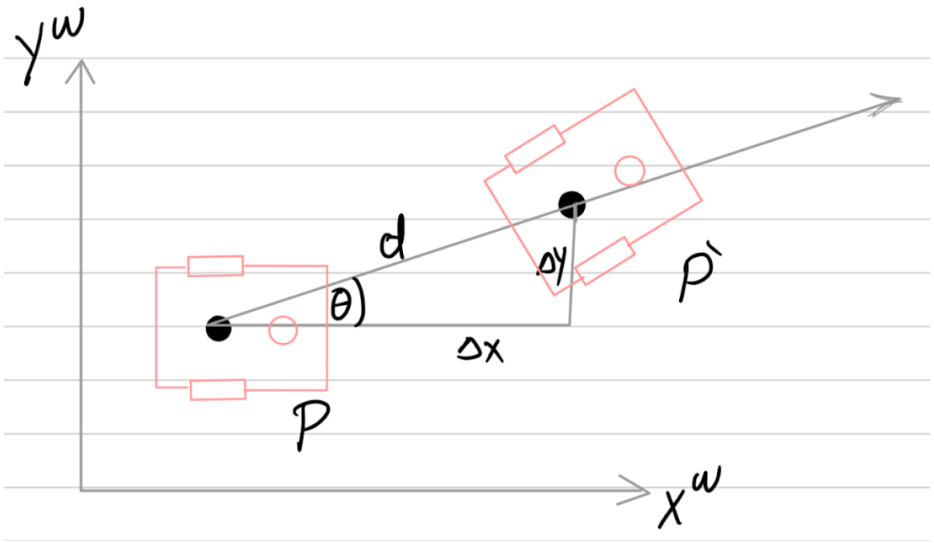


Figure 29 (2.5.1.3.b) diagram showing the next position.

So, the meantime position is P and the next position is $\grave{P}$, from **figure 29** we know that:

$$\Delta x = d\cos\theta$$

$$\Delta y = d\sin\theta$$

$$\therefore \grave{P} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} d\cos\theta \\ d\sin\theta \\ \Delta\theta \end{bmatrix}$$

Previously, we defined the robot as a two-wheel differential-drive mobile robot composed of two motors. The robot can change its direction by varying the relative rotation rate of its wheels and hence does not require an additional steering motor.

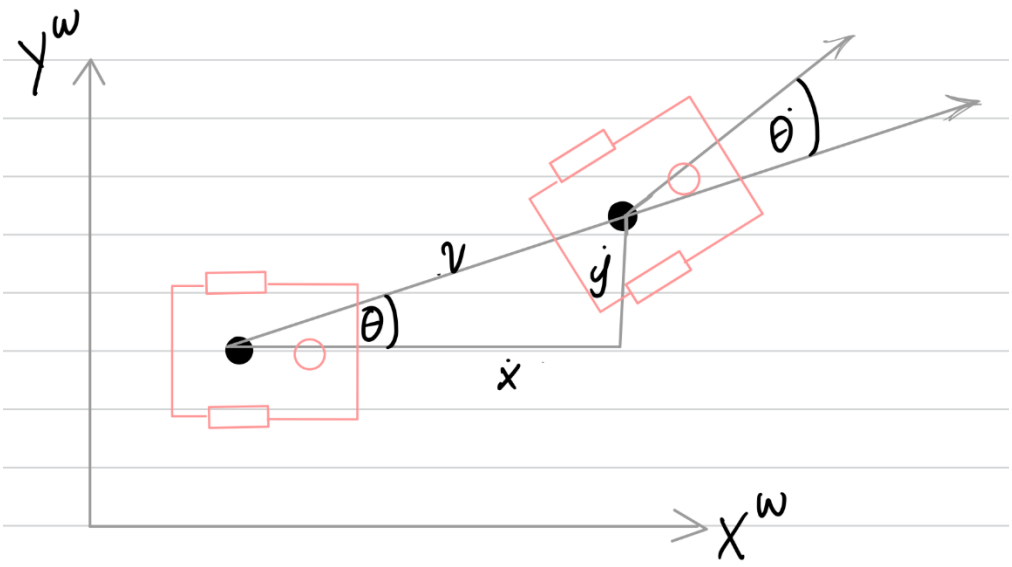


Figure 30 (2.5.1.3.c) Velocity diagram

From **figure 30**, we know that:

$$\dot{x} = v\cos\theta$$

$$\dot{y} = v\sin\theta$$

$$\therefore v = \frac{v_l + v_r}{2}$$

$$\because \dot{\theta} = r\Delta\theta$$

$$\therefore \dot{\theta} = \frac{r}{2R_W}(v_r - v_l)$$

$$\therefore \dot{v} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \frac{r}{2R_W}(v_r - v_l) \end{bmatrix}$$

2.5.1.4. Stress Analysis

2.5.1.4.1. Calculations

	Mass (g)	Weight (N)	Area (mm^2)
Acrylic (Chassis)	1563.96	15.327	_
Battery	1000	9.8	_
Motor	500	4.9	_
Wheel	72	0.7056	_
Bottom chasse area	_	_	127100

Table 4 (2.5.1.4.1) Masses & Weights of the robot parts

Our robot has many parts, their weight affect the chasse directly. The chasse carry many things, but there will many neglected things when we calculate the stress due to its slight weight, the things that we will consider will be the two motos, The battery, and the weight of the chasse itself, the parameters are illustrated in **(table 4)**, therefore, here are our calculations:

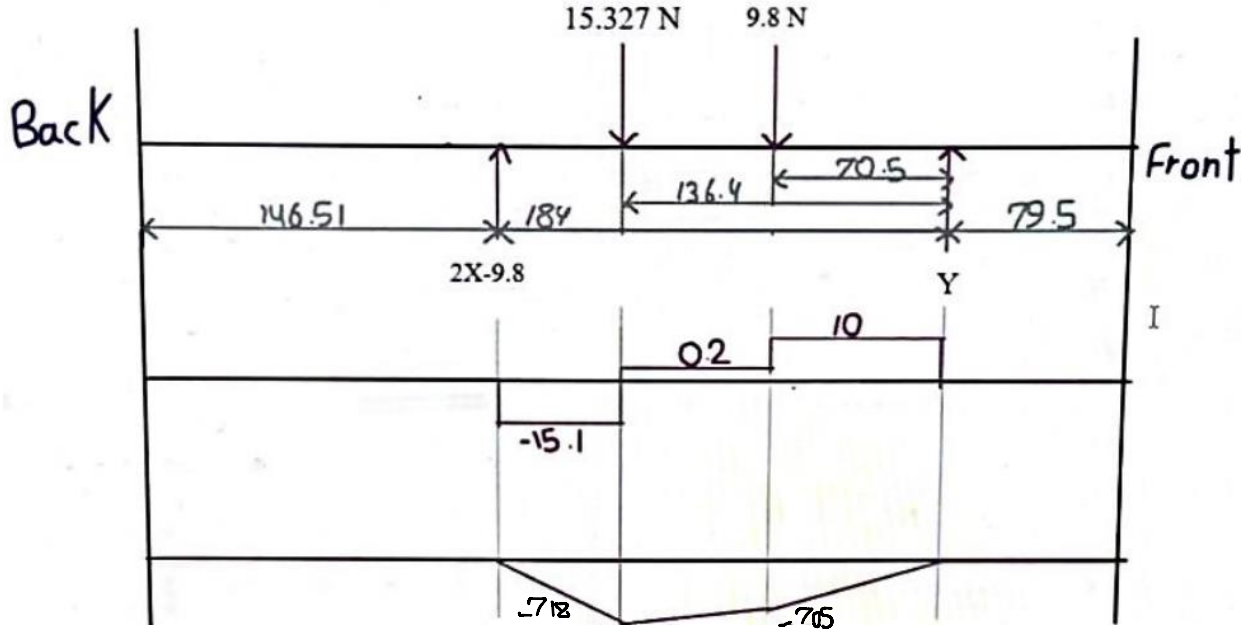


Figure 31 (2.5.1.4.1) Loads distribution

Knowing that:

- X is the reaction of one back wheel.
- Y is the reaction of the castor wheel.
- 9.8 N is the weight of two motors.
- The other 9.8 N is the weight of the battery.

$$\sum Moment_y = zero$$

$$\therefore 9.8(70.5) + 15.327(136.4) - (2X - 9.8)(184) = zero$$

$$\therefore X = 12.4584N\#$$

$$\sum Forces = zero$$

$$\therefore 2(12.4584) - 9.8 + Y = 25.127\ N$$

$$\therefore Y = 10\ N\#$$

$$\sigma = \frac{F}{A} = \frac{34.927}{127100} = 2.75 \times 10^{-4}\ N/mm^2$$

$$\varepsilon = \frac{\sigma}{E} = \frac{2.75 \times 10^{-4}}{2760} = 9.957 \times 10^{-8}$$

$$\delta = \varepsilon l = 9.957 \times 10^{-8} \times 5 = 4.976 \times 10^{-7} mm$$

2.5.2. Electrical Circuit Design

2.5.2.1. Components list and power consumption:

2.5.2.1.1. Components list

Component	Number of items	Approx. Price (EGP)
Arduino Nano	1	220
Raspberry PI 4B	1	5000
Lidar	1	6000
DC Motor (DC Gear Motor GA25-370 12V. 200 RPM, 1.43 kg.cm rated torque, 4.5 kg.cm stall torque and rotary encoder 12 pulse / rev.)	2	770
Motor Driver	2	130
Battery (Lithium – Ion Battery 12 Volt and 10000 mAh)	1	800
XL4016 DC-DC Step down Converter Module	1	75
Mech & electrical components		300
IMU (MPU 6050Acce & Gyro)	1	120
Raspberry PI Camera	1	850
Total		**14265**

Table 5 (2.5.2.1.1) Components list

2.5.2.1.2. Power consumption:

Component	Stall current (mA)
DC Motor	1500
Arduino Nano	500
MPU 6050 (IMU)	3.9
Raspberry PI 4B	3000
Raspberry PI Camera 8 MP	250
Lidar	600

Table 6 (2.5.2.1.2.a) Stall current of components

Component	Stall voltage (V)
DC Motor	12
Arduino Nano	5
MPU 6050 (IMU)	3.6
Raspberry PI 4B	5.25
Raspberry PI Camera 8 MP	1.8
Lidar	10

Table 7 (2.5.2.1.2.b) Stall voltage of components

So, from the previous tables we will calculate the power consumption in (Ah) by multiplying the stall voltage with stall current with time (1 hour) with safety factor (1.3). and finally dividing the result by 5 volts (except for motors we will divide by 12 V) which is the voltage used.

Components	Description	Max(mAh)
2 Motor	$E_{total} = \frac{2 \times 12V \times 1500mA \times 1h \times 1.3}{12V}$	3900
Arduino Nano	$E_{total} = \frac{5V \times 500mA \times 1h \times 1.3}{5V}$	650
MPU 6050	$E_{total} = \frac{3.5V \times 39mA \times 1h \times 1.3}{5V}$	36.50
Raspberry PI 4B	$E_{total} = \frac{5.25V \times 3000mA \times 1h \times 1.3}{5V}$	4305.50
Raspberry PI Camera	$E_{total} = \frac{1.8V \times 100mA \times 1h \times 1.3}{5V}$	47
Lidar	$E_{total} = \frac{5V \times 600mA \times 1h \times 1.3}{5V}$	780
Total		9719

Table 8 (2.5.2.1.2.c) Power consumption

2.5.2.2. The most important components

2.5.2.2.1. Raspberry PI 4B [62]

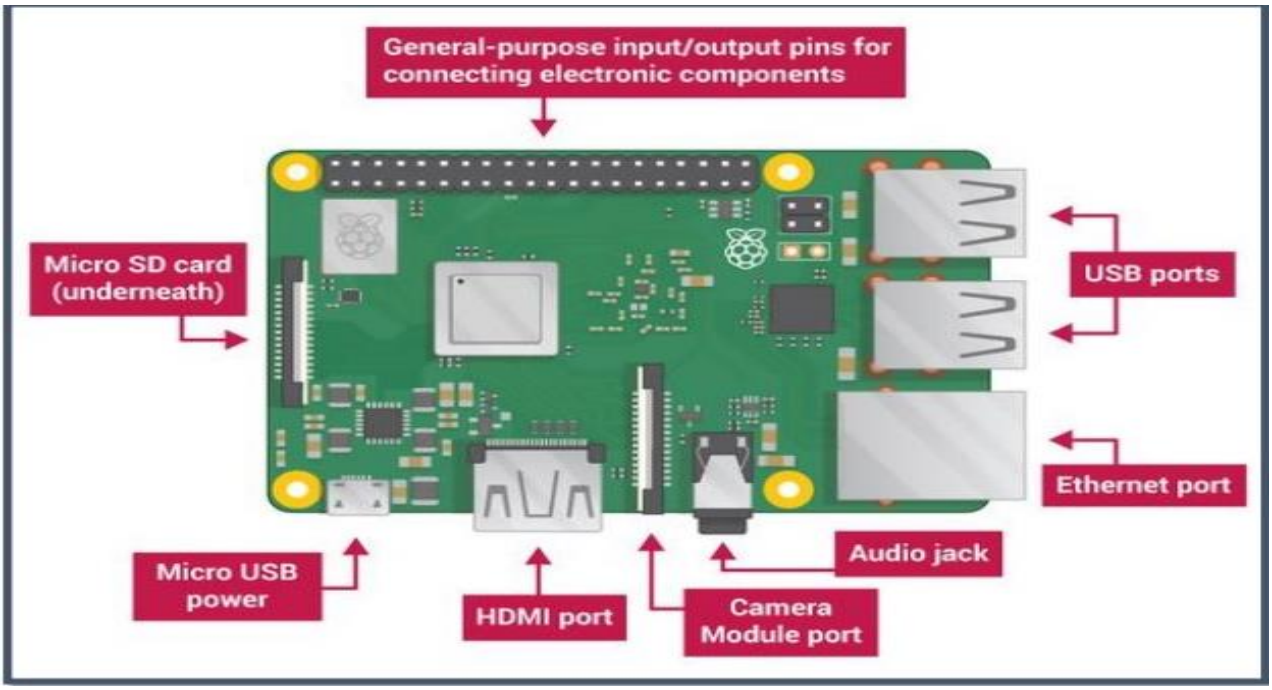


Figure 32 (2.5.2.2.1) Raspberry PI 4

The Raspberry Pi **(figure 32)** is a popular single-board computer. It is compact, affordable, and versatile. It supports various operating systems, has GPIO pins for connecting external devices, and offers connectivity options for different projects.

Raspberry Pi 4 GPIO Pinout has 40 pins: 26 GPIO pins, two 5V pins, two 3V3 pins, and 7 ground pins (0V). GPIO pins of Raspberry PI 4 can generate PWM output, and the board supports SPI, I2C, and UART serial communication protocols.

2.5.2.2.1.1. Specifications

To be more specific we will divide the specifications of Raspberry PI 4 into two aspects hardware and software (interface)

I. Hardware specifications:

- Quad core 64-bit ARM-Cortex A72 running at 1.5GHz
- 1, 2 and 4 Gigabyte LPDDR4 RAM options
- H.265 (HEVC) hardware decode (up to 4Kp60)
- H.264 hardware decode (up to 1080p60)
- Video Core VI 3D Graphics
- Supports dual HDMI display output up to 4Kp60

II. Software / Interface specifications:

- 802.11 b/g/n/ac Wireless LAN
- Bluetooth 5.0 with BLE
- 1x SD Card
- 2x micro-HDMI ports supporting dual displays up to 4Kp60 resolution
- 2x USB2 ports
- 2x USB3 ports
- 1x Gigabit Ethernet port (supports PoE with add-on PoE HAT)
- 1x Raspberry Pi camera port (2-lane MIPI CSI)
- 1x Raspberry Pi display port (2-lane MIPI DSI)
- 28x user GPIO supporting various interface options:

– Up to 6x UART – Up to 6x I2C

– Up to 5x SPI – 1x SDIO interface

– 1x DPI (Parallel RGB Display) – 1x PCM

– Up to 2x PWM channels – Up to 3x GPCLK outputs

2.5.2.2.1.2. Raspberry PI 4 Pinout & interface

Raspberry Pi GPIO stands for General Purpose Input Output pins. These pins are used to connect the PI board to external input/output devices.

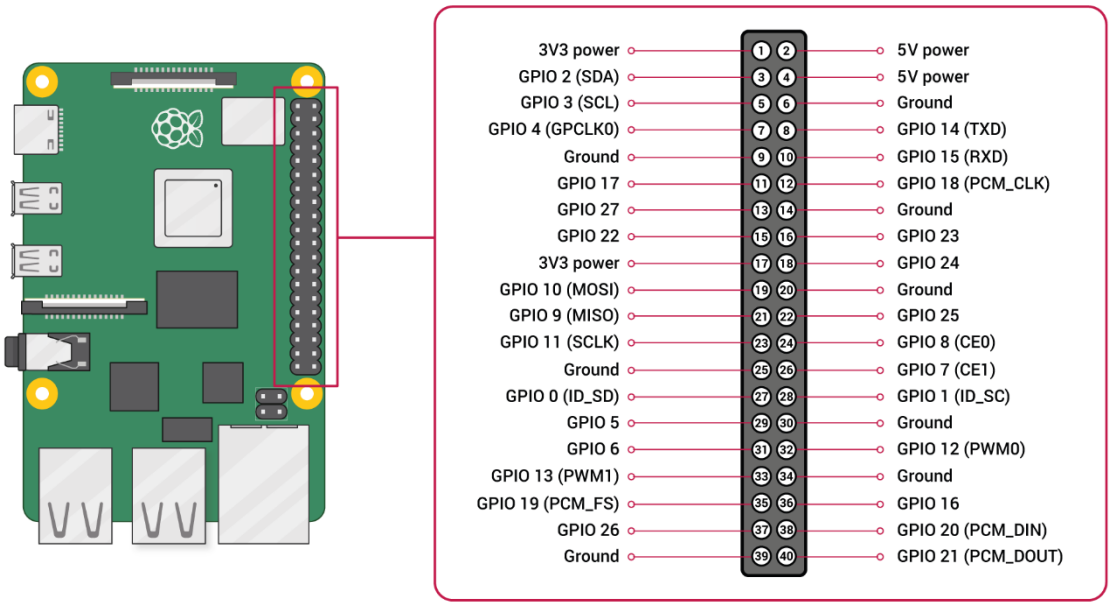


Figure 33 (2.5.2.2.1.2) Raspberry PI 4 GPIO

Pin Type	GPIO Pins
PWM pins	GPIO12, GPIO13, GPIO18, GPIO19
SPI pins	**SPI0:** GPIO9 (MISO), GPIO10 (MOSI), GPIO11 (SCLK), GPIO8 (CE0), GPIO7 (CE1) **SPI1:** GPIO19 (MISO), GPIO20 (MOSI), GPIO21 (SCLK), GPIO18 (CE0), GPIO17 (CE1), GPIO16 (CE2)
I2C Pins	**Data:** (GPIO2), **Clock:** (GPIO3) **EEPROM Date:** (GPIO0), **EEPROM Clock:** (GPIO1)
UART Pins	**TX:** (GPIO14) **RX:** (GPIO15)

Table 9 (2.5.2.2.1.2) GPIO pins types

2.5.2.2.2. IMU (MPU 6050)

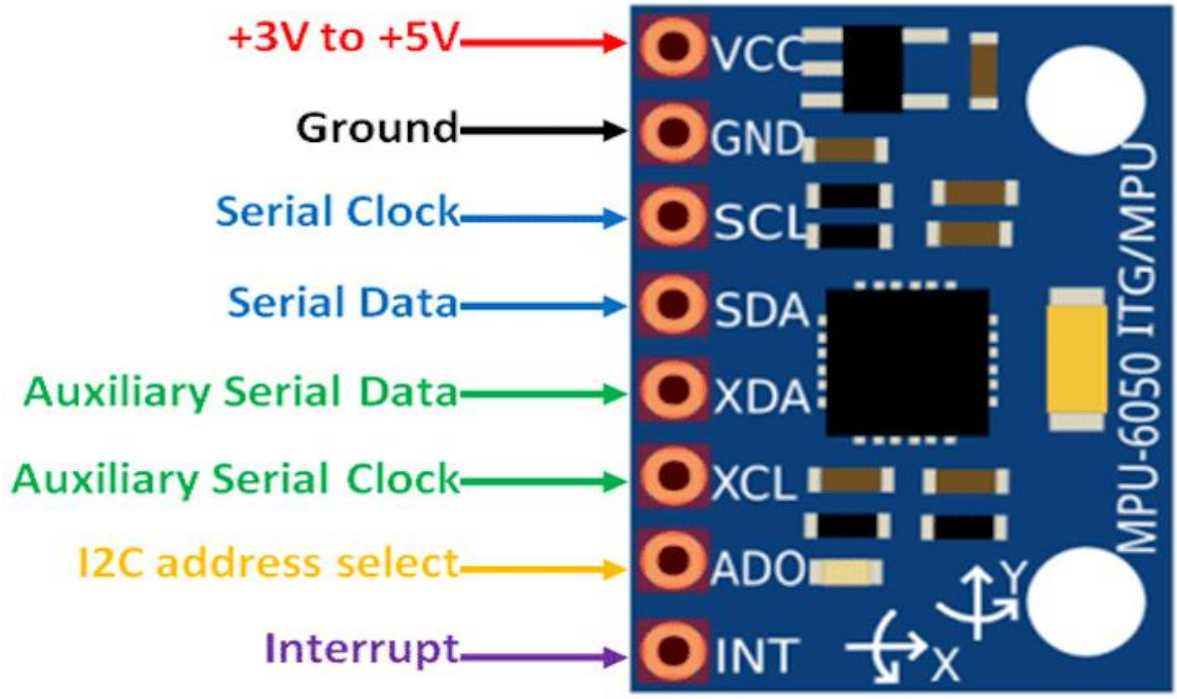


Figure 34 (2.5.2.2.2.a) MPU-6050 IMU

An Inertial Measurement Unit (IMU) is a device that can measure and report specific gravity and angular rate of an object to which it is attached. An IMU typically consists of:

- Gyroscopes: providing a measure angular rate
- Accelerometers: providing a measure specific force/acceleration
- Magnetometers (optional): measurement of the magnetic field surrounding the system.

The addition of a magnetometer and filtering algorithms to determine orientation information results in a device known as an Attitude and Heading Reference Systems (AHRS).

In our robot, we used MPU 6050 which contains a MEMS accelerometer and a MEMS gyro in a single chip. We used IMU to measure acceleration and angular

velocity. So, it can help in the estimation process as we know that displacement is the integral of integral of acceleration. And on the other hand, the heading angle of robot can be calculated from integrating the angular velocity measured by gyroscope.

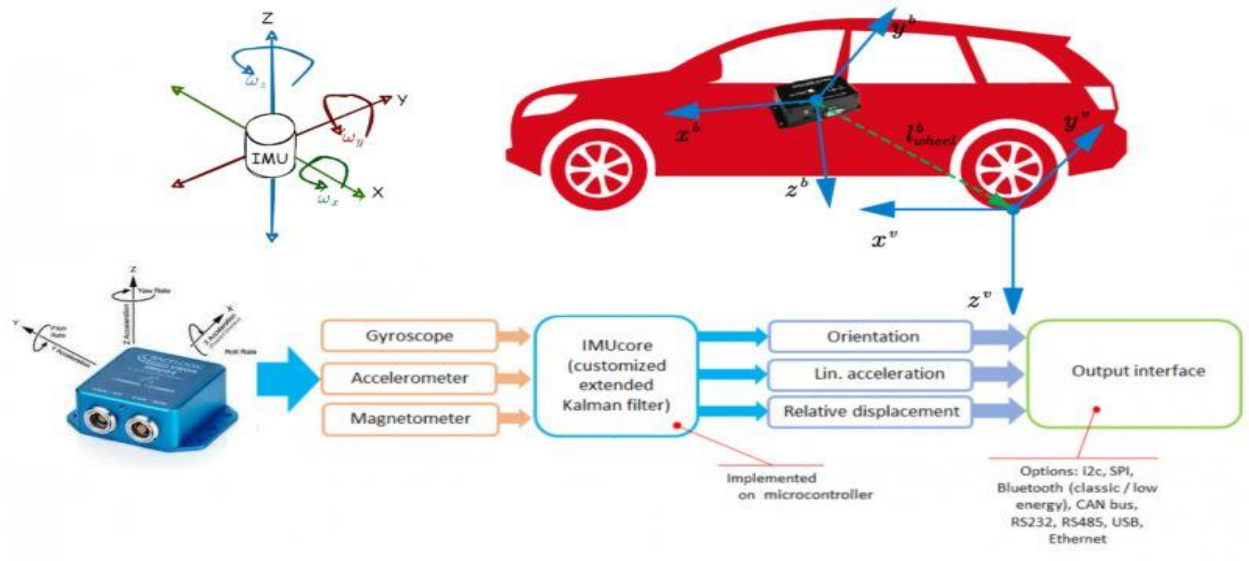


Figure 35 (2.5.2.2.2.b) MPU-6050 IMU

2.5.2.2.3. Arduino Nano [61]

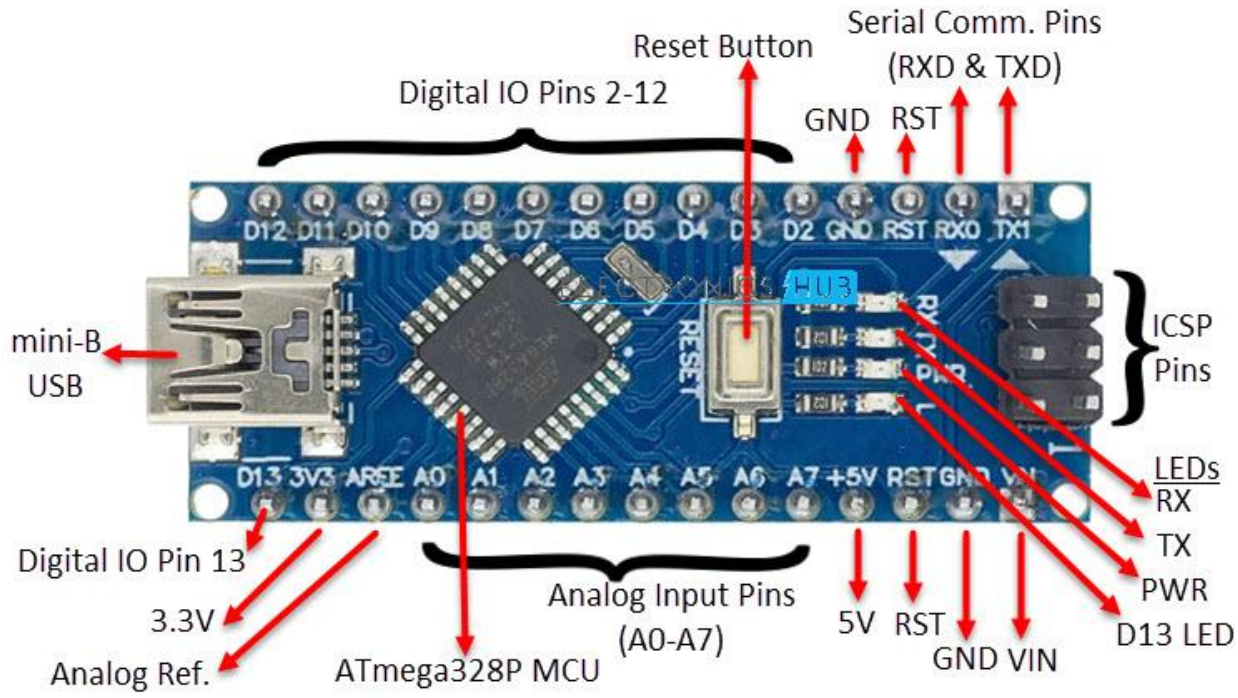


Figure 36 (2.5.2.2.3) Arduino Nano

The Arduino Nano is a small, complete, and breadboard-friendly board based on the ATmega328 (Arduino Nano 3.x). It lacks only a DC power jack and works with a Mini-B USB cable instead of a standard one.

2.5.2.2.3.1. Specifications

- Microcontroller: ATmega328
- Architecture: AVR
- Flash Memory: 32 KB of which 2 KB are used by bootloader.
- SRAM: 2 KB, EEPROM: 1 KB
- Clock Speed: 16 MHz
- Analog I/O Pins: 8, Digital I/O Pins: 22, PWM Output: 6
- Operating Voltage: 5 V, Input Voltage: 7-12 V

- DC Current per I/O Pins 40 mA (I/O Pins)
- Power Consumption: 19 mA

2.5.2.2.3.2. Power and Memory

The Arduino Nano can be powered via the Mini-B USB connection, 6-20V unregulated external power supply (pin 30), or 5V regulated external power supply (pin 27). The power source is automatically selected to the highest voltage source. The ATmega328 has 32 KB, (also with 2 KB used for the bootloader. The ATmega328 has 2 KB of SRAM and 1 KB of EEPROM.

2.5.2.2.3.3. Input and Output

Each of the 14 digital pins on the Nano can be used as an input or output, using pinMode(), digitalWrite(), and digitalRead() functions. They operate at 5 volts. Each pin can provide or receive a maximum of 40 mA and has an internal pull-up resistor (disconnected by default) of 20-50 kOhms. In addition, some pins have specialized functions:

- Serial: 0 (RX) and 1 (TX). Used to receive (RX) and transmit (TX) TTL serial data.
- External Interrupts: 2 and 3. These pins can be configured to trigger an interrupt on a low value, a rising or falling edge, or a change in value.
- PWM: 3, 5, 6, 9, 10, and 11. Provide 8-bit PWM output.
- SPI: 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK). These pins support SPI communication.
- LED: 13. There is a built-in LED connected to digital pin 13. When the pin is HIGH value, the LED is on, when the pin is LOW, it's off.

- I2C: 4 (SDA) and 5 (SCL). Support I2C (TWI) communication
- Reset. Bring this line LOW to reset the microcontroller.

The Nano has 8 analog inputs, each of which provides 10 bits of resolution (i.e., 1024 different values). By default, they measure from ground to 5 volts, though is it possible to change the upper end of their range using the analog Reference() function.

2.5.2.2.3.4. communication

The Arduino Nano has several facilities for communicating with a computer, another Arduino, or other microcontrollers. The ATmega328 provides UART TTL (5V) serial communication, which is available on digital pins 0 (RX) and 1 (TX). An FTDI FT232RL on the board channels this serial communication over USB and the FTDI drivers (included with the Arduino software) provide a virtual com port to software on the computer.

The RX and TX LEDs on the board will flash when data is being transmitted via the FTDI chip and USB connection to the computer (but not for serial communication on pins 0 and 1). The ATmega328 also supports I2C (TWI) and SPI communication.

2.2.5.2.4. Lidar [64]

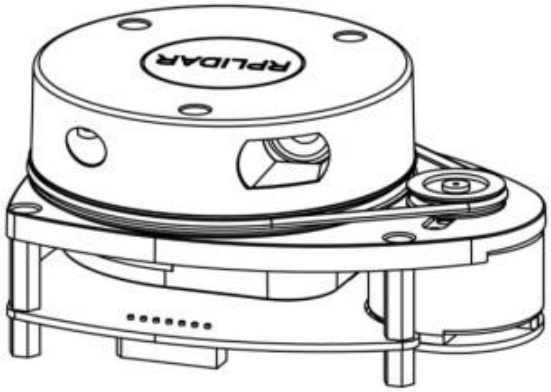


Figure 37 (2.2.5.2.4) RPLIDAR A1

RPLIDAR A1 is a low cost 360-degree 2D laser scanner (LIDAR) solution. The system can perform 360-degree scan within 12-meter range (6-meter range of A1M8-R4 and the belowing models). The produced 2D point cloud data can be used in mapping, localization, and object/environment modeling.
RPLIDAR A1's scanning frequency reached 5.5 Hz when sampling 360 points each round. And it can be configured up to 10 Hz maximum. RPLIDAR A1 is basically a laser triangulation measurement system.

2.5.2.2.4.1. System connection & Mechanism

RPLIDAR A1 contains a range scanner system and a motor system. After power on each sub-system, RPLIDAR A1starts rotating and scanning clockwise. Users can get range scan data through the communication interface (Serial port/USB). RPLIDAR A1 comes with a speed detection and adaptive system. The system will adjust frequency of laser scanner automatically according to motor speed. And the host system can get RPLIDAR A1's real speed through communication interface.

Figure 38 (2.5.2.2.4.1.a) RPLIDAR A1 System Composition

RPLIDAR is based on laser triangulation ranging principle and uses high-speed vision acquisition and processing hardware developed by SLAMTEC. The system measures distance data in more than 2000 times per second and with high resolution distance output (<1% of the distance). RPLIDAR emits modulated infrared laser signal, and the laser signal is then reflected by the object to be detected.

The returning signal is sampled by vision acquisition system in RPLIDAR A1 and the DSP embedded in RPLIDAR A1 starts processing the sample data and output distance value and angle value between object and RPLIDAR A1 through communication interface. The high-speed ranging scanner system is mounted on a spinning rotator with a built-in angular encoding system.

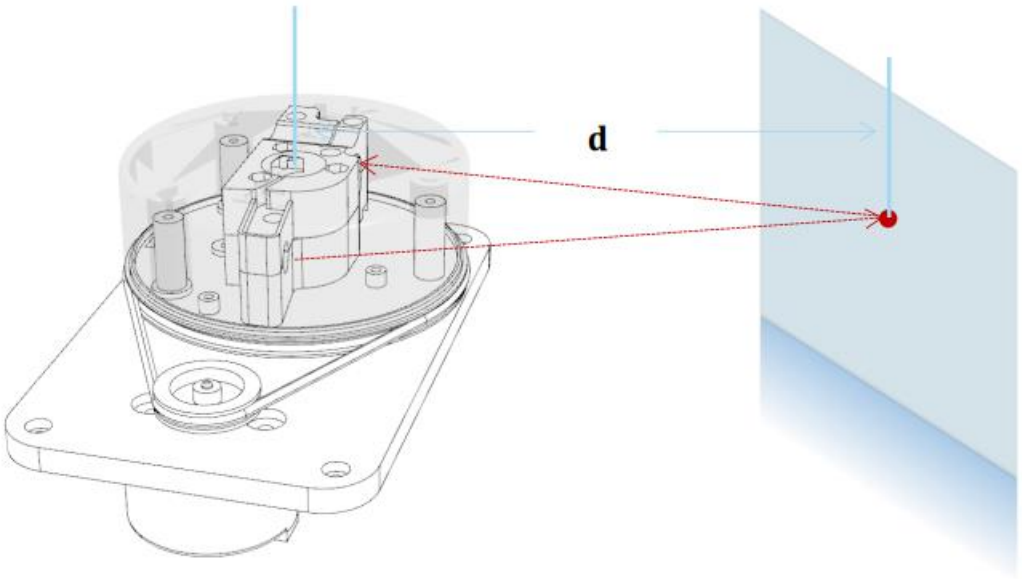


Figure 39 (2.5.2.2.4.1.b) RPLIDAR A1 Working Schematic

2.5.2.2.4.2. Communication interface

RPLIDAR A1 uses 3.3V-TTL serial port (UART) as the communication interface. Other communication interfaces such as USB can be customized according to customer's requirement.

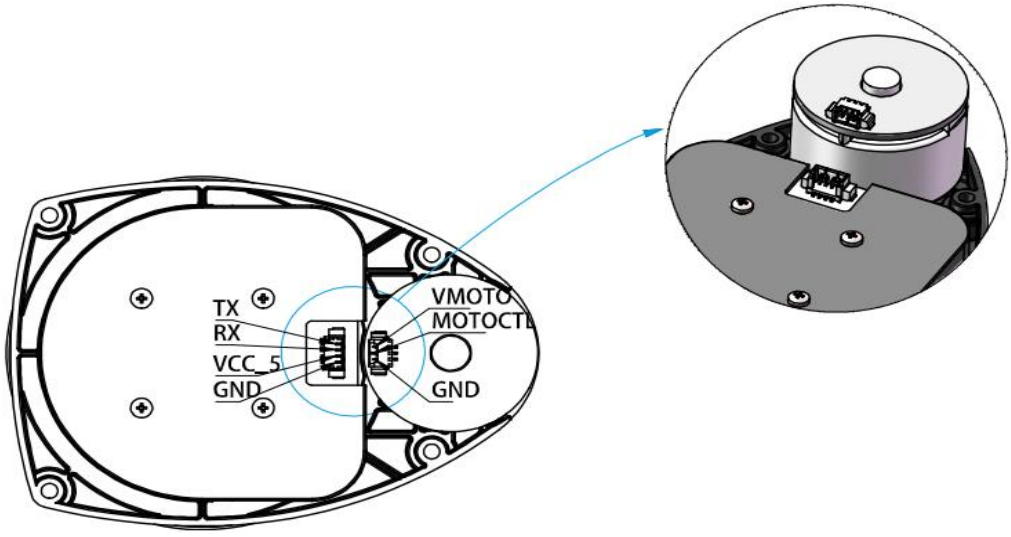


Figure 40 (2.5.2.2.4.2) RPLIDAR A1 Power Interface

2.5.2.2.5. Raspberry PI Camera

High-definition camera module compatible with all Raspberry Pi models. Provides high sensitivity, low crosstalk, and low noise image capture in an ultra-small and lightweight design. The camera module connects to the Raspberry Pi board via the CSI connector designed specifically for interfacing cameras. The CSI bus is capable of extremely high data rates, and it exclusively carries pixel data to the processor.

The camera comes with an image sensor (Sony IMX 219 PQ CMOS image sensor in a fixed-focus module). With a resolution of 8MP and Still picture resolution (3280 x 2464). Also, it has Max image transfer rate:1080p: 30fps (encode and decode), 720p: 60fps.

Image control functions: automatic exposure control, white balance, band filter, 50/60 Hz luminance detection and automatic black level calibration.

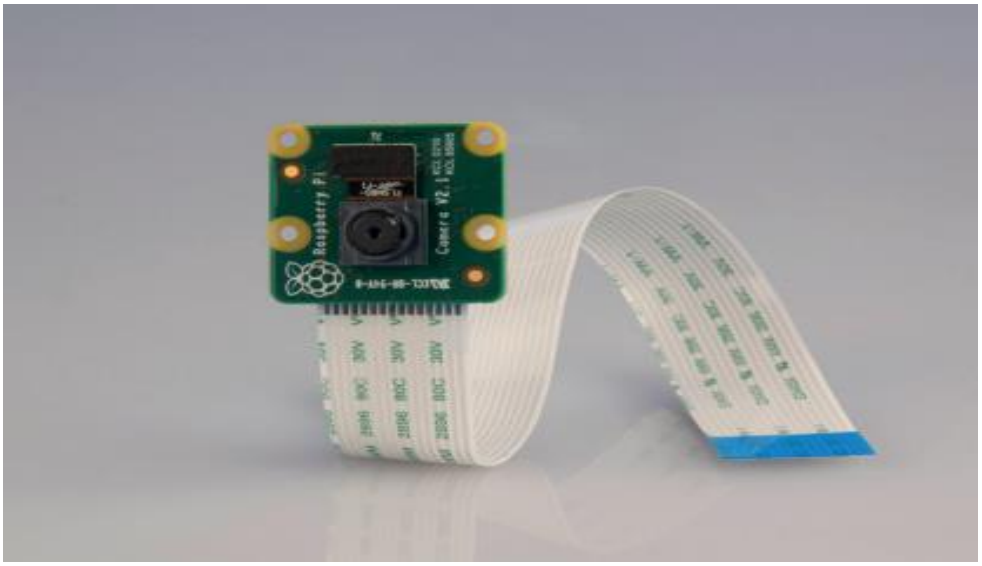


Figure 41 (2.5.2.2.5) Raspberry PI V2 Camera

2.5.2.2.6. DC – DC Step Down Converter

Figure 42 (2.5.2.2.6) DC – DC Step Down Converter

This DC-DC Step-down Adjustable Constant Voltage Module can be used to get adjustable output voltage ranges from 1.5V to 35V. The module provides a wide range of current output. With a heat sink mounted it can easily manage to run high power applications.

Specifications:

1. Input Supply voltage: 7 ~ 32V
2. Output Current up to 8A.
3. Output voltage: Continuously adjustable (0.8 ~ 28V)
4. Conversion efficiency: 95%
5. Operating frequency: 300KHz
6. No-load current: Typical 20mA
7. Output Power: max. 300W

2.5.2.2.7. Motor Driver

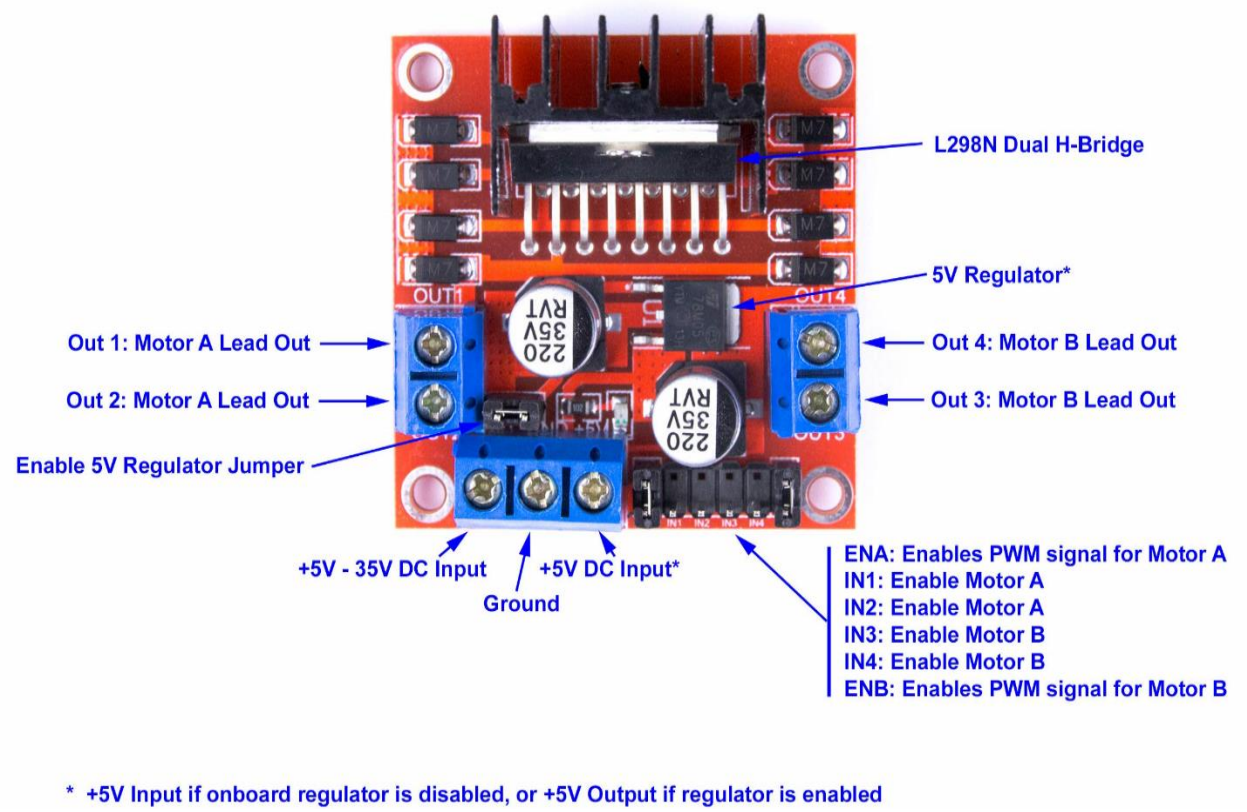


Figure 43 (2.5.2.2.7) L298N Motor Driver

This L298N Motor Driver Module is a high-power motor driver module for driving DC Motors. This module consists of an L298 motor driver IC and a 78M05 5V regulator. L298N Module can control up to 4 DC motors, or 2 DC motors with directional and speed control.

When the power supply is less than or equal to 12V, then the internal circuitry will be powered by the voltage regulator and the 5V pin can be used as an output pin to power the microcontroller. ENA & ENB pins are speed control pins for Motor A and Motor B which enable PWM signals for motor A and motor B, while IN1& IN2 and IN3 & IN4 are direction control pins for Motor A and Motor B.

2.2.5.2.8. Motor

25GA370 DC Gear Motor with Encoder 200 RPM 12V DC.

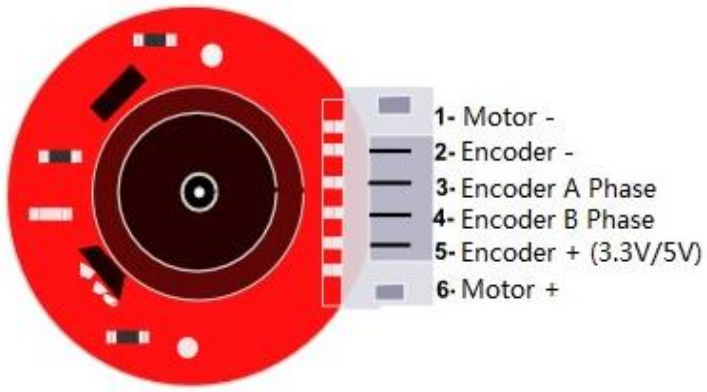


Figure 44 (2.2.5.2.8.a) Motor

Figure 45 (2.2.5.2.8.b) Motor

Pinout Specifications:

- Voltage: DC 12V
- Power: 3W
- Speed: 200RPM
- Stall Torque: 4.5 Kg/cm
- Rated Torque 1.43 Kg/cm.
- Stall current 1500 mA.
- Rated current approx. 330 mA.

Features

- All the gear of metal mechanism, Wear resistance, smooth transmission, compact structure.
- D-shaped double axis, excellent thermal conductivity.

Figure 46 ((2.2.5.2.8.c) Motor dimensions

2.5.2.2.9. Battery

Lithium-ion is the most popular rechargeable battery chemistry used today. Lithium-ion batteries consist of single or multiple lithium-ion cells, along with a protective circuit board. In our project we used a Lithium – Ion Battery with 12 Volts and 10000 mAh.

Lithium-ion characteristics:

1. Reliability and availability due to the different manufacturing processes involved.
2. Longer life and lower maintenance requirements lithium-ion batteries have a much lower failure rate compared to lead-acid batteries.
3. Lithium-ion batteries have a longer cycle life typically supplying 500-7000 discharge cycles.
4. Lithium-ion batteries have a lower self-discharge rate allowing them to be stored longer without draining.

5. Lithium-ion batteries have a higher energy density and power density, which means they can deliver the same energy as lead-acid batteries in a smaller size and lighter weight.

Lithium-ion components:

- Electrodes: The positively and negatively charged ends of a cell. Attached to the current collectors.
- Anode: The negative electrode
- Cathode: The positive electrode
- Electrolyte: A liquid or gel that conducts electricity
- Current collectors: Conductive foils at each electrode of the battery that are connected to the terminals of the cell.
- Separator: A porous polymeric film that separates the electrodes while enabling the exchange of lithium ions from one side to the other.

Concept of working:

In a lithium-ion battery, lithium ions (Li+) move between the cathode and anode internally. Electrons move in the opposite direction in the external circuit. While the battery is discharged, the anode releases lithium ions to the cathode, generating a flow of electrons that helps to power the relevant device. When the battery is charging, the opposite occurs: lithium ions are released by the cathode and received by the anode.

2.5.2.3. Battery comparison

2.5.2.3.1. Comparison table between most common battery types

Specifications	Lead-Acid	Ni-Cd	Ni-MH	Lithium -Ion		
				Cobalt	Manganese	Phosphate
Specific energy (Wh/Kg)	30 – 50	45 – 80	60-120	150 - 250	100 – 150	90 – 120
Internal resistance	Very low	Very low	Low	Moderate	Low	Very low
Charge time	8 – 16 h	1 – 2 h	2 – 4 h	2 – 4 h	1 – 2 h	1 – 2 h
Cell voltage	2 V	1.2 V	1.2 V	3.6 V	3.7 V	3.2 – 3.3 V
Charge temp	-20 – 50 ºC	0 – 45 ºC		0 – 45 ºC		
Maintenance	3 – 6 months	90 days		Maintenance free		
Safety requirements	stable	Thermally stable		Protection circuit mandatory		
Toxicity	Very high	Very high	Low	Low		
Cost	low	Moderate		High		

Table 10 (1.5.2.3.1) Battery comparison

2.5.2.4. Electric Circuit

This circuit consists of the above components where the battery is connected directly to the motor drivers which they connected with the two DC motors, then a parallel connection with motors a DC – DC step down converter to reduce the volt from 12 V to 5 V which is given to Raspberry PI 4B (5V and 3A) and then Arduino Nano powered by Raspberry PI. Finally, we will connect the Raspberry Pi with Lidar and Raspberry PI camera via CSI connector. Also, we will connect the two motors drivers, encoders, IMU and the ultrasonic sensor with Arduino Nano. Raspberry PI and Arduino Nano are connected by serial communication using ROS serial. The reason we made this connection is because we want Raspberry to concentrate more on camera and Lidar readings and to be capable of processing SLAM algorithms to make path planning.

First, the following figure represents the circuit used using fritzing, but we will illustrate the circuit into two figures due to when connect many components the figure will not be clear enough as we should zoom out to make all components obvious.

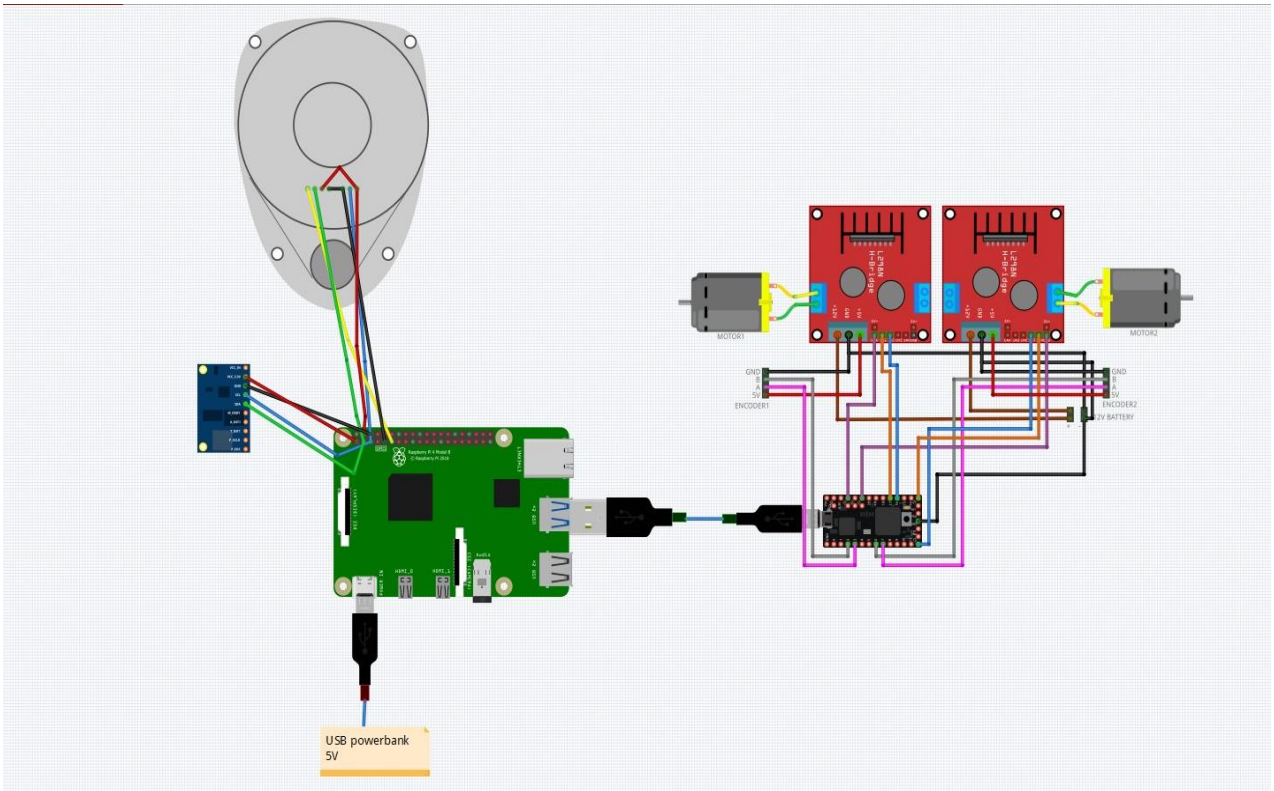


Figure 48 (2.5.2.4.b) Electric circuit 1

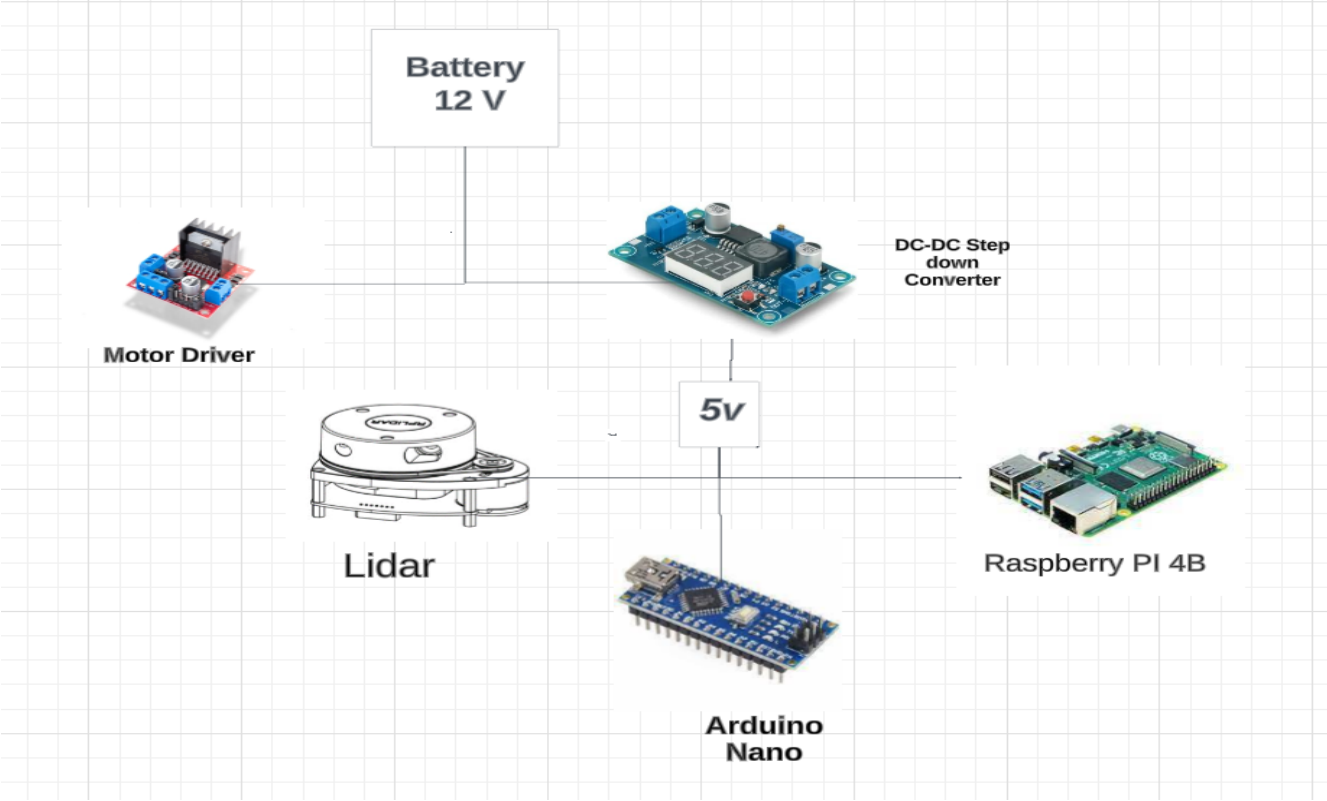


Figure 47 (2.5.2.4.a) Power circuit

2.5.3. Hardware Interfacing

The Drivers will be written to every hardware separately using Arduino IDE Interfacing with ROS via Serial Interface.

The code works by interfacing the motors with the LM298N driver, to control the direction and speed of the motors. The motors are controlled via a differential drive PID controller to improve accuracy and response. The encoders are controlled via the microcontroller's hardware interrupts pins, in which they are used to detect the motor's speed and direction with as they use the quadrature encoder concept, of having two channels that is when one signal is leading the other, the direction whether it is clockwise or anti-clockwise is detected. The code has a ROS-Arduino Bridge that is used to communicate the raspberry pi with the microcontroller via serial interface. The code can be found at the appendices (39).

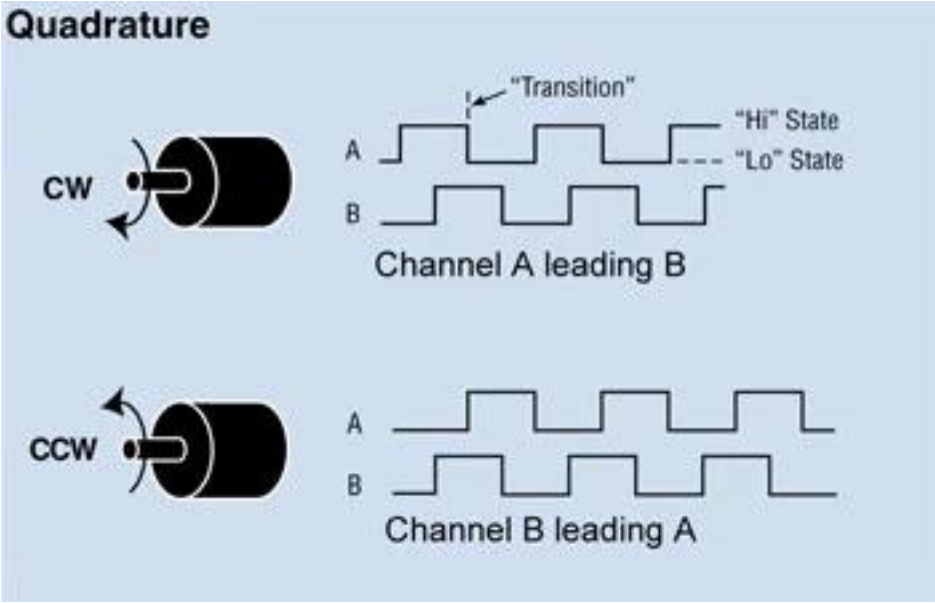


Figure 49 (2.5.3.1) Quadrature encoder concept

We control the motors using ROS serial package which communicates with the motors for the target speed and direction.

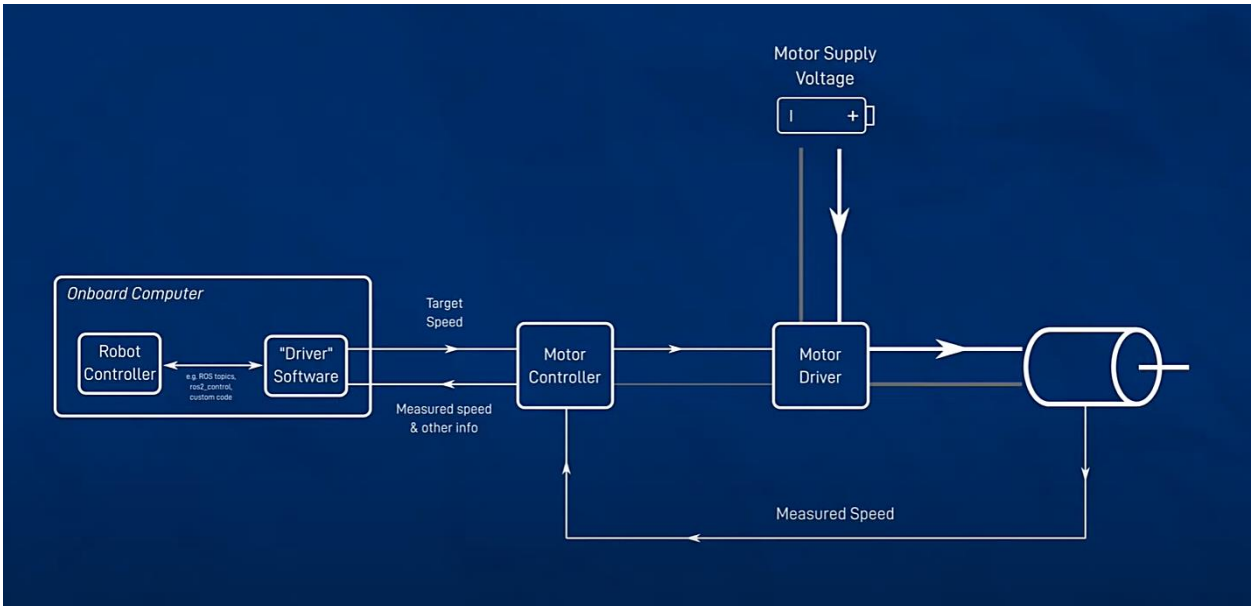


Figure 50 (2.5.3.2) Motors control schematic

2.5.4. Software

We will be using Robot Operating System (ROS2) foxy distribution as our firmware that controls all parts of the robot and to execute the mapping and path planning process.

2.5.4.1. ROS overview:

ROS stands for Robot Operating System, in summary, ROS is a framework by which you could build scalable robotic applications for robots ranging from floor vacuums to ones that go to space. The purpose of ROS is to decompose complex software into smaller and more manageable pieces. ROS simplifies the development process as it provides a unified interface for communication between different processes (or Nodes) on which different robot software components can be run. In addition, ROS has a large ecosystem of sensor, control, and algorithmic packages made available by community contributions, enabling a small team to build complex robotics applications fast.

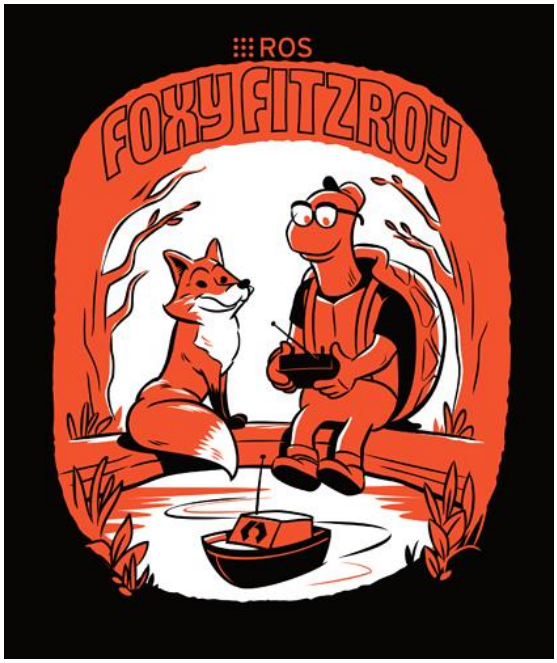


Figure 51 (2.5.4.1.a) ROS Foxy

ROS nodes represent an independent process in the ROS stack, and they can communicate with each other using 3 primary modes:

2.5.4.1.1. Ros Topics (publisher/subscriber):

Enables nodes to broadcast (or publish) messages to any other nodes that choose to listen (or subscribe) to this topic.

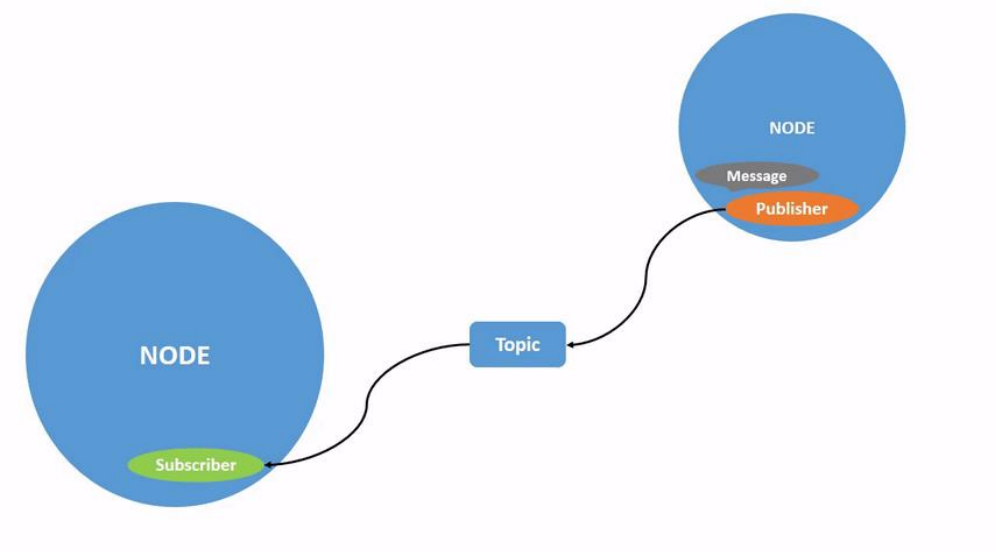


Figure 52 (2.5.4.1.1) ROS Topics

2.5.4.1.2. Ros Services (request/response):

ROS (Robot Operating System) services are a communication mechanism provided by ROS to enable interprocess communication between different nodes in a robotic system. While ROS topics facilitate the exchange of messages in a publish-subscribe manner, ROS services allow for synchronous request-response interactions.

ROS services follow a client-server model. A node can act as a service client, which sends a request to another node that acts as a service server. The server processes the request and sends a response back to the client. This request-response interaction occurs over a bidirectional communication channel.

Here are some key characteristics and features of ROS services:

1. Synchronous communication: ROS services provide synchronous communication, meaning that the client node waits for a response from the server before proceeding with its execution.

2. Defined service type: Services in ROS have a specific service type defined in a .srv file. This file specifies the structure of the request message and the structure of the response message.

3. Service calls: To make a service call, the client node constructs a request message according to the service type and sends it to the server. The server processes the request and generates a response message, which is sent back to the client.

4. Service server: A node that provides a service is referred to as a service server. It registers the service with the ROS Master and waits for incoming requests from clients.

5. Service client: A node that wants to use a service is referred to as a service client. It searches for the server offering the desired service and sends requests as needed.

6. Unique service names: Services are identified by unique names in the ROS network. The names typically follow the format /node_name/service_name.

7. Request-response pattern: ROS services follow a strict request-response pattern. The client initiates a request, and the server responds with a corresponding response. Multiple requests can be made to the same service over time.

8. Blocking behavior: When a client sends a request to a server, it blocks execution until it receives a response or until a timeout occurs.

ROS services are useful in scenarios where you require a direct response from another node and need to wait for that response before proceeding further in your system's execution flow. Examples include querying a robot's state, requesting a sensor reading, or asking a robot to perform a specific action.

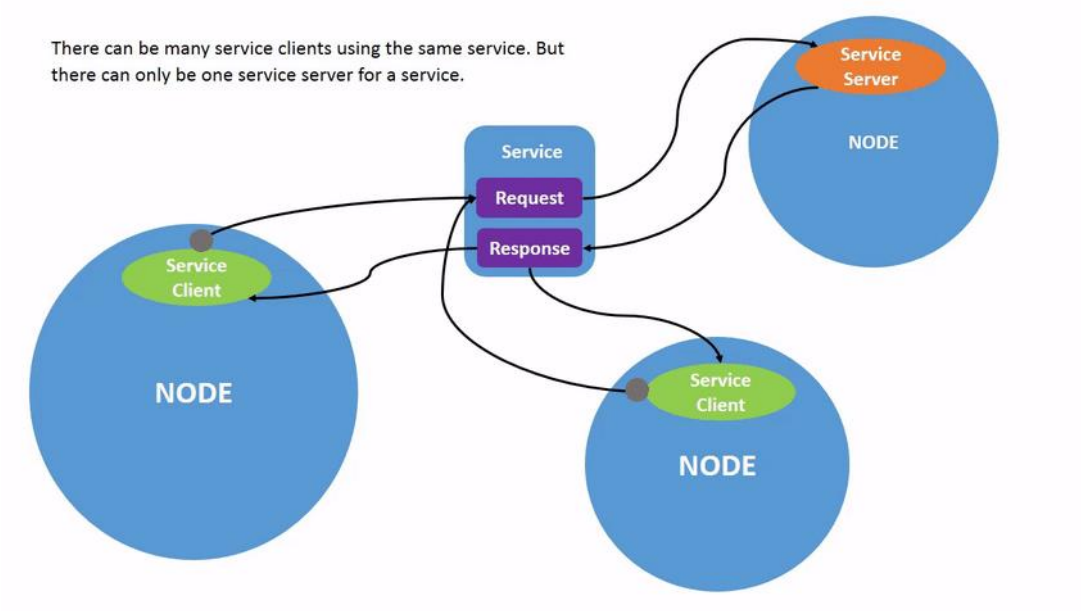


Figure 53 (2.4.5.1.2) ROS Services

2.5.4.1.3. Ros actions (action/feedback/result):

ROS (Robot Operating System) actions are another communication mechanism provided by ROS to enable asynchronous, goal-oriented behavior between different nodes in a robotic system. While ROS topics facilitate the exchange of messages in a publish-subscribe manner and ROS services enable synchronous request-response interactions, ROS actions allow for more complex and long-running behaviors with feedback and goal tracking.

ROS actions are designed to handle tasks that may take some time to complete, such as moving a robot to a specific location, performing a sequence of actions, or executing a long-running computation. They follow a client-server model similar to services but with additional capabilities for tracking progress and providing feedback.

Here are some key characteristics and features of ROS actions:

1. Asynchronous communication: ROS actions provide asynchronous communication, allowing the client node to continue its execution while the server performs the requested task. This is especially useful when dealing with long-running operations.

2. Defined action type: Actions in ROS have a specific action type defined in an action file. This file specifies the structure of the goal message, feedback message, and result message.

3. Action client: A node that wants to initiate an action is referred to as an action client. It sends a goal message to the action server and can optionally receive feedback and track the progress of the action.

4. Action server: A node that provides an action is referred to as an action server. It registers the action with the ROS Master and handles incoming goal requests from clients. The server can provide feedback messages to the client during the execution of the action.

5. Goal message: The client constructs a goal message according to the action type and sends it to the server. The goal message specifies the desired outcome of the action.

6. Feedback message: During the execution of the action, the server can send feedback messages to the client, providing information about the progress or current state of the action.

7. Result message: Once the action is completed, the server sends a result message back to the client, indicating the final outcome of the action.

8. Preemption: Actions can be preempted if necessary. This means that a client can send a cancel request to the server, and the server can interrupt the ongoing action and provide a response indicating the cancellation.

ROS actions provide a more flexible and robust way to handle complex behaviors in a robotic system. They allow for the execution of long-running tasks while providing feedback and progress tracking to the client. Actions are commonly used for tasks such as navigation, manipulation, planning, and coordination in ROS-based robotic applications.

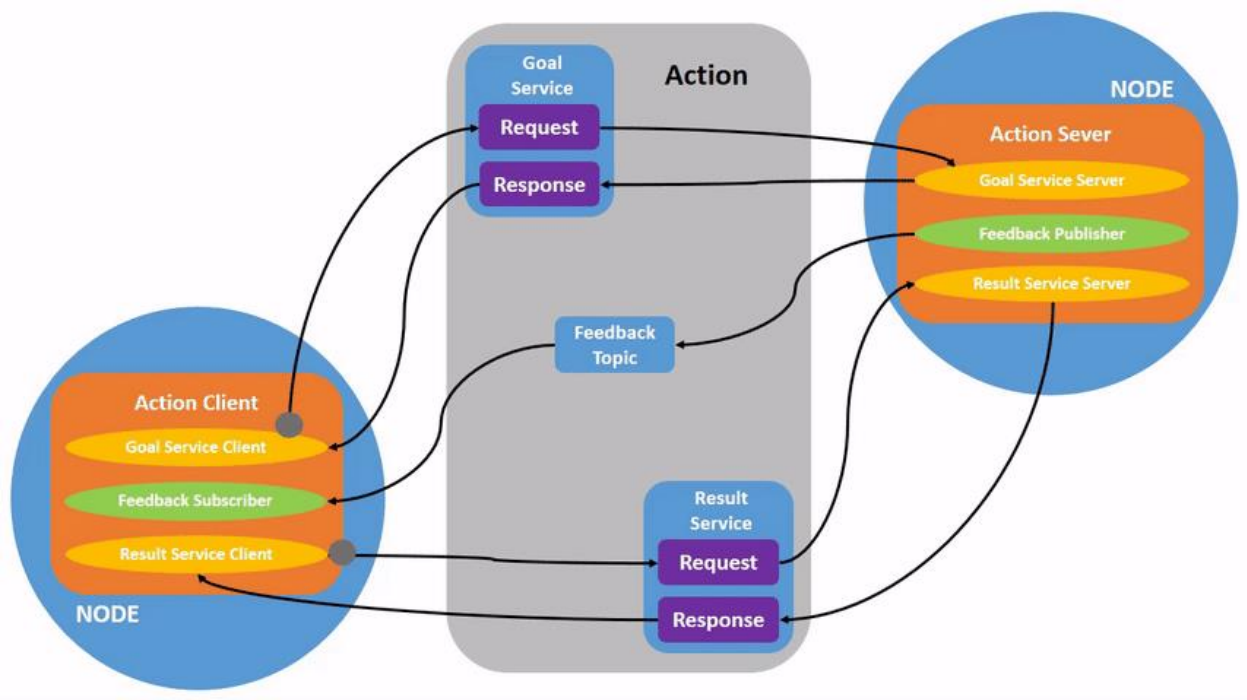


Figure 54 (2.4.5.1.3) ROS Actions

2.5.4.2. Why ROS?

There are several reasons why ROS is widely used in the development of Autonomous Robots.

1. Modularity: ROS is designed to be modular, meaning that it is easy to add or remove features and functionality as needed. This allows developers to build systems that are customized to meet their specific needs and goals.
2. Robustness: ROS is designed to be robust and reliable, which is crucial for autonomous systems that must operate in complex and dynamic environments.
3. Interoperability: ROS is designed to be interoperable with a wide range of sensors, devices, and software platforms. This makes it easier to integrate different components and systems into a cohesive whole.
4. Community support: ROS has a large and active community of users and developers who contribute to its development and provide support to others. This makes it easier for developers to get help and find resources when they are working with ROS.

Overall, ROS is an important tool for building autonomous systems because it provides a flexible, reliable, and interoperable platform that can be customized to meet the needs of a wide range of applications.

2.5.4.3. ROS 1 VS ROS 2

ROS1 was initially built and released by Willow Garage in 2007 to accelerate robotics research. Since then, ROS1 has gained a lot of popularity in the robotics community, and it became the most common platform for performing research and experimentation. However, ROS1 was not built with commercial use in mind, so things like security, network topology, and system uptime were not

prioritized. So, with ROS seeing adoption in the commercial space right now. Hence, there was a need to rebuild ROS from the ground up with commercial use in mind, like ROS 2.

ROS 2 was built from the ground up with the following design requirements in order to make it ready for commercial use and adoption:

1. security: It needs to be safe with proper encryption where needed
2. Embedded Systems: ROS 2 needs to be able to run on embedded systems.
3. Diverse networks: Need to be able to run and communicate across vast networks since robots from LAN to multi-satellite hops to accommodate the variety of environments.
4.Real-time computing: Need to be able to perform computation in real time reliably since runtime efficiency is crucial in robotics.
5.Product Readiness: Need to conform to relevant industrial standards such that it ready for the market.

2.5.4.3.1. Using DDS (Data Distribution Service protocol):

ROS2 designers decided to switch to using DDS as the network protocol for all communication that happens internally in ROS2. DDS stands for the Data Distribution Service protocol and is widely used in critical infrastructures such as battleships, space systems, and others. It provides the security guarantees as well as the reliability needed to maintain good communication in areas with weak or lossy connections. This is in contrast with the TCP/UDP custom protocol that was developed in ROS1 which missed a lot of the security and reliability guarantees provided by DDS.

Unfortunately, ROS 1 does not support DDS which provides security guarantees. So, this is an enormous concern for critical commercial applications

and there were lots of attempts in the community to address this concern for ROS1, but the solutions developed didn't address the fundamental limitations in the design, as ROS1 was primarily built as a research tool.

Finally, switching to DDS in ROS2 made all the security concerns go away. DDS is an established and tested protocol that provides security guarantees that were otherwise not present in ROS1. So, in that regard, commercial companies using ROS2 can have peace of mind if their robots have to communicate over insecure networks.

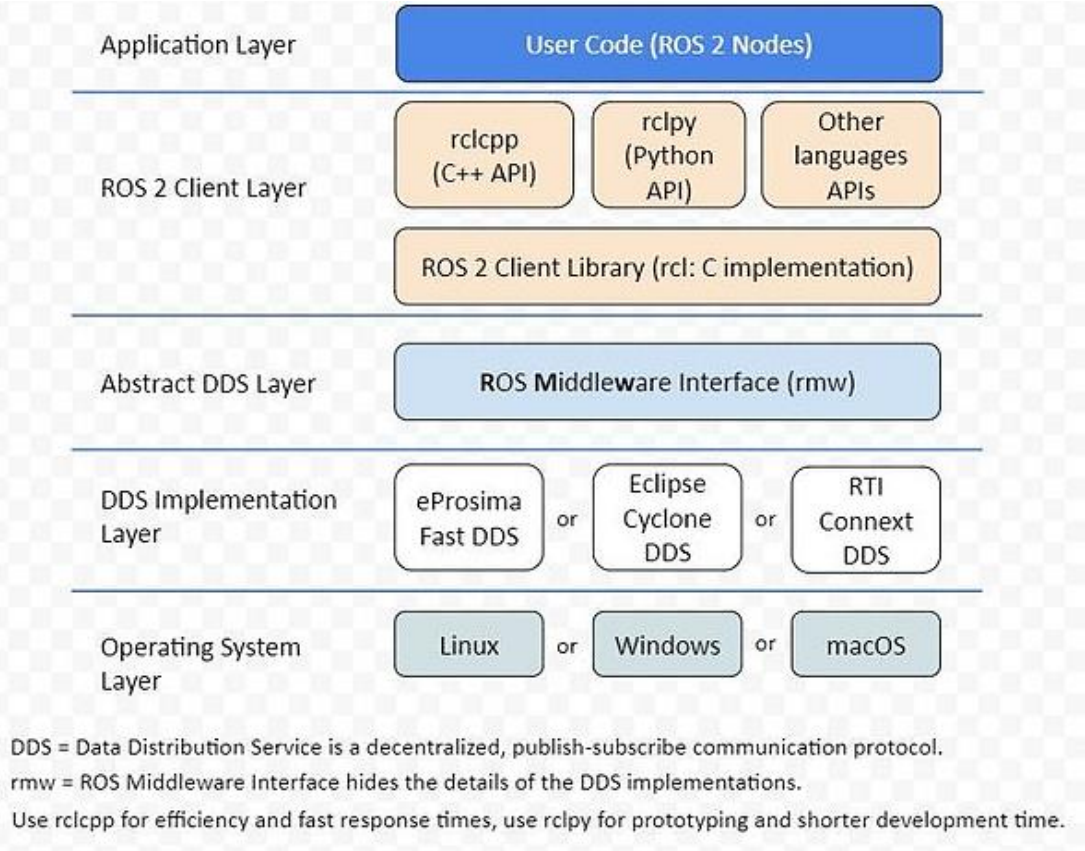


Figure 55 (2.5.4.3.1) ROS2 Architecture Overview

2.5.4.3.2. Sharing client library:

In ROS2, all client libraries now share a common implementation written in C called (rcl), which sits between the client libraries and the DDS interfaces required for communication. This makes each of the client libraries' implementation lighter, as it just wraps the common rcl implementation, which would generally provide more consistent performance across different languages as under the hood they all use rcl. This also eases the ability of developers to create new client libraries for new languages.

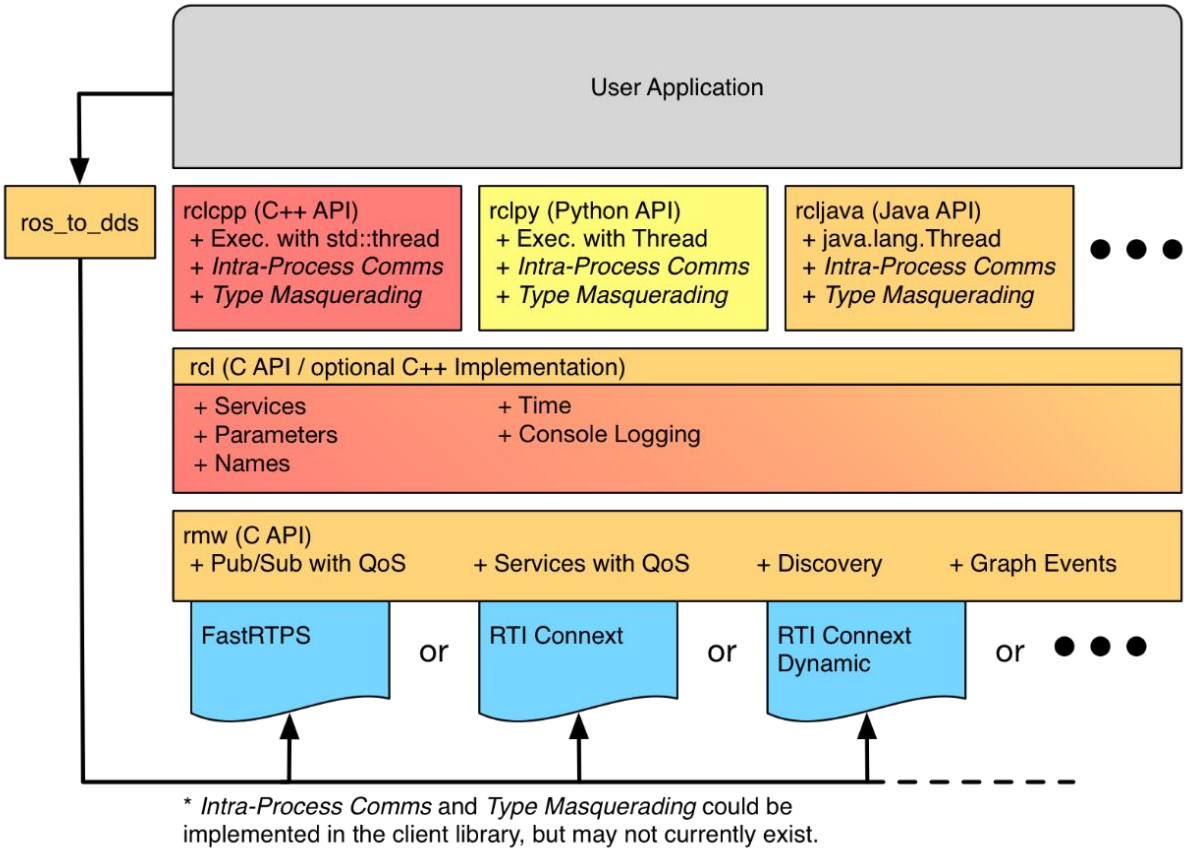


Figure 56 (2.5.4.3.2) ROS 2 sharing client libraries.

2.5.4.4 ROS2 Communication

TCP/IP (Transmission Control Protocol/Internet Protocol) is a suite of communication protocols that form the foundation of the modern internet. In the context of ROS 2, TCP/IP is used as one of the transport options for communication between nodes in a distributed robotic system. Here's how TCP/IP is used in ROS 2:

1) Transport Layer: TCP/IP provides a reliable transport layer protocol (TCP) that ensures data integrity and ordered delivery. It guarantees that messages sent from one node to another are received without errors and in the correct order. This is important for critical applications where data consistency and reliability are essential.

2) ROS 2 Communication: ROS 2 uses the concept of a middleware to enable communication between nodes. The ROS 2 middleware provides a set of protocols and transports, including TCP/IP, to facilitate the exchange of messages. TCP/IP is used as a transport option alongside other protocol like UDP (User Datagram Protocol).

3) ROS 2 Nodes: In ROS 2, nodes can communicate with each other using publishers and subscribers. Publishers send messages over a chosen transport, such as TCP/IP, and subscribers receive those messages. TCP/IP provides a reliable and ordered message delivery mechanism between nodes.

4) Distributed Systems: ROS 2 is designed to support distributed robotic systems where nodes can run on different machines across a network.

TCP/IP allows nodes running on separate machines to communicate with each other seamlessly. This enables the distribution of computational load and facilitates the coordination of multiple robots or components in a networked environment.

5) Quality of Service (QoS): ROS 2 allows users to configure Quality of Service parameters to optimize communication based on their application requirements. TCP/IP provides QoS features such as reliability, durability, and flow control, which can be utilized to ensure message delivery, persistence, and manage bandwidth utilization.

6) Firewalls and Security: TCP/IP operates over standard network infrastructure, making it compatible with firewalls and network security measures. It enables ROS 2 communication to traverse network boundaries, allowing nodes to interact across different network segments while maintaining security protocols.

It's important to note that while TCP/IP offers reliability and guarantees data integrity, it introduces additional overhead compared to UDP. In ROS 2, both TCP/IP and UDP transport options are available, allowing users to choose the appropriate transport based on their application's specific requirements, network conditions, and trade-offs between reliability and latency.

2.5.4.4.1. TCP/IP Protocol [43]

2.5.4.4.1.1. What is TCP/IP Protocol

TCP/IP Model helps you to determine how a specific computer should be connected to the internet and how data should be transmitted between them. It helps you to create a virtual network when multiple computer networks are connected. The purpose of TCP/IP model is to allow communication over large distances.

TCP/IP stands for Transmission Control Protocol/ Internet Protocol. TCP/IP Stack is specifically designed as a model to offer highly reliable and end-to-end byte stream over an unreliable internetwork.

2.5.4.4.1.2. TCP Characteristics

- Support for a flexible TCP/IP architecture
- Adding more system to a network is easy.
- In TCP IP protocols suite, the network remains intact until the source, and destination machines were functioning properly.
- TCP is a connection-oriented protocol.
- TCP offers reliability and ensures that data which arrives out of sequence should put back into order.
- TCP allows you to implement flow control, so sender never overpowers a receiver with data.

2.4.4.1.1.3. Four Layers of TCP/IP model

In this TCP/IP tutorial, we will explain different layers and their functionalities in TCP/IP model:

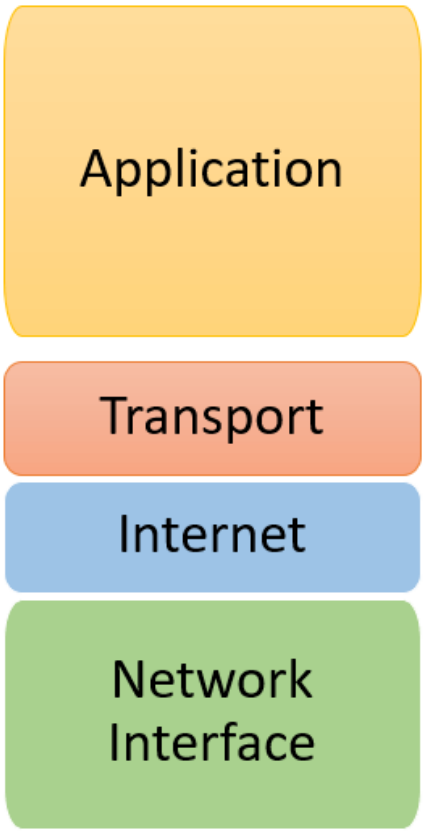


Figure 57 (2.4.4.1.1.3.a) TCP/IP Conceptual Layers

The functionality of the TCP IP model is divided into four layers, and each includes specific protocols.

TCP/IP is a layered server architecture system in which each layer is defined according to a specific function to perform. All these four TCP IP layers work collaboratively to transmit the data from one layer to another.

- Application Layer
- Transport Layer
- Internet Layer
- Network Interface

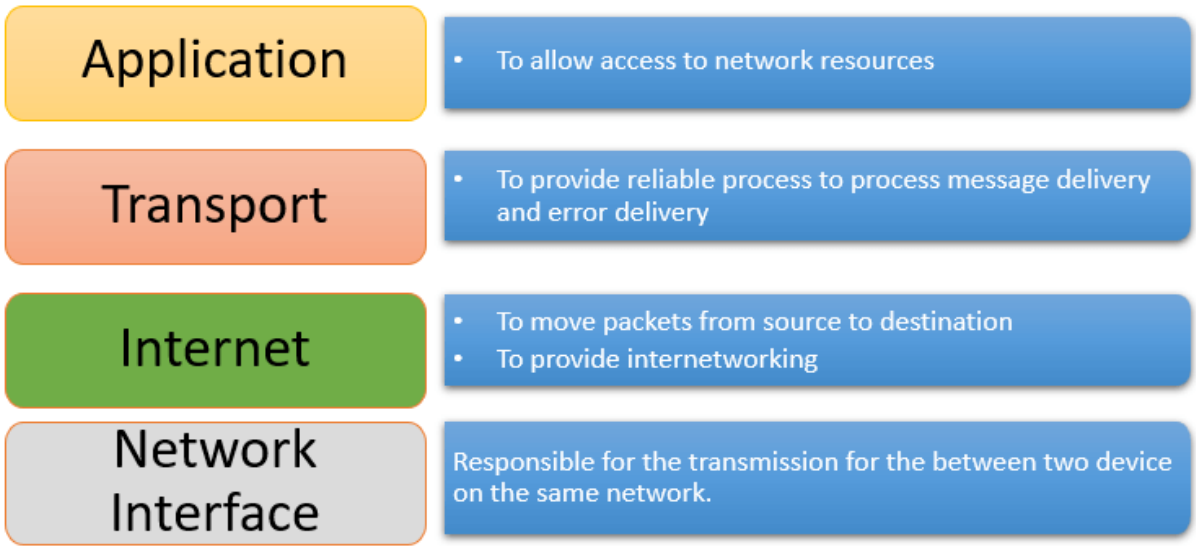


Figure 58 (2.4.4.1.1.3.b) Four Layers of TCP/IP Model

2.4.4.1.1.4. Application Layer

Application layer interacts with an application program, which is the highest level of OSI model. The application layer is the OSI layer, which is closest to the end-user. It means the OSI application layer allows users to interact with other software applications.

Application layer interacts with software applications to implement a communicating component. The interpretation of data by the application program is always outside the scope of the OSI model.

Example of the application layer is an application such as file transfer, email, remote login, etc.

2.4.4.1.1.4.1. The function of the Application Layers is:

- Application-layer helps you to identify communication partners, determining resource availability, and synchronizing communication.
- It allows users to log on to a remote host
- This layer provides various e-mail services

- This application offers distributed database sources and access for global information about various objects and services.

2.4.4.1.1.5. Transport Layer

Transport layer builds on the network layer in order to provide data transport from a process on a source system machine to a process on a destination system. It is hosted using single or multiple networks, and also maintains the quality-of-service functions.

It determines how much data should be sent where and at what rate. This layer builds on the message which are received from the application layer. It helps ensure that data units are delivered error-free and in sequence.

Transport layer helps you to control the reliability of a link through flow control, error control, and segmentation or de-segmentation.

The transport layer also offers an acknowledgment of the successful data transmission and sends the next data in case no errors occurred. TCP is the best-known example of the transport layer.

2.4.4.1.1.5.1. Important functions of Transport Layers:

- It divides the message received from the session layer into segments and numbers them to make a sequence.
- Transport layer makes sure that the message is delivered to the correct process on the destination machine.
- It also makes sure that the entire message arrives without any error else it should be retransmitted.

2.4.4.1.1.6. Internet Layer

An internet layer is a second layer of TCP/IP layers of the TCP/IP model. It is also known as a network layer. The main work of this layer is to send the packets from any network, and any computer still they reach the destination irrespective of the route they take.

The Internet layer offers the functional and procedural method for transferring variable length data sequences from one node to another with the help of various networks.

Message delivery at the network layer does not give any guaranteed to be reliable network layer protocol.

Layer-management protocols that belong to the network layer are:

1. Routing protocols
2. Multicast group management
3. Network-layer address assignment.

2.4.4.1.1.7. The Network Interface Layer

Network Interface Layer is this layer of the four-layer TCP/IP model. This layer is also called a network access layer. It helps you to define details of how data should be sent using the network.

It also includes how bits should optically be signaled by hardware devices which directly interface with a network medium, like coaxial, optical, coaxial, fiber, or twisted-pair cables.

A network layer is a combination of the data line and defined in the article of OSI reference model. This layer defines how the data should be sent physically through the network. This layer is responsible for the transmission of the data between two devices on the same network.

2.4.4.1.1.8. Most Common TCP/IP Protocols

TCP:

Transmission Control Protocol is an internet protocol suite which breaks up the message into TCP Segments and reassembling them at the receiving side.

IP:

An Internet Protocol address that is also known as an IP address is a numerical label. It is assigned to each device that is connected to a computer network which uses the IP for communication. Its routing function allows internetworking and essentially establishes the Internet. Combination of IP with a TCP allows developing a virtual connection between a destination and a source.

HTTP:

The Hypertext Transfer Protocol is a foundation of the World Wide Web. It is used for transferring webpages and other such resources from the HTTP server or web server to the web client or the HTTP client. Whenever you use a web browser like Google Chrome or Firefox, you are using a web client. It helps HTTP to transfer web pages that you request from the remote servers.

SMTP:

SMTP stands for Simple mail transfer protocol. This protocol supports the e-mail is known as a simple mail transfer protocol. This protocol helps you to send the data to another e-mail address.

SNMP:

SNMP stands for Simple Network Management Protocol. It is a framework which is used for managing the devices on the internet by using the TCP/IP protocol.

DNS:

DNS stands for Domain Name System. An IP address that is used to identify the connection of a host to the internet uniquely. However, users prefer to use names instead of addresses for that DNS.

TELNET:

TELNET stands for Terminal Network. It establishes the connection between the local and remote computer. It established connection in such a manner that you can simulate your local system at the remote system.

FTP:

FTP stands for File Transfer Protocol. It is a mostly used standard protocol for transmitting the files from one machine to another.

2.4.4.1.1.9. Advantages of the TCP/IP model

- It helps you to establish/set up a connection between different types of computers.
- It operates independently of the operating system.
- It supports many routing-protocols.
- It enables internetworking between the organizations.
- TCP/IP model has a highly scalable client-server architecture.
- It can be operated independently.
- Supports several routing protocols.

- It can be used to establish a connection between two computers.

2.4.4.1.1.10. Disadvantages of the TCP/IP model

- TCP/IP is a complicated model to set up and manage.
- The shallow/overhead of TCP/IP is higher-than IPX (Internetwork Packet Exchange).
- In this, model the transport layer does not guarantee delivery of packets.
- Replacing protocol in TCP/IP is not easy.
- It has no clear separation from its services, interfaces, and protocols.

2.5.4.5. Simulation Tools

2.5.4.5.1. Gazebo [3]

Gazebo is an open-source 3D robotics simulator. Gazebo simulated real-world physics in a high-fidelity simulation. It helps developers rapidly test algorithms and design robots in digital environments. Gazebo is currently the leader in robotic simulation, but other big companies like Unity and Nvidia are starting to catch up and create their robotic simulation software. Robotics simulation is an ever-growing space. Companies are investing more and more money to improve their workflow through robotic simulation. Robotic simulation saves a lot of time and money because it allows people to test how robots work without huge investments.

Gazebo helps to integrate a multitude of sensors and gives the tools to test these sensors and develop your robots to best use them. Even if having access to hardware, Gazebo is still a useful tool because it allows to test the robotic design before implementing it in the real world. This is why companies are investing so much money into robotic simulators and digital twins. They want to increase their manufacturing processes' workflow and speed without spending too much money

on hardware. Since Gazebo is open-source software, there are also many 3rd party plugins and solutions that help to solve specific problems which might come across or speed up the workflow.

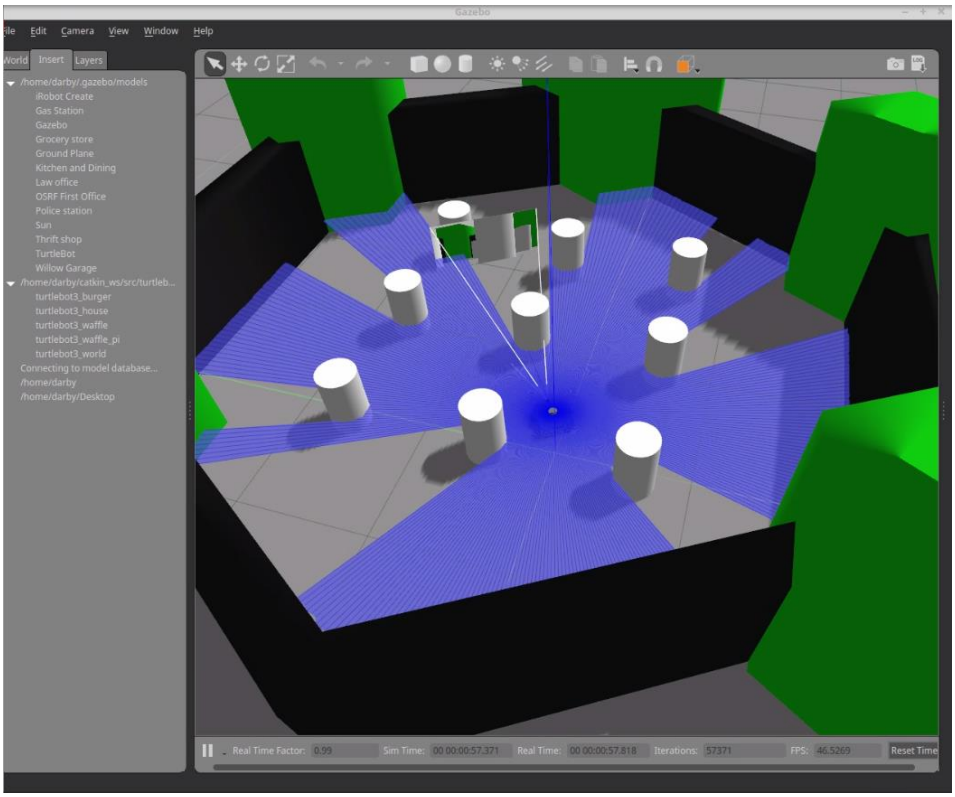


Figure 59 (2.5.4.5.1.a) Gazebo Simulation Example

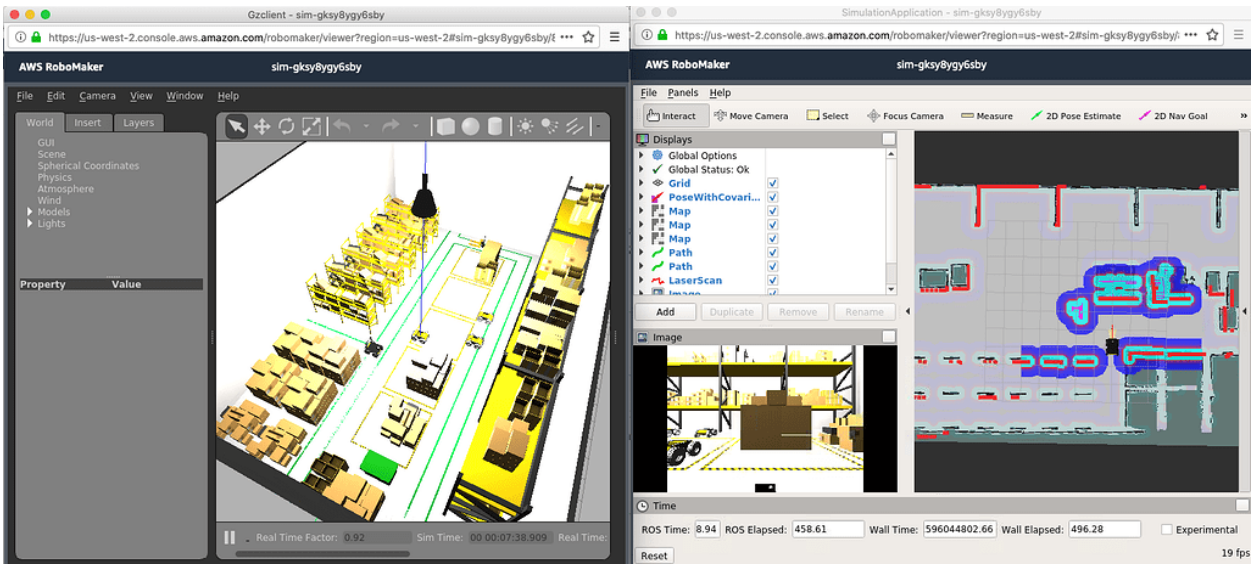


Figure 60 (2.5.4.5.1.b) Gazebo with RViz Simulator for Creating Maps

2.5.4.5.2. RViz [42]

RViz (ROS Visualization) is a powerful 3D visualization tool that is commonly used in ROS (Robot Operating System) for visualizing and debugging robotic systems. It provides a real-time graphical interface to display sensor data, robot models, and various types of visualization markers. Here's an overview of RViz and its uses in ROS:

1) Visualization: RViz allows users to visualize data from sensors, robots, and other sources in a 3D virtual environment. It supports the display of sensor data such as point clouds, laser scans, images, and depth maps. Users can visualize robot models, including their joints, links, and transforms. Additionally, RViz provides visualization markers to display other relevant information like paths, grids, and text labels.

2) Debugging and Analysis: RViz is a valuable tool for debugging and analyzing robotic systems. It enables users to inspect the state and behavior of robots by visualizing sensor readings, robot poses, and trajectories. By examining the visual representation of data, users can identify potential issues, validate sensor measurements, and debug complex robot behaviors.

3) Interactive Control: RViz provides interactive control features that allow users to manipulate the visualization environment. Users can select objects, move and rotate the camera view, measure distances, and interact with interactive markers associated with the robot model. These capabilities facilitate intuitive exploration and inspection of the robot's surroundings.

4) Configuration and Customization: RViz offers extensive configuration options to tailor the visualization to specific needs. Users can customize the layout, display settings, and appearance of various elements in the RViz interface. They can save and load configurations for quick setup and reuse. RViz also supports plugins, allowing users to extend its capabilities by adding custom visualization types or integrating with other ROS tools.

5) Integration with ROS: RViz seamlessly integrates with ROS, making it easy to subscribe to ROS topics and display relevant data in real-time. Users can visualize data published on ROS topics without writing additional code. RViz leverages the ROS tf library to handle coordinate frame transformations, enabling the visualization of data in different reference frames.

6) Simulation and Planning: RViz can be used in conjunction with other ROS tools and libraries for simulation and planning purposes. For example, it can display simulated sensor data, visualize the results of path planning algorithms, or show the predicted motion of a robot based on a given trajectory. This helps users evaluate and validate the behavior and performance of robotic systems in simulated environments.

RViz is a versatile and widely used visualization tool in ROS, providing an intuitive interface for understanding and debugging robotic systems. Its rich set of features and integration with ROS make it a valuable asset for visualizing and analyzing sensor data, robot behavior, and simulation results.

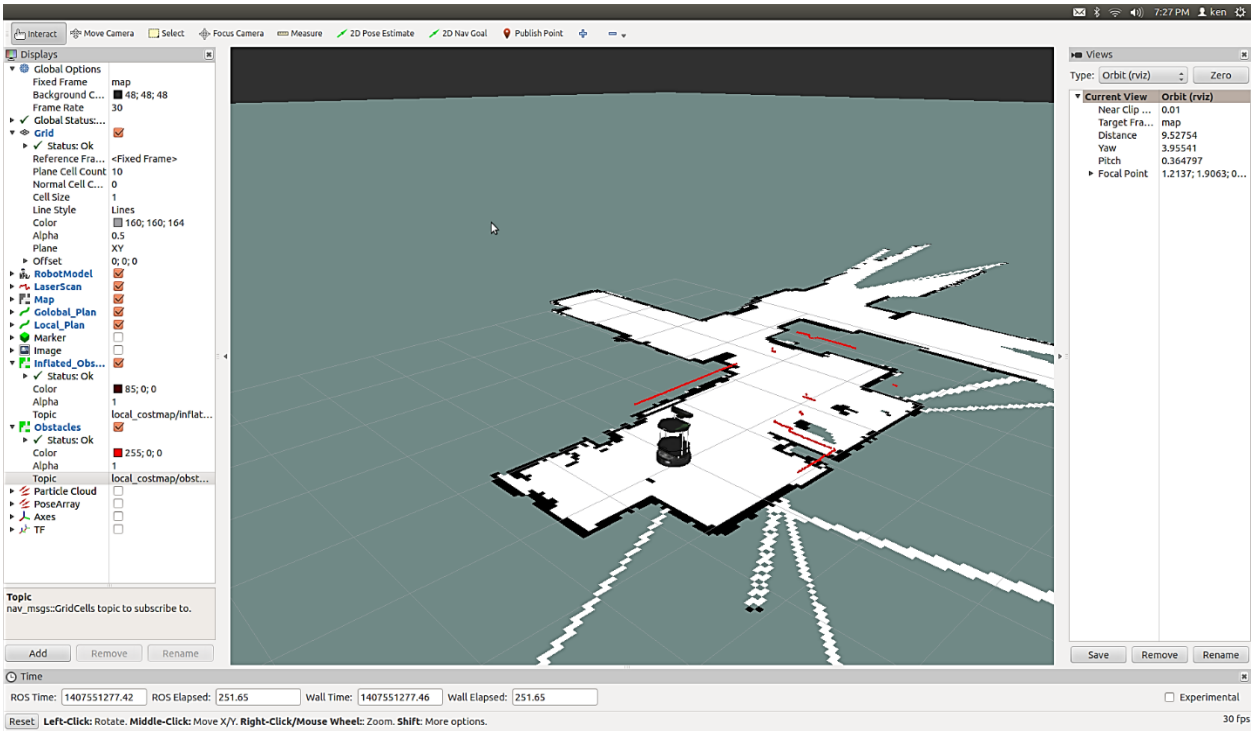


Figure 61 (2.5.4.5.2) Rviz

2.5.4.5.3. Debugging Tools in ROS2

ROS 2 provides several debugging tools and utilities to assist in the development and debugging of robotic applications. These tools help diagnose issues, monitor system behavior, and analyze the performance of ROS 2 components. Here are some commonly used debugging tools in ROS 2:

1. rqt_console: rqt_console is a graphical tool that displays log messages generated by ROS nodes. It allows users to monitor and filter log messages based on severity levels, node names, or specific topics. rqt_console helps identify errors, warnings, and other informative messages generated during the execution of ROS nodes.

2. rqt_graph: rqt_graph provides a visual representation of the ROS computation graph. It displays the nodes, topics, and connections between them. This tool is useful for inspecting the structure and communication patterns within a ROS system, helping to identify missing connections or unexpected interactions.

3. rviz: As mentioned earlier, RViz is a powerful visualization tool in ROS 2. It can be used for debugging purposes by visualizing sensor data, robot states, and other relevant information. RViz allows users to inspect the behavior of robotic systems, validate sensor readings, and debug complex robot behaviors in a 3D environment.

4. ROS2 topic echo: The `ros2 topic echo` command-line tool enables users to subscribe to a specific topic and display the messages being published on that topic. It is useful for observing the content and frequency of messages being exchanged between nodes, helping to identify data inconsistencies or unexpected behavior.

5. ROS2 node info: The `ros2 node info` command-line tool provides information about a specific ROS node. It displays details such as the list of topics, services, and actions that the node advertises or subscribes to. This tool helps to verify the configuration and connections of a node in the ROS system.

6. ROS2 bag: The `ros2 bag` command-line tool allows users to record and play back ROS topics, capturing data streams for offline analysis. It enables

users to record sensor data, logs, and other messages during runtime and later replay them for debugging or analysis purposes.

7. ROS Debugging Tools: Apart from ROS 2-specific tools, general debugging techniques and tools can also be used in ROS 2 development. These include using standard debugging tools like GDB (GNU Debugger) or integrated development environments (IDEs) that provide debugging capabilities for C++ or Python code.

These debugging tools and utilities help ROS 2 developers diagnose issues, analyze system behavior, and ensure the proper functioning of robotic applications. They facilitate the identification of errors, monitoring of ROS components, and verification of data exchanged between nodes, leading to more effective debugging and improved system performance.

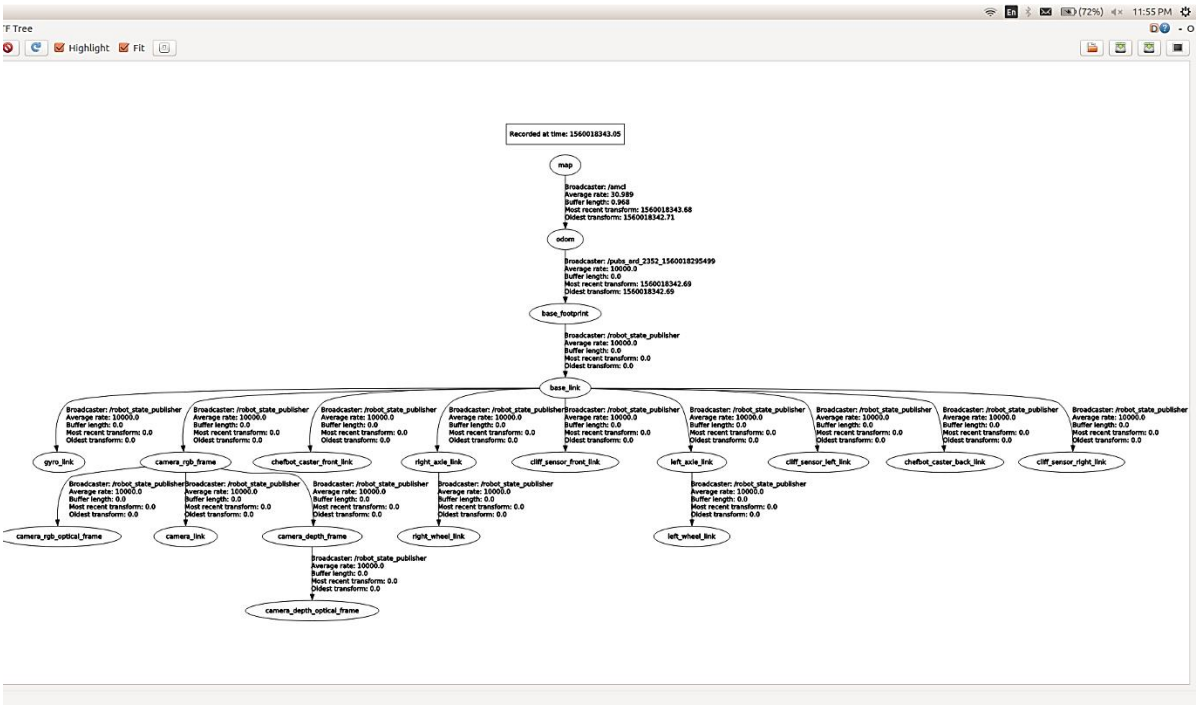


Figure 62 - (2.5.4.5.3) rat tree

2.5.4.6. Robot Algorithms & Concepts

2.5.4.6.1. Frame Transformation:

We can't use the sensory data without knowing where it is referenced to. And we may have the sensor connect with a little shift or in a position different form we must read data from. So, we perform Frame transformation to this data to the position we want the data to be read from.

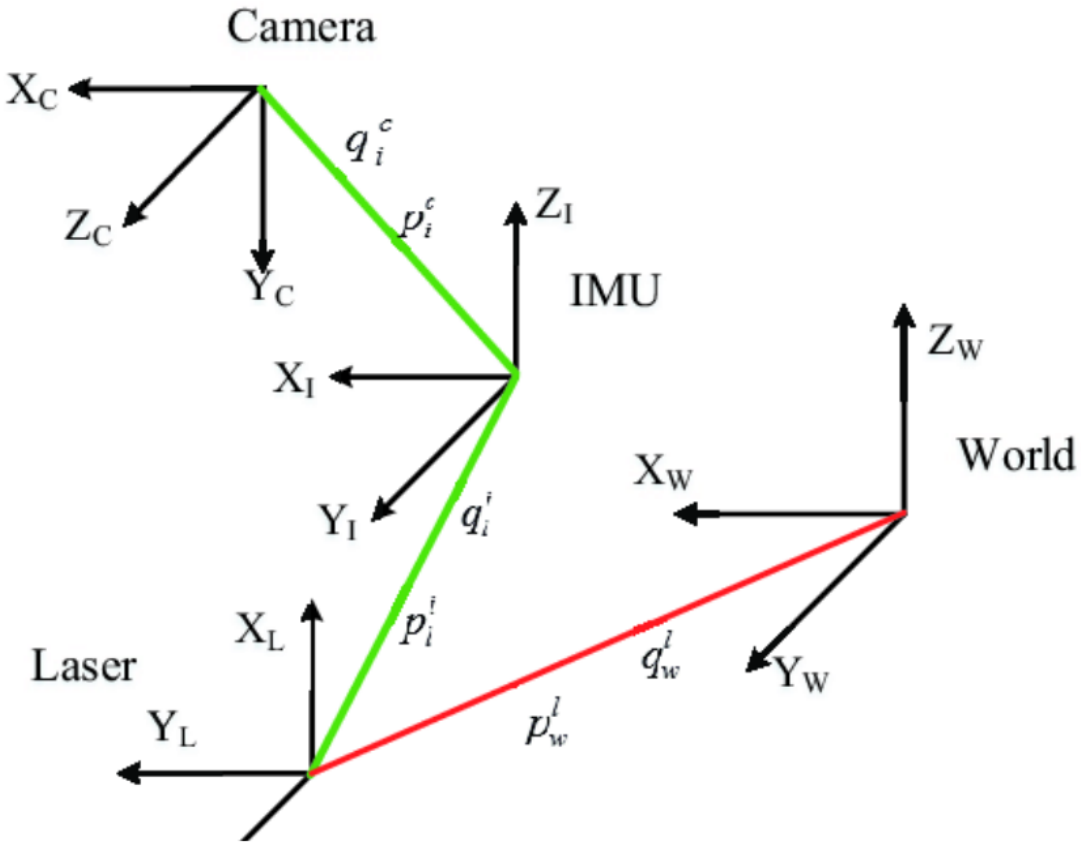


Figure 63 (2.5.4.6.1.a) Sensors Frame Transformation to the Reference

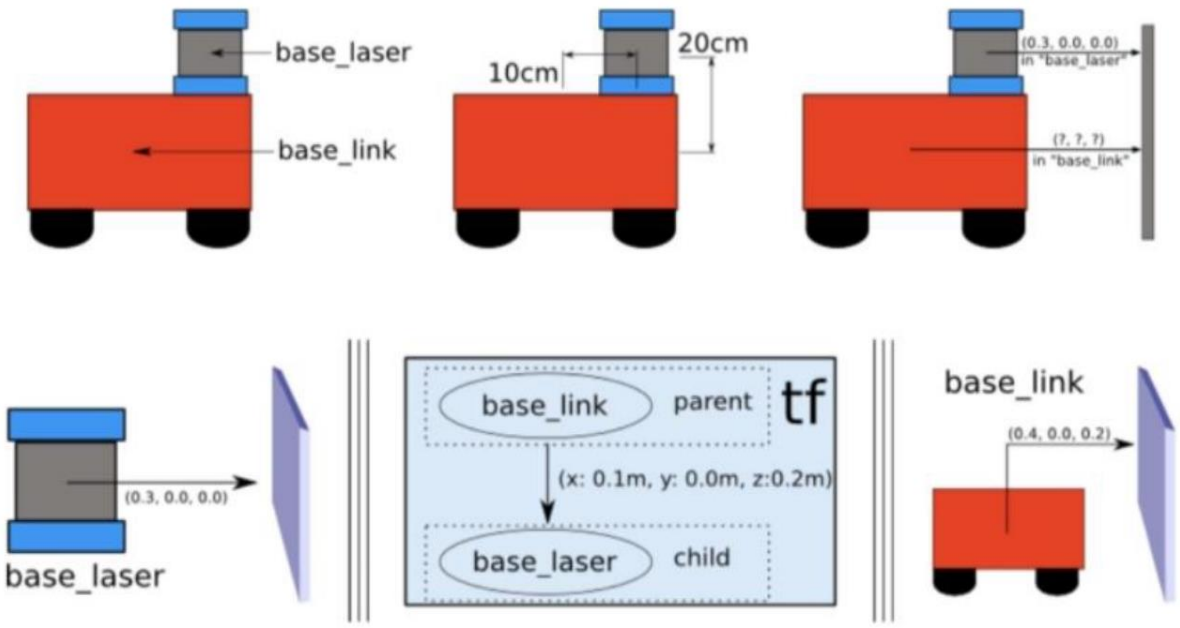


Figure 64 (2.5.4.6.1.b) Lidar Frame Transformation to base link

2.5.4.6.1.1. Rotation

A rotating frame of reference is a special case of a non-inertial reference frame that is rotating relative to an inertial reference frame. An everyday example of a rotating reference frame is the surface of the Earth.

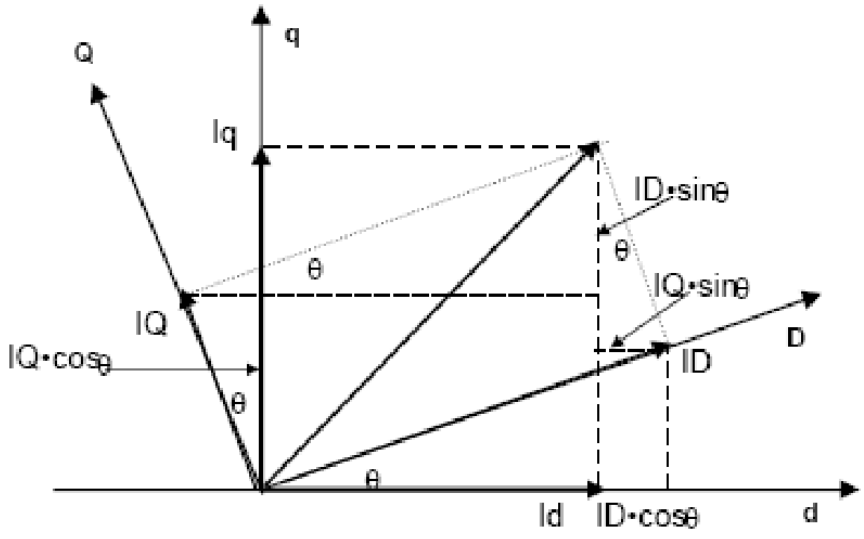


Figure 65 (2.5.4.5.1.1) Frame Rotation.

The following equations are used to rotate the readings of the sensors around the principal axes.

$$Rot(y,\emptyset) = \begin{bmatrix} C\emptyset & 0 & S\emptyset \\ 0 & 1 & 0 \\ -S\emptyset & 0 & C\emptyset \end{bmatrix}$$

$$Rot(z,\theta) = \begin{bmatrix} C\theta & -S\theta & 0 \\ S\emptyset & C\emptyset & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$Rot(x,\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & C\alpha & -S\alpha \\ 0 & S\alpha & C\alpha \end{bmatrix}$$

2.5.4.6.1.2. Transformation

Transformation is the rotation and translation add to it. Using these two operations we can move all sensory data to any position reference to another position.

$$\begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{xx} & a_{xy} & a_{xz} & a_{xt} \\ a_{yx} & a_{yy} & a_{yz} & a_{yt} \\ a_{zx} & a_{zy} & a_{zz} & a_{zt} \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix}$$

2.5.4.6.2. PID Controller:

The Proportional-Integral-Derivative (PID) controller is the most common control algorithm used for this application. It can correct the present error through proportional action, eliminate steady-state offsets through integral action, and better estimate the future trends through a derivative action.

A closed-loop system like a PID controller includes a feedback control system. This system evaluates the feedback variable using a fixed point to generate an error signal. Based on that, it alters the system output. This procedure will

continue till the error reaches Zero otherwise the value of the feedback variable becomes equivalent to a fixed point. PID controller uses three basic control behavior.

The following two equations represent PID equation with time domain and S domain (taking Laplace transform) respectively.

$$u(t) = K_p e(t) + K_i \int_0^t e(t')dt' + K_d \frac{de(t)}{dt}$$

$$K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s}$$

At time domain equation u(t) is the control signal to each motor; e(t) is the error regarding the tangential velocity of each wheel; and Kp, Ki and Kd, all non-negative, denote the coefficients for the proportional, integral, and derivative terms, respectively.

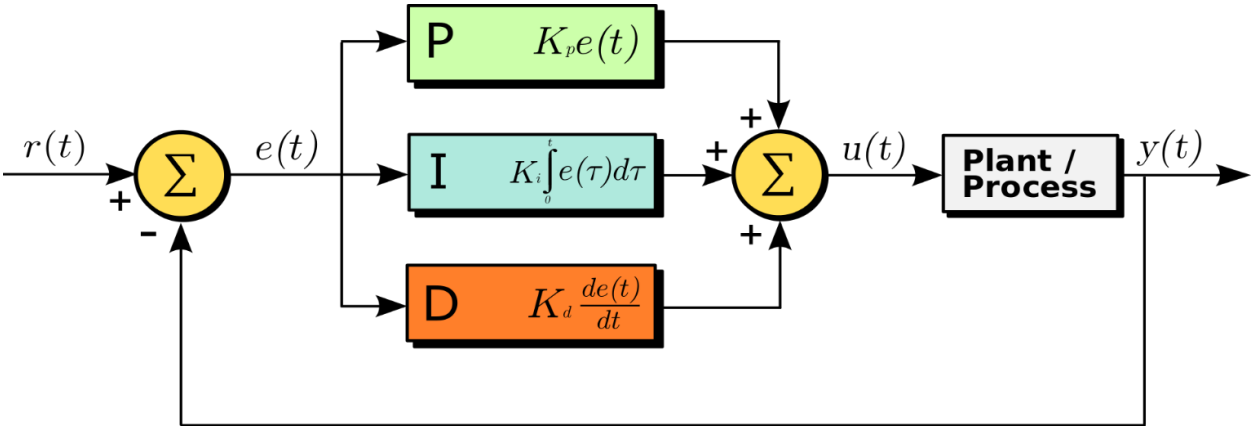


Figure 66 (2.5.6.2.a) PID controller

Controller parameter	Rise time	Overshoot	Stelling time	S – S Error
K_P	Decrease	Increase	Small change	Decrease
K_i	Decrease	Increase	Increase	Decrease
K_d	Small change	Decrease	Decrease	No change

Table 11 (2.5.6.2.b) Effect of controller parameters

2.5.4.6.2.1 Proportional (P) – controller:

Proportional (P) – controller gives an output that is proportional to current error e (t). It compares the desired or set point with the actual value or feedback process value. The resulting error is multiplied with a proportional constant to get the output. If the error value is zero, then this controller output is zero.
This controller requires biasing or manual reset when used alone. This is because it never reaches the steady-state condition. It provides stable operation but always maintains the steady-state error.

Increasing the proportional gain K_P has the effect of proportionally increasing the control signal for the same level of error. The fact that the controller will "push" harder for a given level of error tends to cause the closed-loop system to react more quickly, but also to overshoot more. Another effect of increasing K_P is that it tends to reduce, but not eliminate, the steady state error.

2.5.4.6.2.2. Integral (I) controller:

Due to the limitation of P-controller where there always exists an offset between the process variable and setpoint, I-controller is needed, which provides necessary action to eliminate the steady-state error. It integrates the error over a

period until the error value reaches zero. It holds the value to the final control device at which error becomes zero.

Integral control decreases its output when a negative error takes place. It limits the speed of response and affects the stability of the system. The speed of the response is increased by decreasing integral gain, Ki.

Also, the addition of integral control K_i tends to decrease the rise time, increase both the overshoot and the settling time, and reduces the steady state error. PI controller is used particularly where the high-speed response is not required. While using the PI controller, I-controller output is limited to somewhat range to overcome the integral wind-up conditions where the integral output goes on increasing even at zero error state, due to nonlinearities in the plant.

2.5.4.6.2.3. Derivative (D) control:

I-controller doesn’t have the capability to predict the future behavior of error. So, it reacts normally once the setpoint is changed. D-controller overcomes this problem by anticipating the future behavior of the error. Its output depends on the rate of change of error with respect to time, multiplied by derivative constant. It gives the kick start for the output thereby increasing system response.

The addition of a derivative term to the controller K_d adds the ability of the controller to "anticipate" error. With simple proportional control, if K_d is fixed, the only way that the control will increase is if the error increases. With derivative control, the control signal can become large if the error begins sloping upward, even while the magnitude of the error is still relatively small. This anticipation tends to add damping to the system, thereby decreasing overshoot. The addition of a derivative term, however, has no effect on the steady-state error.

2.5.4.6.4.4. Types of PID controller based on control system:

1. An on-off control method:

The simplest type of device used for temperature control. The device output may be ON/OFF through no center state. This controller will turn ON the output simply once the temperature crosses the fixed point.

2. Proportional control:

This kind of controller is designed to remove the cycling which is connected through ON/OFF control. This PID controller will reduce the normal power which is supplied toward the heater once the temperature reaches the fixed point. This controller has one feature to control the heater so that it will not exceed the fixed point however it will reach the fixed point to maintain a steady temperature.

3. Standard Type PID Controller:

This kind of PID controller will merge proportional control through integral & derivative control to automatically assist the unit to compensate modifications within the system. These modifications, integral & derivative are expressed in time-based units.

2.5.4.6.2.5 Tuning of PID:

The most common way for now is automatic tuning where MATLAB provides tools for automatically choosing optimal PID gains which makes the trial-and-error process is unnecessary. By accessing the tuning algorithm directly using pidtune or through a nice graphical user interface (GUI) using pidTuner. The MATLAB automated tuning algorithm chooses PID gains to balance performance (response time, bandwidth) and robustness (stability margins). By default, the algorithm designs for a 60-degree phase margin.

2.5.4.6.3. ros2_control Library [44][45]:

The ros2_control library is a core component of the Robot Operating System 2 (ROS 2) that provides a standardized interface and framework for robot control. It aims to simplify the development and integration of control algorithms and hardware interfaces for robotic systems in a consistent and modular manner. The ros2_control library offers several key features and benefits:

1) Hardware Abstraction: ros2_control provides a hardware interface layer that abstracts the underlying robot hardware. This allows developers to write control code independently of the specific robot hardware, making it easier to port and reuse control algorithms across different robots.

2) Controller Interfaces: ros2_control defines standardized controller interfaces for different types of robot joints, such as position, velocity, and effort (torque) control. These interfaces enable the implementation of control algorithms that can seamlessly work with different types of joints and robots.

3) Real-Time Safe: ros2_control is designed to be real-time safe, ensuring that control algorithms can run predictably within specified time constraints. It leverages the Real-Time Publish-Subscribe (RTPS) protocol provided by ROS 2 for efficient and timely communication between the controller and the robot hardware.

4) Hardware Integration: ros2_control provides a modular architecture that allows for easy integration of different robot hardware. It supports various communication protocols and mechanisms, such as EtherCAT, CAN bus, or direct memory access (DMA), to interface with different hardware components like actuators, sensors, and motor drivers.

5) Flexible Configuration: ros2_control allows for flexible configuration of control systems. It supports the dynamic loading and unloading of controller configurations, making it possible to switch between different control algorithms and configurations during runtime without stopping the robot.

6) Safety and Fault Handling: ros2_control provides mechanisms for handling safety and fault conditions. It supports features such as joint limits, soft limits, and safety controllers to prevent robot damage and ensure safe operation. It also allows for the implementation of fault handling strategies, such as error recovery and emergency stop procedures.

7) Integration with ROS 2 Ecosystem: ros2_control seamlessly integrates with other ROS 2 components and tools. It leverages the ROS 2 middleware for communication, making it compatible with ROS 2 message types and topics. It also works well with visualization tools like RViz and analysis tools like ros2 bag for debugging and analysis.

By using the ros2_control library, developers can focus on designing control algorithms and behavior without worrying about the low-level details of hardware integration and communication. It promotes code reusability, modular design, and

standardized interfaces, making it easier to develop, test, and deploy control systems for a wide range of robotic platforms and applications in ROS 2.

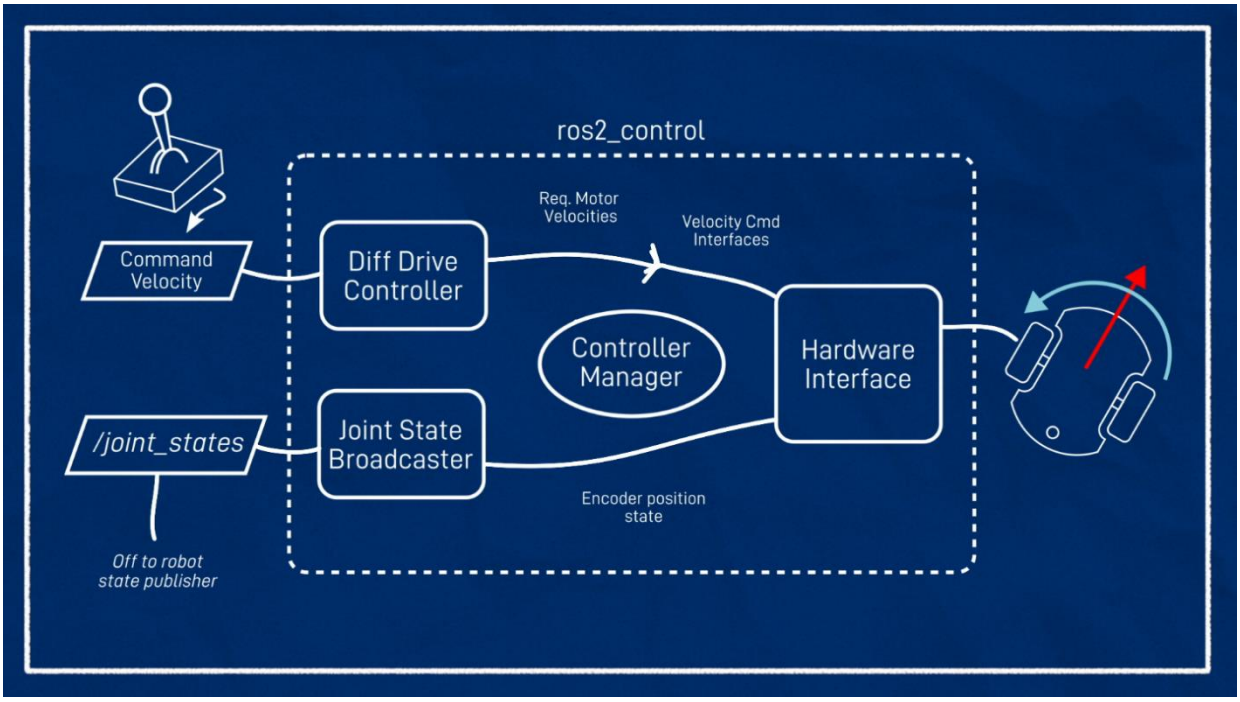


Figure 67 (2.5.4.6.3.a) ros2_control

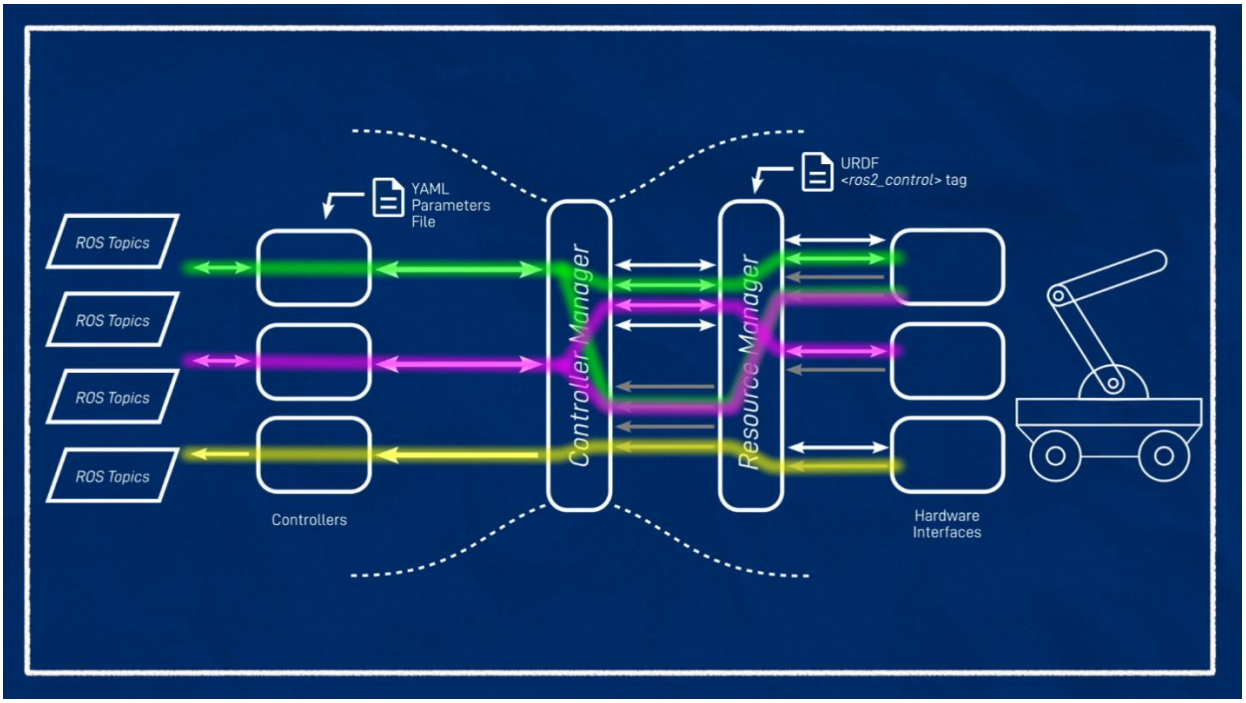


Figure 68 (2.5.4.6.3.b) ros2 controller manager

2.5.4.6.4 Perception [51]

In robotics, perception is understood as a system that endows the robot with the ability to perceive, comprehend, and reason about the surrounding environment. The key components of a perception system are essentially sensory data processing, data representation (environment modeling), and ML-based algorithms.

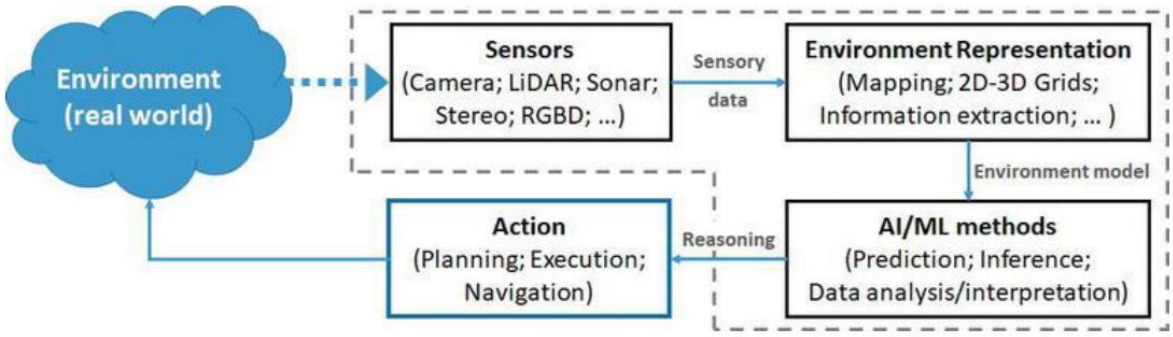


Figure 69 (2.5.4.6.7) typical robotic perception system

Sensor-based environment representation/mapping is a very important part of a robotic perception system. Mapping here encompasses both the acquisition of a metric model and its semantic interpretation and is therefore a synonym of environment/scene representation. This semantic mapping process uses ML at various levels, e.g., reasoning on volumetric occupancy and occlusions, or identifying, describing, and matching optimally the local regions from different timestamps/models, i.e., not only higher-level interpretations. However, in the majority of applications, the primary role of environment mapping is to model data from exteroceptive sensors, mounted onboard the robot, in order to enable reasoning and inference regarding the real-world environment where the robot operates.

2.5.4.6.4.1. Data [52]

In this topic we will talk about the sensory data our robot precept and how filter it is using famous techniques like Extended Kalman Filter (EKF), Particle Filter and many other gaussians based error handling ways.

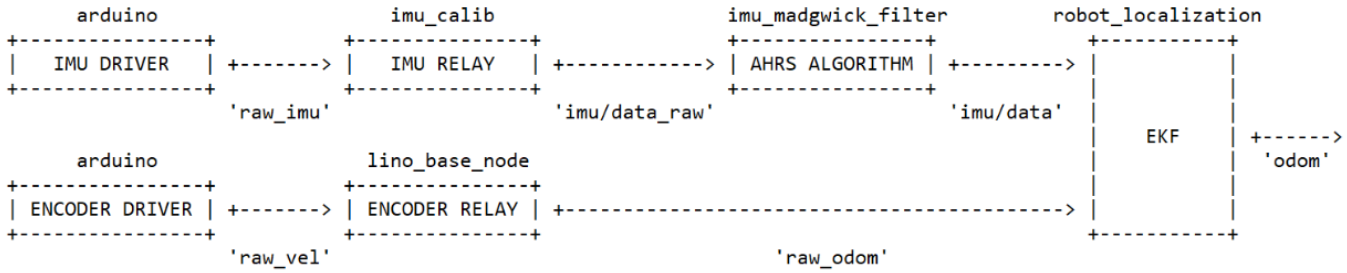


Figure 70 (2.5.4.6.4.1) data flow from each node and packages used before it reaches the filter.

Odometry information is used to estimate the robot's position relative to its origin. Primarily, Our Turtle Robot's linear and angular velocity found in the odometry data, published in "raw_odom", is calculated by counting the change in number of ticks over time. However, dead-reckoning that is solely based on motor encoders could be prone to errors due to system noise and wheel slippage. As a remedy, these velocities are fused with the orientation data produced by the Inertial Measurement Unit (IMU), published in "imu/data" topic, using an Extended Kalman Filter (EKF) through robot_localization package. This provides the robot an educated guess of its current pose when a sensor gets too noisy or when data starts missing during estimation.

2.5.4.6.4.2 Filtering

As mentioned, we deal with many sensors to percept our environment, but each sensor has its own uncertainty and mixing them together will lead to more unnecessary uncertainty and errors that will be hard to handle. So, we pass these data on some stages before fusion this sensory data.

In this section we will talk about these data and how to handle them.

2.5.4.6.4.2.1 LIDAR

Lidar (Light Detection and Ranging) is a technology that uses laser light to measure distances and create detailed 3D maps of the surrounding environment. In the context of perception in autonomous robots, lidar plays a crucial role by providing accurate and real-time information about the robot's surroundings. Here are some key aspects of lidar's contribution to perception in autonomous robots:

1) Obstacle Detection and Avoidance: Lidar sensors emit laser beams and measure the time it takes for the light to bounce back after hitting an object. This data is used to create a point cloud representation of the environment, which helps the robot detect and localize obstacles such as pedestrians, vehicles, buildings, or other objects. By analyzing the point cloud, the robot can plan its path, avoid collisions, and navigate safely.

2) Mapping and Localization: Lidar allows robots to build detailed maps of their environment. By continuously scanning the surroundings, the lidar sensor generates a point cloud that represents the geometric structure of the environment. This information is crucial for simultaneous localization and mapping (SLAM) algorithms, which enable the robot to determine its own position within the map and update it as it moves. Lidar-based SLAM algorithms are often used in autonomous navigation systems.

3) Object Recognition: Lidar data can be used to identify and classify objects in the environment. By analyzing the shape, size, and location of the points in

the point cloud, machine learning algorithms can recognize and categorize various objects, such as cars, pedestrians, traffic signs, or trees. This object recognition capability is essential for decision-making processes in autonomous robots, enabling them to understand their surroundings and make appropriate responses.

4) Environmental Perception: Lidar provides rich spatial information that complements other perception sensors like cameras or radar. By combining data from multiple sensors, the robot can obtain a more comprehensive understanding of its surroundings. Lidar's ability to directly measure distances and generate accurate 3D representations makes it particularly valuable for perceiving the environment in scenarios with complex geometries or poor lighting conditions.

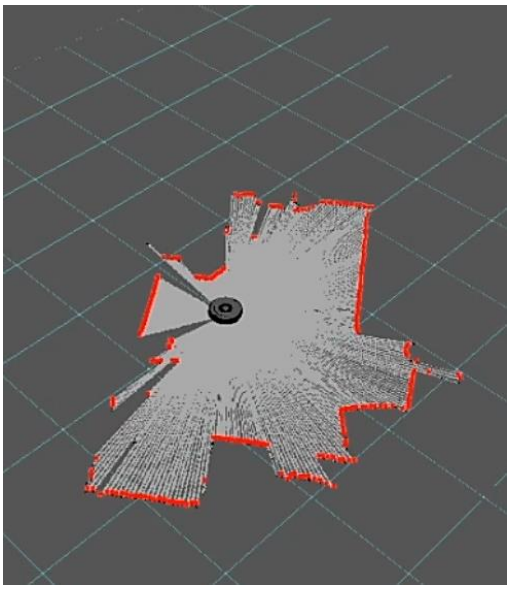


Figure 71 (2.5.4.6.4.2.1) LIDAR perception

2.5.4.6.4.2.2 Encoder Data

Encoder sensor produces pulses as the robot moves. If we counted these pulses, we could estimate the position of the robot by reference to the starting point. We count these pulses in each time interval to estimate the current speed of the motor. But as we established, we need odometry as feedback to our motion and it's a Quaternion frame not a speed or pulses, so we pass these raw speed values to a node to apply numerical integration to the raw velocity, this will result in the desired frame.

$$\omega = [0, \omega_1, \omega_2, \omega_3\ ]$$

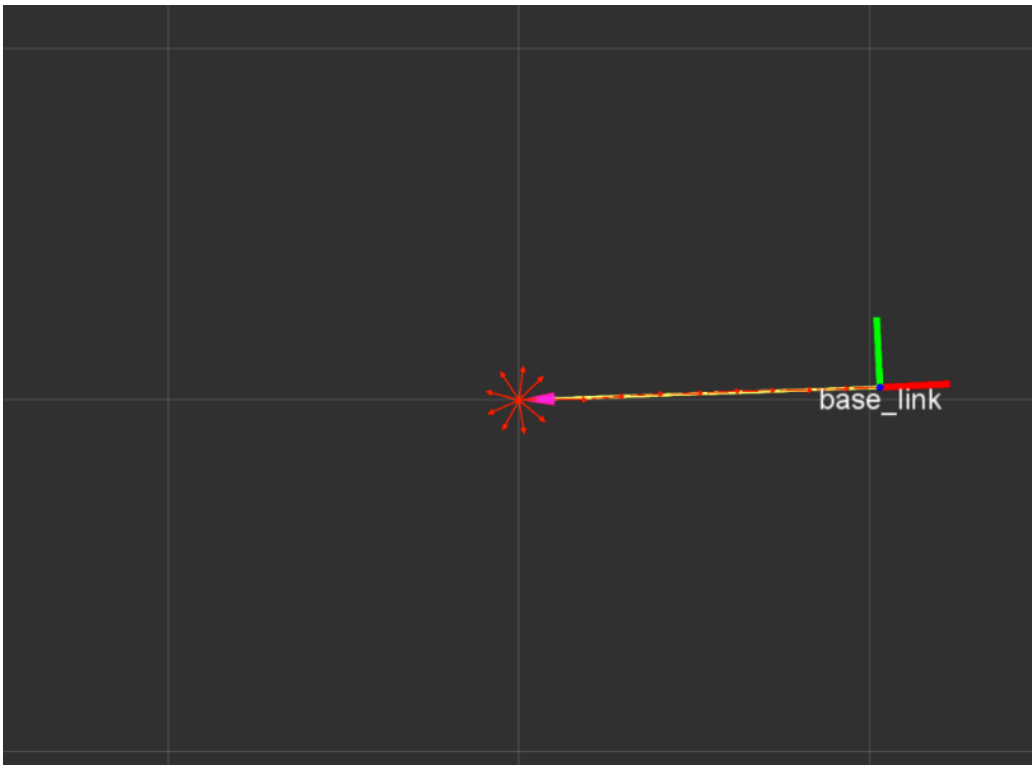


Figure 72 (2.5.4.6.4.2.2) Odometry frame reference to the Robot

$$\dot{q} = \frac{1}{2}\Omega q$$

$$\Omega = \begin{bmatrix} 0 & -\omega_1 & -\omega_2 & -\omega_3 \\ \omega_1 & 0 & \omega_3 & -\omega_2 \\ \omega_2 & -\omega_3 & 0 & \omega_1 \\ \omega_3 & \omega_2 & -\omega_1 & 0 \end{bmatrix}$$

2.5.4.6.4.2.3. IMU Data [53]

As we mentioned in the previous chapter, IMU sensor generates 9 Values; 3 for angular rotation speed and 3 for linear acceleration. And finally, 3 values indicating the magnetic field in the linear direction.

As we did with the encoder data, we want to filter this data and convert it to Quaternion frame of the odometry. So, we use two stages of filtering. First a calibration bias that we add to the reading due to the shifts and variance we may add due to mis fixing the sensor on the robot with a shifted angle. And the second and the most important is the "IMU Madgwick Filter" which applies quaternion transformation to result with the odometry frame.

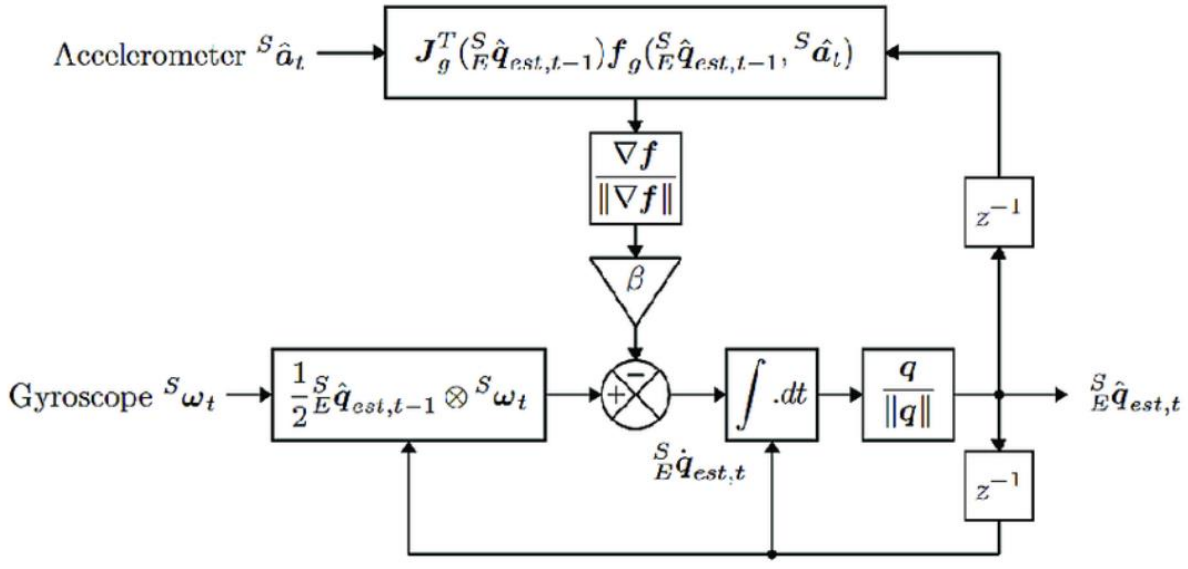


Figure 73 (2.5.4.6.4.2.3) MU Madgwick Filter Block Diagram.

2.5.4.6.4.3 Fusion

Now we have two sources of odometry but off course they have their own uncertainty, and we need to get rid of this error or at least minimize it. In doing so we will use a technique called Extended Kalman filter (EFK).

2.5.4.6.4.3.1 Extended Kalman Filter (EKF) [47][54]

A Kalman filter is a mathematical algorithm used for state estimation in systems that involve uncertain measurements and noisy sensor data. It is a recursive filter that combines predictions from a dynamic model with measurements to estimate the true state of a system. The Kalman filter is widely used in various fields, including robotics, and it plays a significant role in Simultaneous Localization and Mapping (SLAM) algorithms. Here's an explanation of the Kalman filter and its use in SLAM:

1) State Estimation: The Kalman filter is designed to estimate the state of a system based on available measurements and predictions. In SLAM, the state represents the robot's pose (position and orientation) and the map of the environment. The filter maintains an estimate of the current state and updates it as new measurements are obtained.

2) Prediction Step: The Kalman filter predicts the next state based on the system's dynamic model and the previous state estimate. It uses the motion model and control inputs (e.g., velocity, acceleration) to forecast the expected state of the robot at the next time step. The prediction incorporates uncertainties and noise associated with the dynamics of the system.

3) Measurement Update Step: Once a new measurement is obtained from sensors (e.g., range measurements from a laser scanner), the Kalman filter incorporates this information to update the state estimate. It compares the predicted measurement based on the estimated state with the actual measurement and adjusts the state estimate accordingly, taking into account the uncertainty in the measurements and the predicted state.

4) Covariance Matrix: The Kalman filter maintains a covariance matrix that represents the uncertainty associated with the state estimate. It quantifies the covariance between different elements of the state and reflects the uncertainty in the estimation. As new measurements are incorporated, the covariance matrix is updated to reflect the reduced uncertainty in the estimated state.

5) Optimal Estimation: The Kalman filter is an optimal estimator when the system dynamics and measurement models are linear and the noise follows Gaussian distributions. It provides the best linear unbiased estimate (BLUE) of the state by minimizing the mean squared error between the true state and the estimated state.

In SLAM, the Kalman filter is commonly used to estimate the robot's pose and simultaneously build a map of the environment. It integrates motion model predictions with measurements from sensors (e.g., laser scanners or cameras) to refine the state estimate and the map. By continuously updating the state estimate and refining the map, the Kalman filter enables the robot to simultaneously localize itself and map the environment in real-time.

The Kalman filter's ability to handle uncertainties, incorporate measurements, and perform recursive estimation makes it a powerful tool for state estimation in SLAM algorithms. It helps improve the accuracy and consistency of the state estimate and the resulting map, facilitating robust localization and mapping in dynamic environments.

In the extended Kalman filter, the state transition and observation models don't need to be linear functions of the state but may instead be differentiable functions.

$$x_k = f(x_{k-1}, u_k) + w_k$$
$$z_k = h(x_k) + v_k$$

Here w_k and v_k are the process and observation noises which are both assumed to be zero mean multivariate Gaussian noises with covariance Q_k and R_k respectively. u_k is the control vector. The function f can be used to compute the predicted state from the previous estimate and similarly the function h can be used to compute the predicted measurement from the predicted state. However, f and h cannot be applied to the covariance directly. Instead, a matrix of partial derivatives (the Jacobian) is computed.

At each time step, the Jacobian is evaluated with current predicted states. These matrices can be used in the Kalman filter equations. This process essentially linearizes the non-linear function around the current estimate.

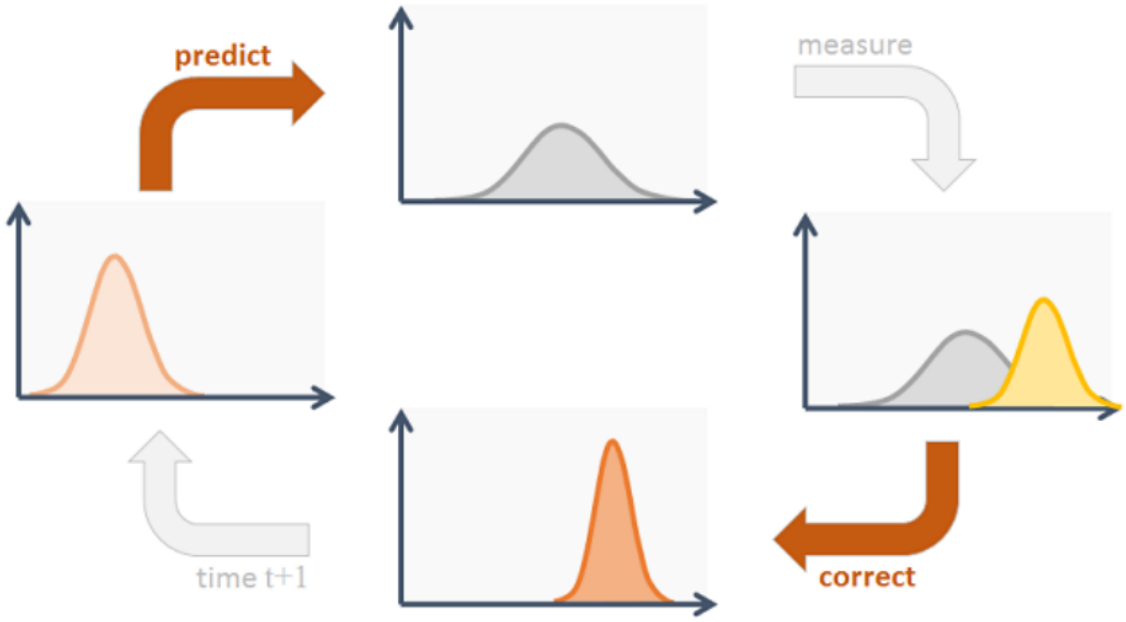


Figure 74 (2.5.4.6.4.3.1.a) Kalman Filter predict, measure, and update process.

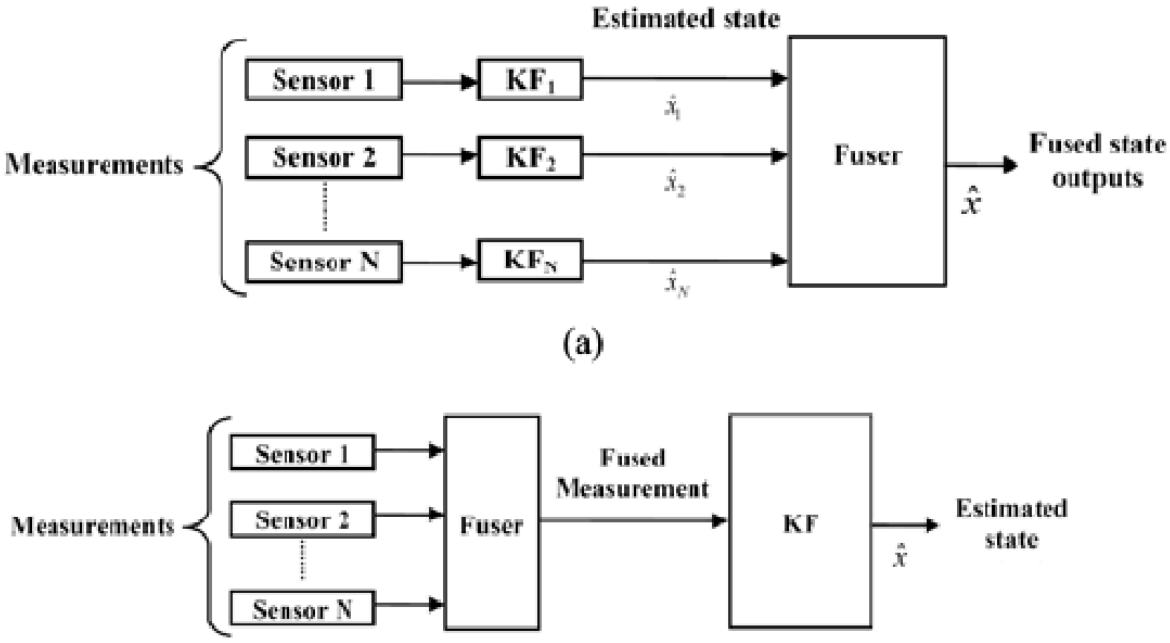


Figure 75 (2.5.4.6.4.3.1.b) Sensor fusion block Diagram Using Kalman Filter

2.5.4.6.5. Navigation

In robotic mapping and navigation, simultaneous localization, and mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. While this initially appears to be a chicken-and-egg problem there are several algorithms known for solving it, at least approximately, in tractable time for certain environments. Popular approximate solution methods include the particle filter, extended Kalman filter, and Graph SLAM.

SLAM algorithms are tailored to the available resources, hence not aimed at perfection, but at operational compliance. Published approaches are employed in self-driving cars, unmanned aerial vehicles, autonomous underwater vehicles, planetary rovers, newer domestic robots and even inside the human body.

2.5.4.6.5.1. Problem Formation [55]

Given a series of sensor observations O_t, over discrete time steps t, the SLAM problem is to compute an estimate of the agent's location X_t, and a map of the environment m_t. All quantities are usually probabilistic, so the objective is to compute:

$$P(m_t, x_t | o_{1:t})$$

Applying Bayes 'rule gives a framework for sequentially updating the location posteriors, given a map and a transition function $P(x_t \mid x_{t-1})$.

$$P(x_t \mid o_{1:t}, m_t) = \sum_{m_{t-1}} P(o_t \mid x_t, m_t) \sum_{x_{t-1}} P(x_t \mid x_{t-1}) P(x_{t-1} \mid m_t, o_{1:t-1}) / Z$$

Similarly, the map can be updated sequentially by:

$$P(m_t \mid x_t, o_{1:t}) = \sum_{x_t} \sum_{m_t} P(m_t \mid x_t, m_{t-1}, o_t) P(m_{t-1}, x_t \mid o_{1:t-1}, m_{t-1})$$

Like many inference problems, the solutions to inferring the two variables together can be found, to a local optimum solution, by alternating updates of the two beliefs in a form of EM algorithm.

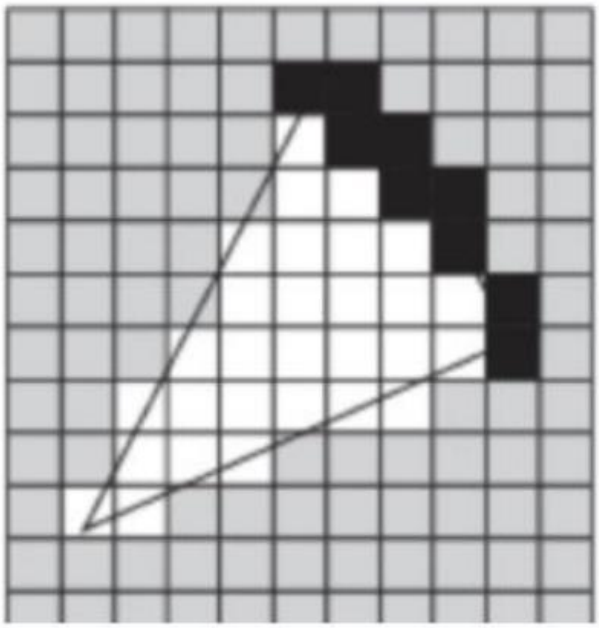


Figure 76 (2.5.4.6.5.1.a) Map Grid Cell Occupancy

So, we represent occupant grid cell with black and white grid cell as free and for unexplored areas they are colored with gray.

Now that we have constructed the map. And we want to know where our robot is on this map. This is known as Localization. Referring to locating the robot reference to the map center.

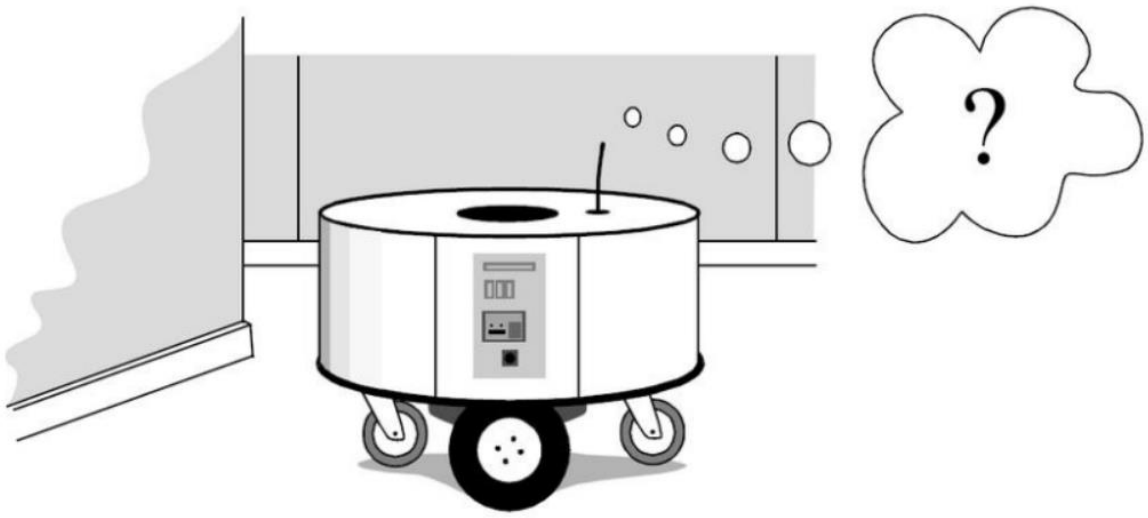


Figure 77 (2.5.4.6.5.1.b) Robot doesn't know where it is.

We Can use several methods such as Adaptive Monte Carlo Localization (AMCL) or Simultaneous localization and mapping (SLAM). AMCL is primarily focused on localization within a known map, utilizing sensor data to estimate the robot's pose accurately. It is beneficial when the map of the environment is available and the emphasis is on localizing the robot within that map. On the other hand, SLAM is a more comprehensive technique that simultaneously constructs a map of an unknown environment while estimating the robot's pose within that map. SLAM is advantageous when the robot needs to explore and build a map of an unfamiliar or dynamic environment. Both AMCL and SLAM play crucial roles in robotics applications, and the choice between them depends on the specific requirements and constraints of the given scenario.

2.5.4.6.5.2. Adaptive Monte Carlo Localization [56]

AMCL is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot against a known map.

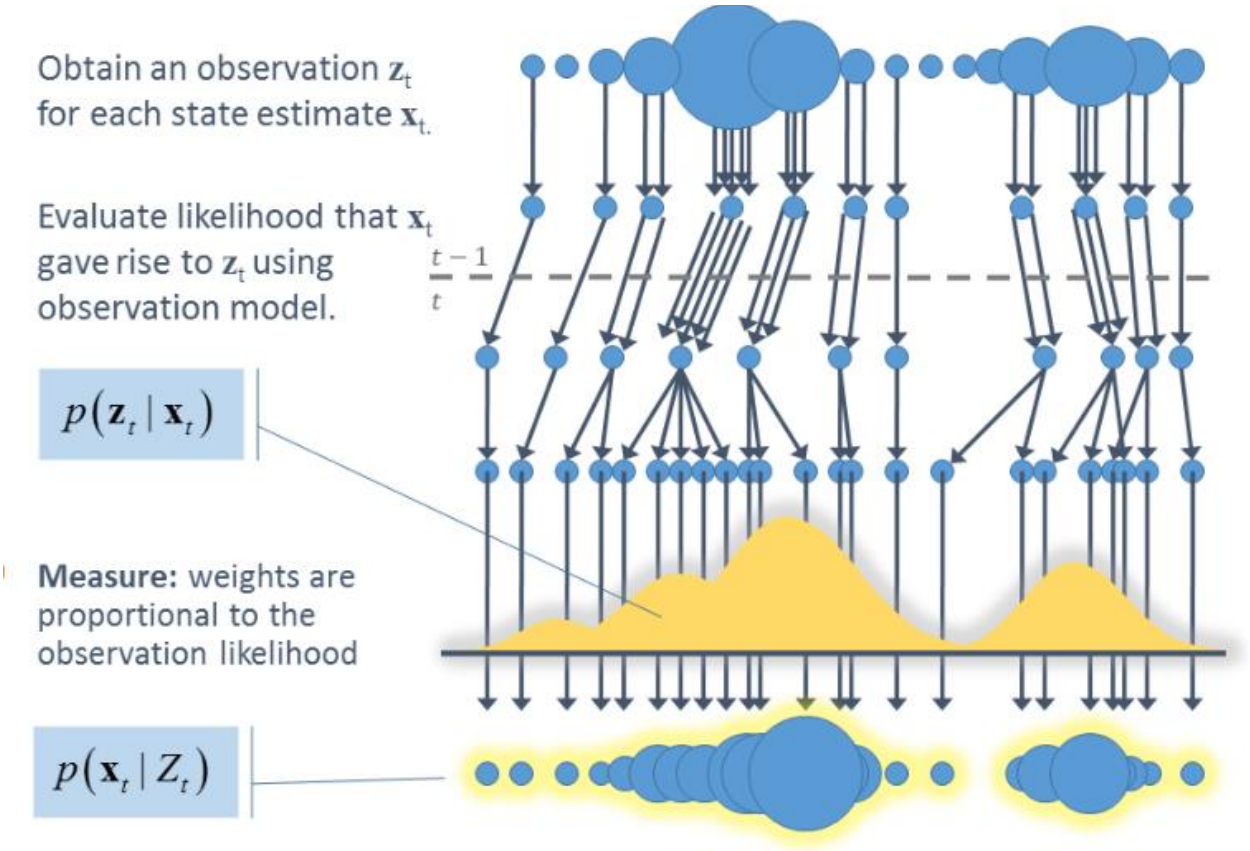


Figure 78 (2.5.4.6.5.2.a) Particle Filter sample Matching

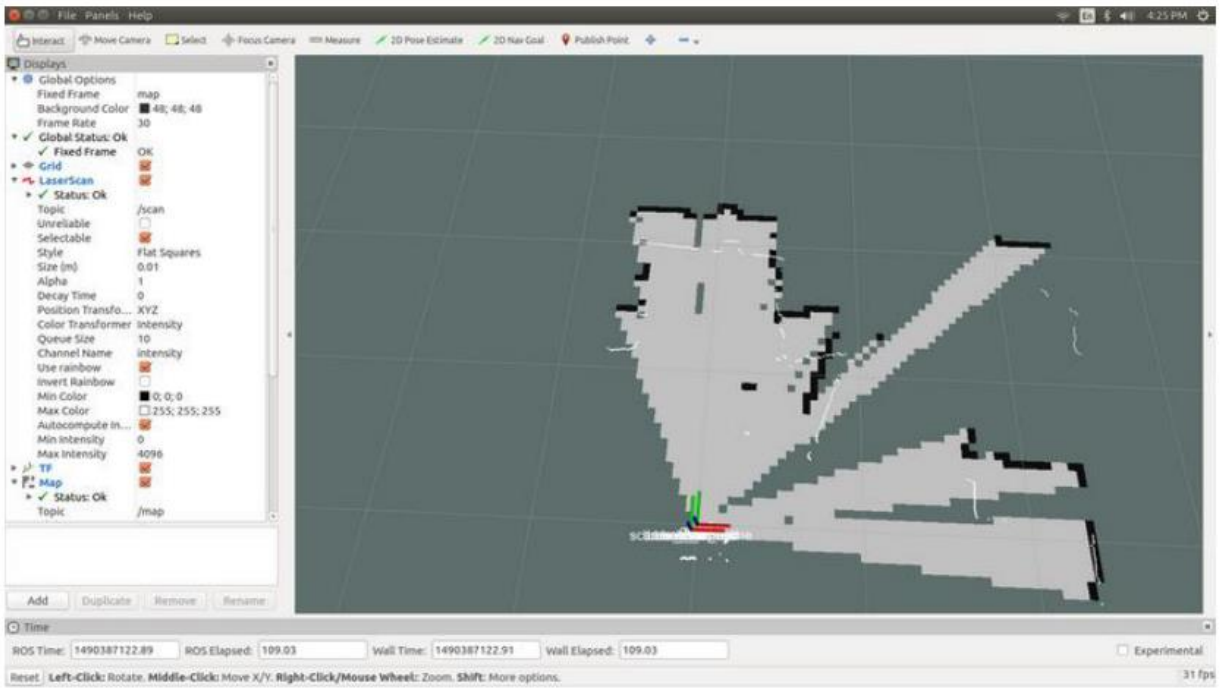


Figure 79 (2.5.4.6.5.2.b) Wrong Localization before Using AMCL.

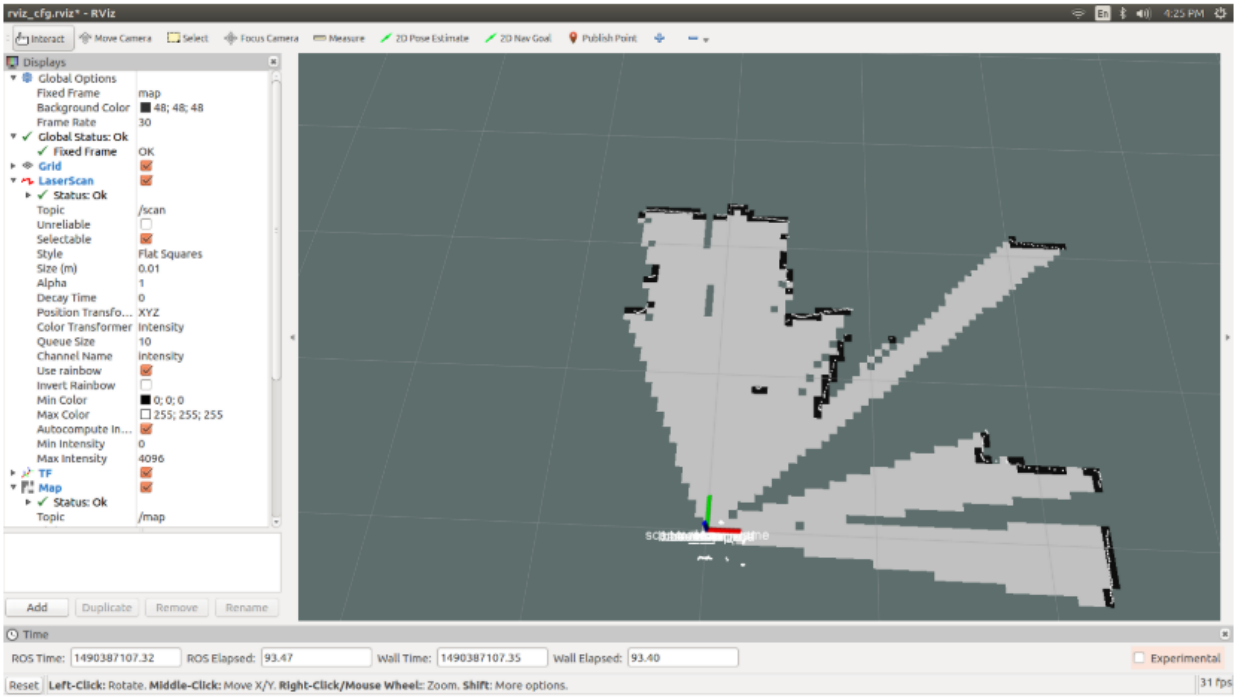


Figure 80 (2.5.4.6.5.2.c) Corrected position localized using AMCL.

The AMCL algorithm:

1- Initialize particle poses and weights according to known start-state information

2- Randomly re-sample particles based on previously computed weights

3- Update particle positions based on motion model and available motion sensor information

4- Update particle weights based on a sensor model by comparing hypotheses to the occupancy grid map and sensor readings

5- Compute expected value of robot pose based on the particle distribution 6- Repeat from step 2

2.5.4.6.5.3. Simultaneous Localization and Mapping (SLAM) [46]

SLAM (simultaneous localization and mapping) is a method used for autonomous vehicles that lets you build a map and localize your vehicle in that map at the same time. SLAM algorithms allow the vehicle to map out unknown environments. Engineers use the map information to carry out tasks such as path planning and obstacle avoidance.

SLAM has been the subject of technical research for many years. But with vast improvements in computer processing speed and the availability of low-cost sensors such as cameras and laser range finders, SLAM is now used for practical applications in a growing number of fields.

surroundings.

Here are the key components and concepts of SLAM:

1) Localization: Localization refers to estimating the robot's position and orientation (pose) within the map. By integrating sensor measurements and motion data over time, the robot tries to determine its pose relative to the environment. This process is often performed using techniques like odometry, which tracks the robot's movement based on wheel encoders or other motion sensors.

2) Mapping: Mapping involves constructing a representation of the environment using the sensor measurements collected by the robot. The robot combines sensor data (such as range measurements from a laser scanner or images from a camera) to build a map that represents the physical features and layout of the environment. The map can be represented in various forms, such as occupancy grids, point clouds, or feature-based representations.

3) Data Association: Data association refers to the process of associating sensor measurements with previously mapped features or landmarks in the environment. By identifying corresponding features between consecutive sensor readings, the robot can refine its map and reduce uncertainty in the localization process.

4) Loop Closure: Loop closure detection is a crucial step in SLAM. It involves identifying places or locations that the robot has previously visited and recognizing when it revisits those places. By detecting loop closures, the

robot can correct errors in its pose estimation and map, improving the overall accuracy and consistency of the SLAM solution.

5) Graph Optimization: SLAM often employs graph optimization techniques to refine the estimated poses and map based on the collected sensor data. Graph optimization algorithms minimize the inconsistencies and errors in the estimated poses by adjusting the robot's trajectory and optimizing the map based on the observed measurements.

6) Online and Offline SLAM: SLAM can be performed in real-time (online SLAM) as the robot moves through the environment or in a post-processing manner (offline SLAM) by analyzing recorded sensor data after the robot's exploration. Online SLAM is typically used for real-time navigation and mapping, while offline SLAM is useful for generating accurate maps and performing analysis.

SLAM has broad applications in robotics, including autonomous navigation, robotic mapping, augmented reality, and more. It enables robots to operate in unknown or changing environments, build accurate maps, and localize themselves within those maps. SLAM algorithms and techniques have been extensively researched and developed, leading to various approaches and implementations that cater to different robotic platforms and sensor configurations.

2.5.4.6.5.3.1. How SLAM Works

Broadly speaking, there are two types of technological components used to achieve SLAM. The first type is sensor signal processing, including the front-end

processing, which is largely dependent on the sensors used. The second type is pose-graph optimization, including the back-end processing, which is sensor-agnostic.

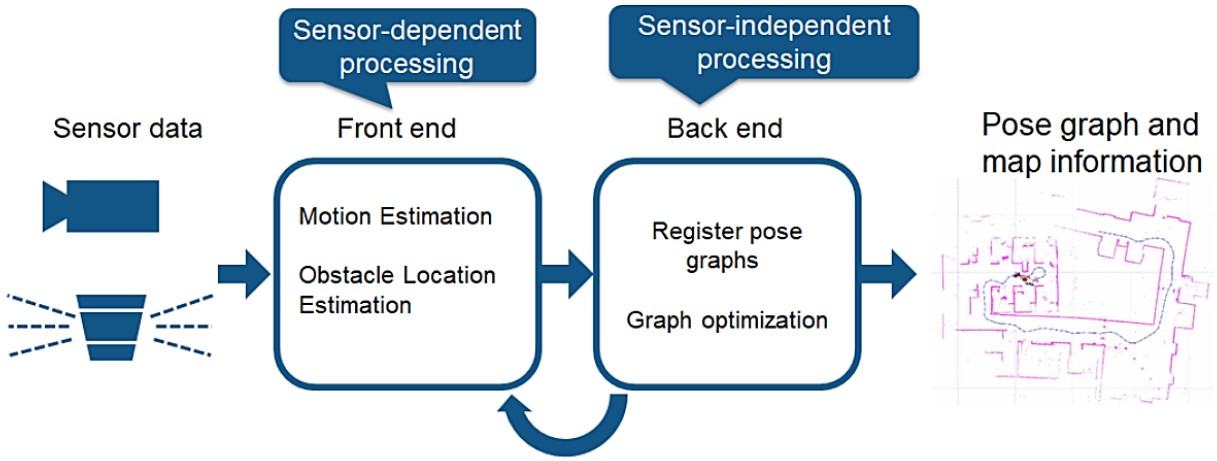


Figure 81 (2.5.4.6.5.3.1) How SLAM Works

2.5.4.6.5.3.2. LiDAR SLAM

Light detection and ranging (lidar) is a method that primarily uses a laser sensor (or distance sensor).

Compared to cameras, ToF, and other sensors, lasers are significantly more precise, and are used for applications with high-speed moving vehicles such as self-driving cars and drones. The output values from laser sensors are generally 2D (x, y) or 3D (x, y, z) point cloud data. The laser sensor point cloud provides high-precision distance measurements and works very effectively for map construction with SLAM. Generally, movement is estimated sequentially by matching the point clouds. The calculated movement (traveled distance) is used for localizing the vehicle. For lidar point cloud matching, registration algorithms such as iterative closest point (ICP) and normal distributions transform (NDT) algorithms are used. 2D or 3D point cloud maps can be represented as a grid map or voxel map.

On the other hand, point clouds are not as finely detailed as images in terms of density and do not always provide sufficient features for matching. For example, in places where there are few obstacles, it is difficult to align the point clouds and this may result in losing track of the vehicle location. In addition, point cloud matching generally requires high processing power, so it 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. For applications such as warehouse robots, 2D lidar SLAM is commonly used, whereas SLAM using 3-D lidar point clouds can be used for UAVs and automated driving.

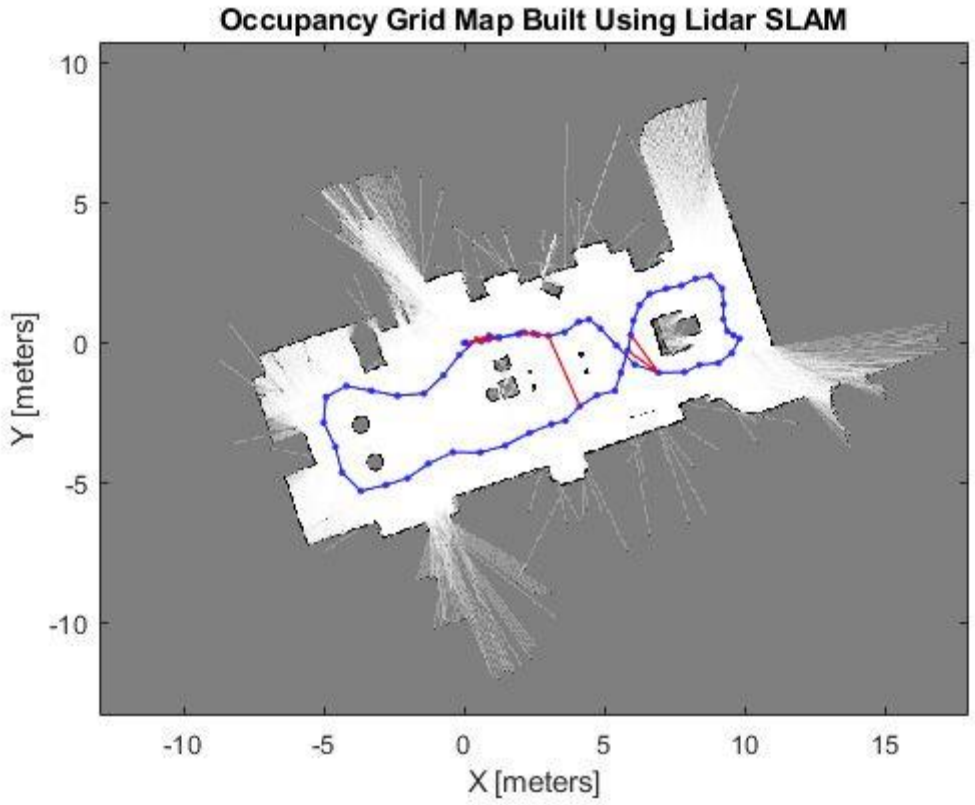


Figure 82 (2.5.4.6.5.3.2) SLAM with a 2D Lidar

2.5.4.6.5.3.3. Common Challenges with SLAM

Although SLAM is used for some practical applications, several technical challenges prevent more general-purpose adoption. Each has a countermeasure that can help overcome the obstacle.

1. Localization errors accumulate, causing substantial deviation from actual values SLAM estimates sequential movement, which include some margin of error. The error accumulates over time, causing substantial deviation from actual values. It can also cause map data to collapse or distort, making subsequent searches difficult. Let's take an example of driving around a square-shaped passage. As the error accumulates, robot's starting and ending point no longer match up. This is called a loop closure problem. Pose estimation errors like these are unavoidable. It is important to detect loop closures and determine how to correct or cancel out the accumulated error.

Figure 83 (2.5.4.6.5.3.3.a) Example of constructing a pose graph and minimizing errors.

One countermeasure is to remember some characteristics from a previously visited place as a landmark and minimize the localization error. Pose graphs are constructed to help correct the errors. By solving error minimization as an optimization problem, more accurate map data can be generated. This kind of optimization is called bundle adjustment in visual SLAM.

Figure 84 (2.5.4.6.5.3.2.b) Example of constructing a pose graph and minimizing errors.

2. Localization fails and the position on the map is lostmage and point-cloud mapping does not consider the characteristics of a robot's movement. In some cases, this approach can generate discontinuous position estimates. For example, a calculation result showing that a robot moving at 1 m/s suddenly jumped forward by 10 meters. This kind of localization failure can be prevented either by using a recovery algorithm or by fusing the motion model with multiple sensors to make calculations based on the sensor data.

There are several methods for using a motion model with sensor fusion. A common method is using Kalman filtering for localization. Since most differential drive robots and four-wheeled vehicles generally use nonlinear motion models, extended Kalman filters and particle filters (Monte Carlo localization) are often used. More flexible Bayes filters such as unscented Kalman filters can also be used in some cases. Some commonly used sensors are inertial measurement devices such as IMU, Attitude and Heading Reference System or AHRS, Inertial Navigation System or INS, accelerometer sensors, gyro sensors, and magnetic sensors). Wheel encoders attached to the vehicle are often used for odometry.

When localization fails, a countermeasure to recover is by remembering a landmark as a key-frame from a previously visited place. When searching for a landmark, a feature extraction process is applied in a way that it can scan at high speeds. Some methods based on image features include bag of features (BoF) and bag of visual words (BoVW). More recently, deep learning is used for comparison of distances from features.

3. High computational cost for image processing, point cloud processing, and optimization.

Computing cost is a problem when implementing SLAM on vehicle hardware. Computation is usually performed on compact and low-energy embedded microprocessors that have limited processing power. To achieve accurate localization, it is essential to execute image processing and point cloud matching at high frequency. In addition, optimization calculations such as loop closure are high computation processes. The challenge is how to execute such computationally expensive processing on embedded microcomputers.

One countermeasure is to run different processes in parallel. Processes such as feature extraction, which is preprocessing of the matching process, is relatively suitable for parallelization. Using multicore CPUs for processing, single instruction multiple data (SIMD) calculation, and embedded GPUs can further improve speeds in some cases. Also, since pose graph optimization can be performed over a relatively long cycle, lowering its priority and carrying out this process at regular intervals can also improve performance.

2.5.4.6.5.3.4. SLAM Using Pose Graph Optimization [48]

In Slam Algorithm, we assume that the robot does not have a previous map. There are two types of algorithms used in SLAM, one is Filtering such as using Particle filters or extended Kalman filters, in which the state is estimated on the go with the latest measurement. The other type is Smoothing in which the Full trajectories are estimated using complete set of measurements, the frequently used algorithm in modern years has been pose graph optimization.

In SLAM using Pose Graph Optimization, the robot starts recording its poses via the odometry, and starts comparing the new pose to the previous pose, with the encoder calculation, and apply an algorithm to smooth the sides getting a map, building at the end Binary Occupancy Grid.

In **(figure 85)**, in an ideal situation, where there are no uncertainity or errors in the lidar or odometry measurments, the robot will move and take the poses and all new poses will allign with each other to make the full map. But this is not the case in real applications, there will be odometry errors due to wheel slip and the lidar measurment may have some noise.

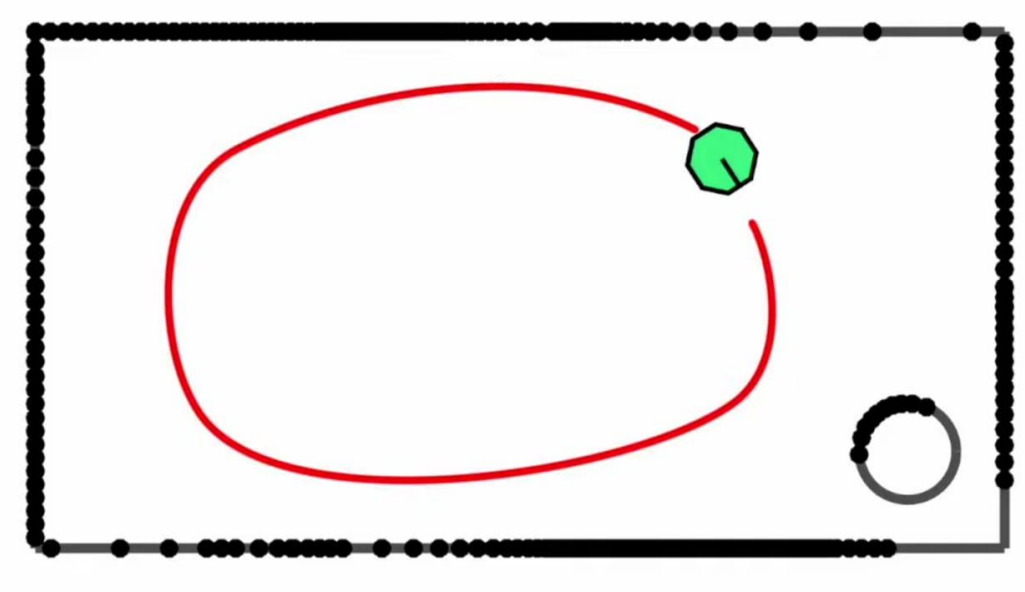


Figure 85 (2.5.4.6.5.3.4.a) SLAM Algorithm - Ideal Map

In **figure (86)**, we can see a more realistic situation of having a large uncertainty in the odometry, where we notice that the error causes our estimated pose to deviate from the real pose, building up an inaccurate map that does not resemble the real environment.

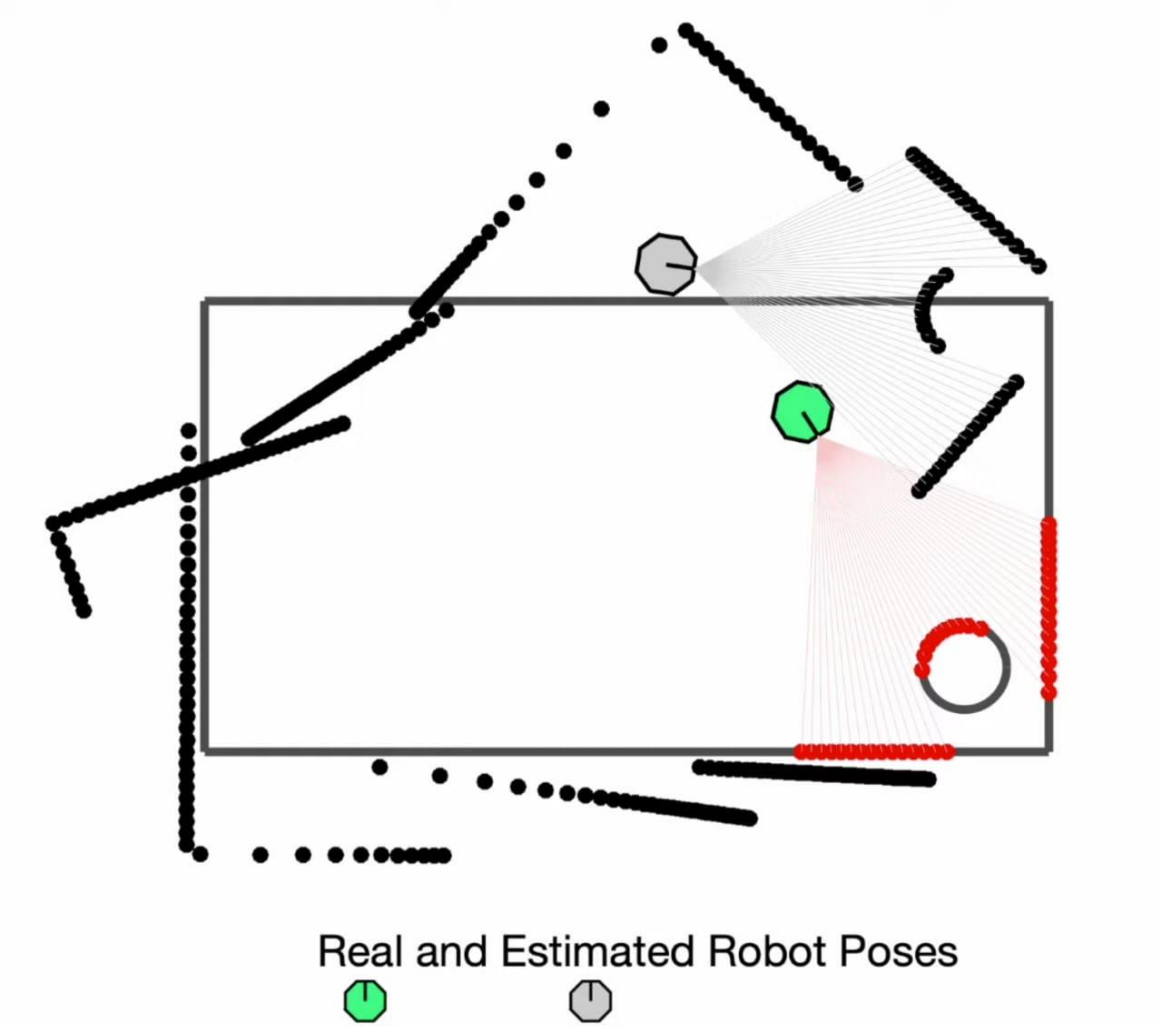


Figure 86 (2.5.4.6.5.3.4.b) SLAM Algorithm - Real and Estimated Robot Poses

In **figure (87)**, when the robot first takes a measurement of the environment, and this measurement is associated with the current pose as an x and y location and an orientation angle and saved to the pose graph with the distance and angles to the sensed obstacles. The robot starts moving and takes another pose that is deviated from the estimated pose, but which gets saved with the real pose in the pose graph, and even though we don't know where are those two poses in the environment, we do have an idea of about how far apart they are from each other. In our case the knowledge came from counting the wheel rotation, we have some best guess as to how far away these two poses are from each other as well as a measure of how confident we are in that guess in this way there's a constraint that we can place on the relative distance between these two poses ideally they would stay exactly this far apart, since that's our best estimate but due to our uncertainty in the odometry process maybe we'd be better off if we moved these two poses around relative to each other for example it would be nice if there was a mechanism that would be able to move the two poses around to align the currently misaligned obstacles well there is and the constraints are the key to doing it.

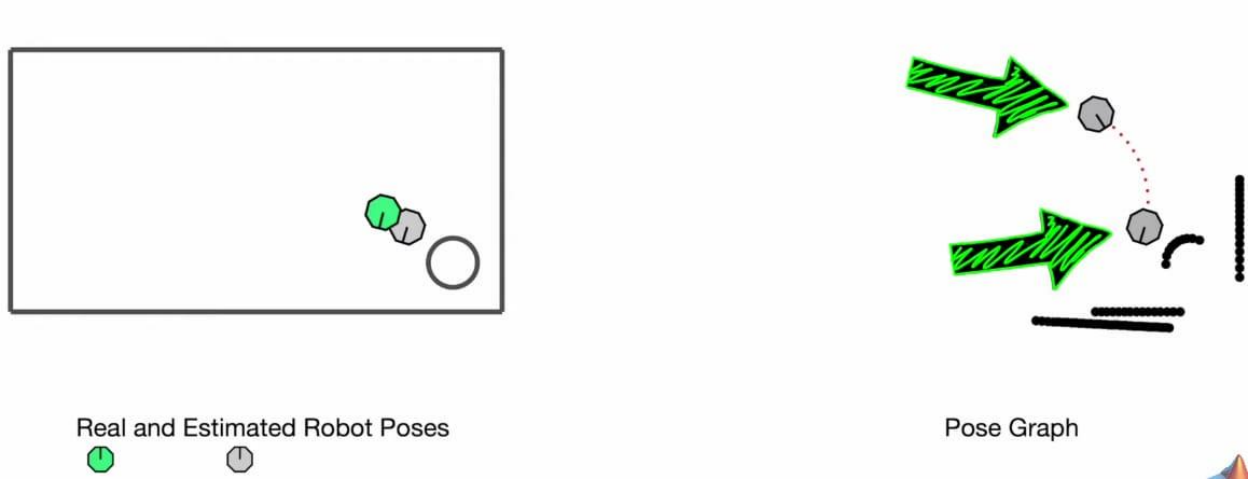


Figure 87 (2.5.4.6.5.3.4.c) SLAM Algorithm - Comparing Poses

In **figure (88)**, to visualize a constraint it's helpful to imagine a rubber bar connecting the two poses the nominal length of the bar is how far apart we estimate them to be and with no external forces on these poses the bar will just keep them at this fixed distance however if I hold one pose fixed and move the other pose closer to or further from it will compress or stretch the rubber are and there will be a force that wants to restore the poses back to their nominal distance and the strength of the rubber bar depends on how confident we are in the distance estimate if we have more confidence or we have a really good odometry process then this is a really strong bar that makes it difficult to change this nominal distance if we have almost no confidence then this rubber bar is very weak and provides almost no restoring force so in Poe's graph nomenclature we say that the poses are the nodes of the graph and the constraint or the rubber bar is an edge. of course this constraint acts in all three pose dimensions both x and y and in rotation always trying to bring it back to its estimated distance.

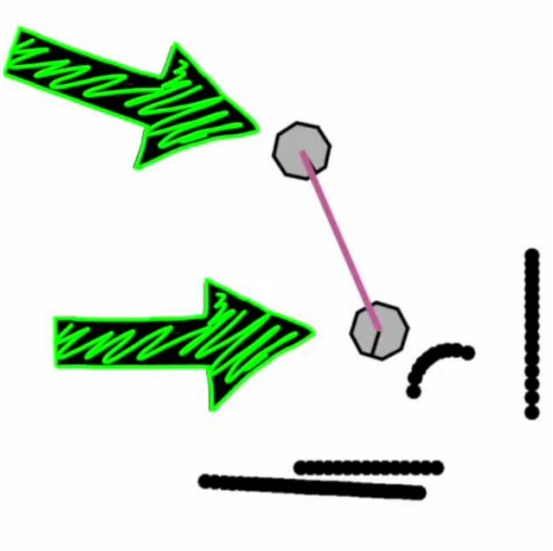


Figure 88 (2.5.4.6.5.3.4.d) SLAM Algorithm - Getting Poses to align

In **figure (89)**, if we continue this process, taking out one pose at a time to complete this graph, getting a map that is the same as the inaccurate map with the exception of having those constraints.

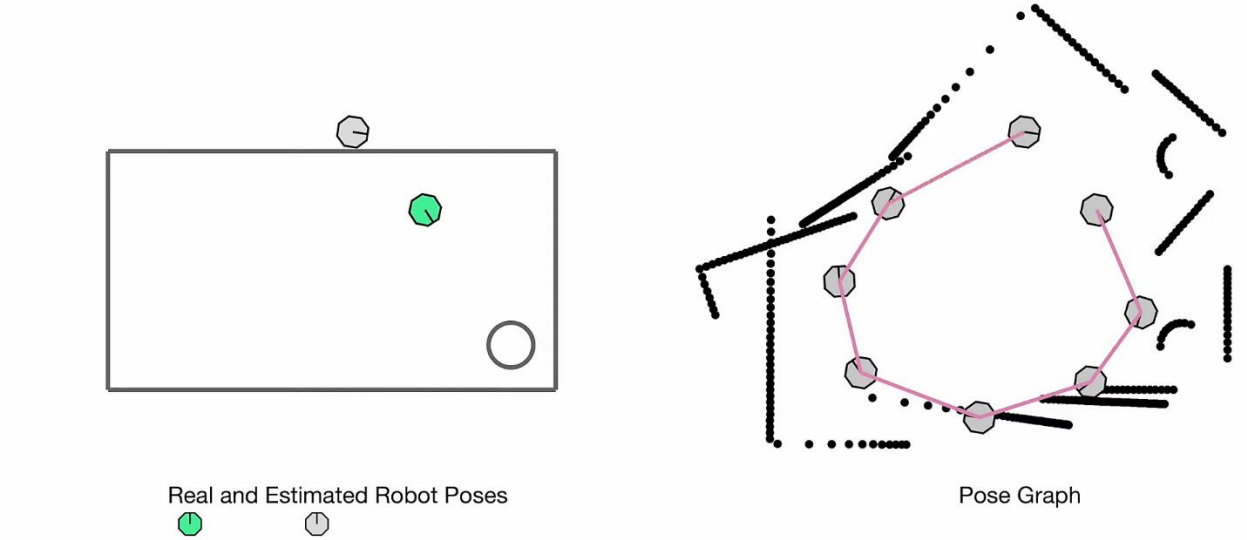


Figure 89 (2.5.4.6.5.3.4.e) SLAM Algorithm - Map with constraints

In **figure (90)**, we can see that our first pose and the current pose are both observing the same feature in the environment this means that we can build a new edge a new constraint between these two nodes we just need to understand how these two features align to figure out where the two poses have to be relative to each other and in this example they would have to be in the exact same location so now we can add a new constraint one that connects the first and last poses together and closes the loop this bar has a nominal length of zero since these two poses want to be right on top of each other and the strength of this rubber bar depends on how confident you are in your external measurements and your ability to associate the two sets of data as the same feature if you're really confident like we are in this example this bar is extremely strong it really wants to pull these two poses together.

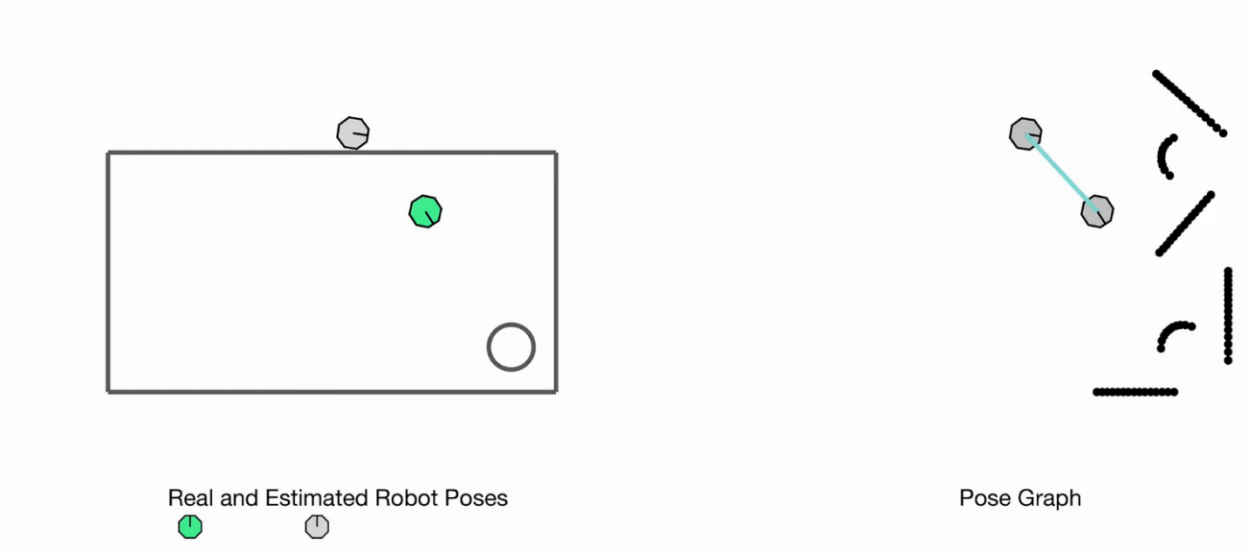


Figure 90 (2.5.4.6.5.3.4.f) SLAM Algorithm - Using constraints.

In **figure (91)**, a loop closure is the thing that makes all of this work without a closure all of the constraints were just happily being met but with a loop closure we have a way to inject tension into this graph the blue bar wants to pull those two notes together and the purple bars all want to keep the relative distances the way that they are.

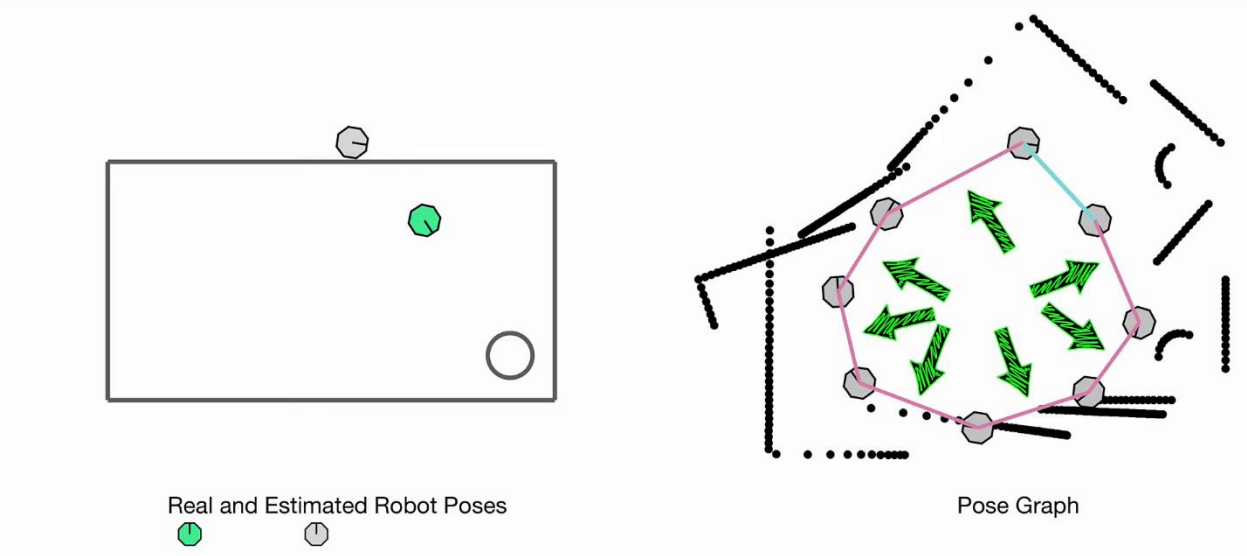


Figure 91 (2.5.4.6.5.3.4.g) SLAM Algorithm - Aligning constraints.

In **figure (92)**, allowing this graph to settle to equilibrium or to balance all of the forces that each of the constraints are imposing is the optimization part of pose graph optimization and by optimizing the pose graph not only do we have a better estimate of the current pose and a better model of the environment but we also have a better estimate of where the robot was in the past since all of the past poses were updated as well so we got a lot of value from this one loop closure of course you can see that we don't have a perfect model of the environment or the robot poses.

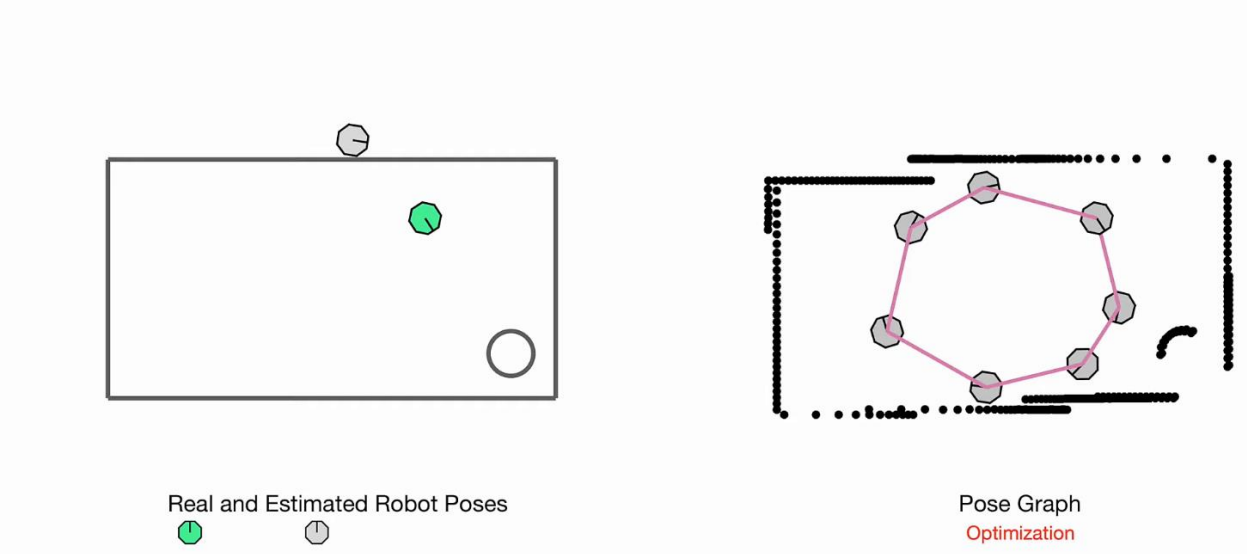


Figure 92 (2.5.4.6.5.3.4.h) SLAM Algorithm - Pose Graph optimization.

In **figure (93)**, adding more loop closures adds up with new constraints and rerunning the process, the graph will continue to improve its global model of the environment and the past robot poses that is as long as the two poses that you connect are truly looking at the same environment feature if you make an incorrect Association and you build an edge with a constraint that is not real you can imagine how this would persist in your graph and continue to skew your optimization process every time you run, it would always be pulling those nodes in a direction away from the truth. So it's incredibly important that you are confident in any loop closures that you make in fact it's better to miss a real loop closure than it is to add a false one you can always drive around some more and make another closure in the future but it's harder to remove the bad ones once they're in . It's also important to note that a loop closure requires an absolute measurement of the environment like we got with lidar or you could get with a visible camera or some other environment sensor you can't close the loop using a relative sensor like an IMU or the wheel encoders because there's no way to relate that information back to an older pose you may think you're near a past node but without checking the environment you can't know for sure.

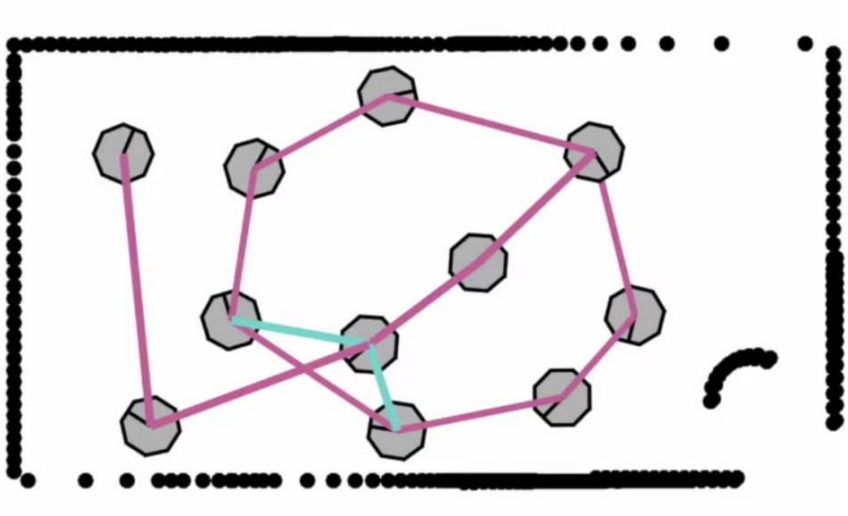


Figure 93 (2.5.4.6.5.3.4.i) SLAM Algorithm - Multiple poses processing

In **figure (94)**, to make a useful map we can introduce the binary occupancy grid the idea behind it is that the environment is broken up into a grid of cells if you believe the cell is occupied you set the value to a1 and if it's not occupied you set it to a zero, assuming the entire environment is empty and then filling in the cells where I believe there is an obstacle but you could also assume the opposite where the cells are all occupied and you set them to zero when your sensor indicates that there's nothing there you can see that with just ones and zeroes we've generated a pretty accurate model of the rectangular room and part of the circular obstacle there are some holes and we can fill them in by driving around a bit more but overall it looks okay.

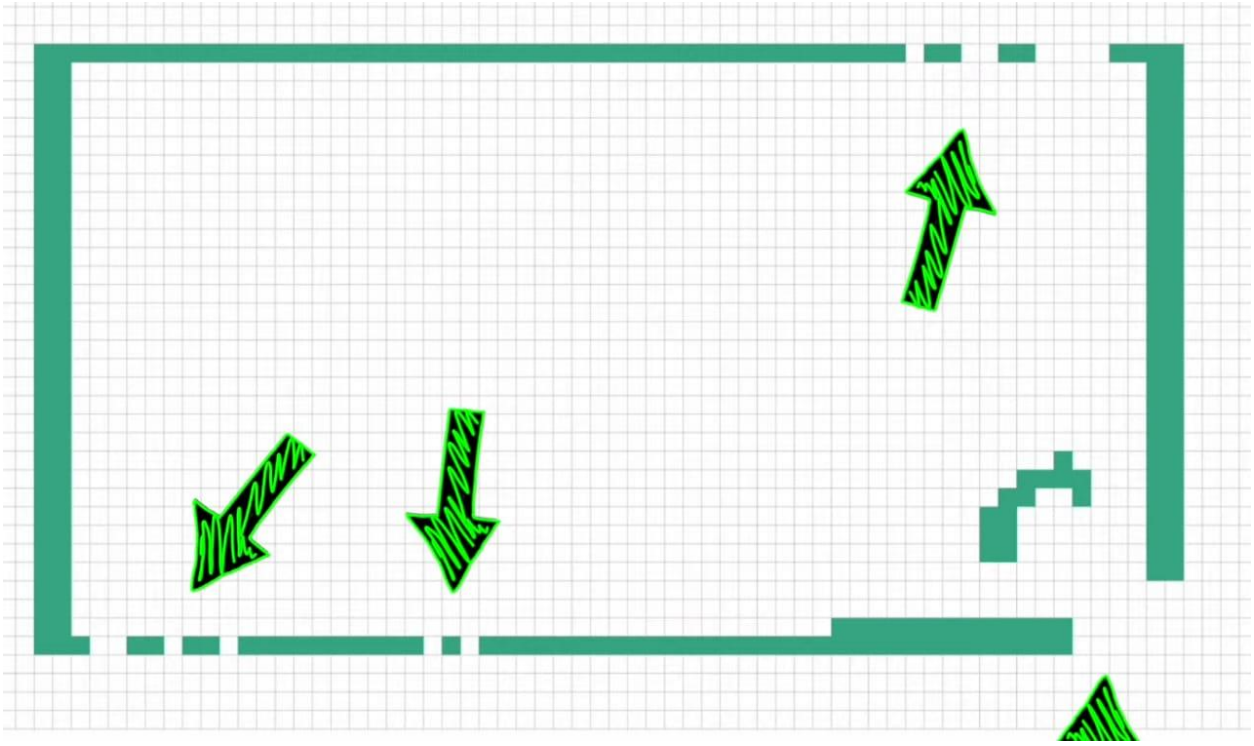
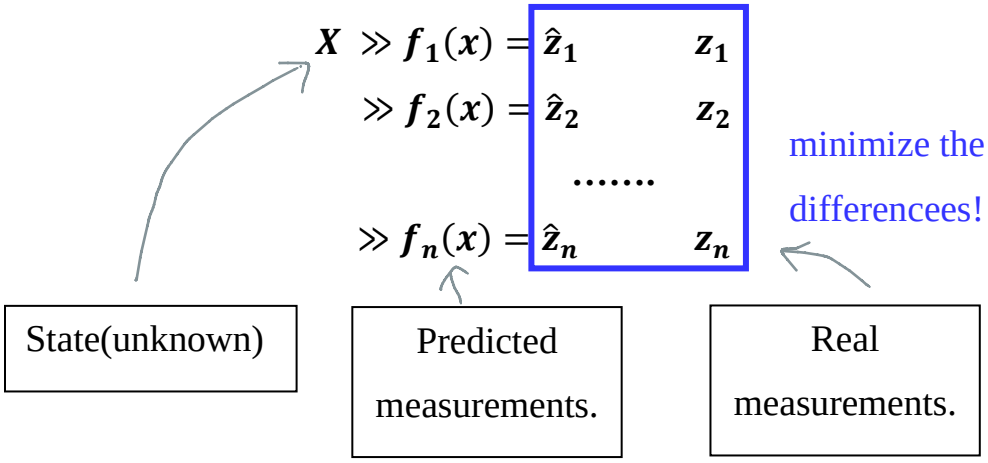


Figure 94 (2.5.4.6.5.3.4.j) SLAM Algorithm - Building Occupancy grid Map

2.5.4.6.5.3.5. SLAM Mathematical Model [49]

Graphical Explanation:



Error Function:

Error e_i is typically the difference between the predicted and actual measurement.

$e_i(x) = Z_i - f_i(x)$

We assume that the measurement error has zero mean and is normally distributed.

Gaussian error with information matrix Ω_i

The squared error of a measurement depends only on the state and is a scalar.

$e_i(x) = e_i(x)^T \Omega_i e_i(x)$

Goal finds the minimum:

Find the state X^* which minimizes the error given all measurements.

$$X^* = \underset{x}{argmin}\, F(x)$$

Global error (scaler)

$$= \underset{x}{argmin} \sum_i e_i(x)$$

Squared error terms (scaler)

$$= \underset{x}{argmin} \sum_i e_i(x)\ \Omega_i\ e_i(x)$$

Error terms (vector)

Goal: Find the Minimum:

Find the state x* which minimizes the error given all measurements.

$$X^* = \underset{x}{argmin} \sum_i e_i(x)\ \Omega_i\ e_i(x)$$

A general solution is to derive the global error function and find its nulls.

In general complex and no closed form solution (Numerical approach)

Assumption:

A good initial guess is available. The error functions are "smooth" in the neighborhood of the (hopefully global) minima. Then, we can solve the problem by iterative local linearization.

Solve Via Iterative Linearization

Linearize the error terms around the current solution/initial guess.

Compute the first derivative of the squared error function.

Set it to zero and solve linear system.

Obtain the new state (that is hopefully closer to the minimum) Iterate.

Linearizing the Error Function

Approximate the error functions around an initial guess x via Taylor expansion:

$$e_i(x + \Delta x) \simeq \underbrace{e_i(x)}_{e_i} + J_i(x)\Delta x$$

Reminder: Jacobian

$$J_f(x) = \begin{pmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_1(x)}{\partial x_2} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_2} & \cdots & \frac{\partial f_2(x)}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_m(x)}{\partial x_1} & \frac{\partial f_m(x)}{\partial x_2} & \cdots & \frac{\partial f_m(x)}{\partial x_n} \end{pmatrix}$$

Squared Error

With this linearization, we can fix X and carry out the minimization in the increments Δx

We replace the Taylor expansion in the squared error terms:

$$e_i(x + \Delta x) = \cdots$$

We replace the Taylor expansion in the squared error terms:

$$\begin{aligned} e_i(x + \Delta x) &= e_i^T(x + \Delta x)\Omega_i e_i(x + \Delta x) \\ &\simeq e_i(x + J_i\Delta x)^T\Omega_i(e_i + J_i\Delta x) \end{aligned}$$

We replace the Taylor expansion in the squared error terms:

$$\begin{aligned} e_i(x + \Delta x) &= e_i^T(x + \Delta x)\Omega_i e_i(x + \Delta x) \\ &\simeq e_i(x + J_i\Delta x)^T\Omega_i(e_i + J_i\Delta x) \end{aligned}$$

$$= e_i^T \Omega_i e_i + e_i^T \Omega_i J_i \Delta x + \Delta x^T J_i^T \Omega_i e_i + \Delta x^T J_i^T \Omega_i J_i \Delta x$$

By grouping similar terms, we obtain:

$$e_i(x + \Delta x) \simeq \underbrace{e_i^T \Omega_i e_i}_{c_i} + 2\underbrace{e_i^T \Omega_i J_i}_{b_i^T} \Delta x + \Delta x^T \underbrace{J_i^T \Omega_i J_i}_{H_i} \Delta x$$

$$= c_i + 2b_i^T \Delta x + \Delta x^T H_i \Delta x$$

Global Error:

The global error is the sum of the squared errors terms corresponding to the individual measurements.

$$F(x + \Delta x) \simeq \sum_i (c_i + b_i^T \Delta x + \Delta x^T H_i \Delta x)$$

$$= \underbrace{\sum_i c_i}_{c} + 2\underbrace{\left(\sum_i b_i^T\right)}_{b^T} \Delta x + \Delta x^T \underbrace{(\sum_i H_i)}_{H} \Delta x$$

$$= c + 2b^T \Delta x + \Delta x^T H \Delta x$$

Where b^T is: $b^T = \sum_i e_i^T \Omega_i J_i$

Where H is: $H = \sum_i J_i^T \Omega J_i$

Quadratic Form:

We can write the global error terms as a quadratic form in Δ x

$F(x + \Delta x) \simeq c + 2b^T \Delta x + \Delta x^T H \Delta x$

Our goal is to minimize this function.

We need to compute the derivative of

$F(x + \Delta x)$ W.R.T. Δx

Deriving a Quadratic Form:

Given a quadratic form:

$f(x) = x^T H x + b^T x$

its first derivative is:

$\frac{\partial f}{\partial x} = (H + H^T)x + b$

We can write the linearized global error terms as a quadratic form in Δ x

$F(x + \Delta x) \simeq c + 2b^T \Delta x + \Delta x^T H \Delta x$

The derivative of F(x+Δx) W.R.T. Δx is then:

$\frac{\partial F(x + \Delta x)}{\partial \Delta x} \simeq 2b + 2H\Delta x$

Minimizing the Quadratic Form

Derivative:

$\frac{\partial F(x + \Delta x)}{\partial \Delta x} \simeq 2b + 2H\Delta x$

Setting it to zero leads to:

$0 = 2b + 2H\Delta x$

Which leads to the linear system:

$H\Delta x = -b$

The solution for the increment Δx^* is:

$\Delta x^* = -H^{-1}b$

Overall Procedure:

Linearize around x and compute for each measurement:

$e_i(x + \Delta x) \simeq e_i(x) + J_i \Delta x$

Compute the terms for the linear system:

$$b^T = \sum_i e_i^T \Omega_i J_i$$

$$H = \sum_i J_i^T \Omega_i J_i$$

Solve the linear system:

$\Delta x^* = -H^{-1}b$

Updating state:

$x \leftarrow x + \Delta x^*$

2.5.4.6.5.3.6. SLAM Toolbox [50]

In ROS (Robot Operating System), the SLAM_toolbox is a package that provides a set of tools and nodes for performing Simultaneous Localization and Mapping (SLAM) tasks. It offers a modular and flexible framework for mapping and localization in ROS-based robotic systems. The SLAM_toolbox package includes several key components and functionalities:

1) Mapping Nodes: The SLAM_toolbox package provides mapping nodes that can generate maps of the environment using sensor data. These nodes utilize SLAM algorithms such as occupancy grid mapping, grid-based mapping, or point cloud mapping to construct maps based on data from sensors like laser scanners or depth sensors.

2) Localization Nodes: The package also includes localization nodes that estimate the robot's position and orientation within a known map. These nodes utilize algorithms like Monte Carlo Localization (MCL), Extended Kalman Filter (EKF), or particle filters to determine the robot's pose based on sensor data and the map.

3) Sensor Data Fusion: The SLAM_toolbox incorporates sensor data fusion techniques to improve the accuracy and robustness of SLAM. It can fuse data from multiple sensors, such as laser scanners, cameras, or inertial measurement units (IMUs), to enhance the quality of the generated maps and improve the localization accuracy.

4) ROS Integration: The SLAM_toolbox is designed to work seamlessly with the ROS ecosystem. It leverages ROS message types and topics for data

exchange, allowing integration with other ROS components, visualization tools (such as RViz), and analysis tools.

5) Configuration and Parameter Tuning: The package provides a flexible configuration interface, allowing users to adjust various parameters and settings based on their specific robot and environment. This enables fine-tuning of the SLAM algorithms and adaptation to different scenarios.

6) Map Management: The SLAM_toolbox includes functionalities for map management, such as saving and loading maps, combining maps from multiple sessions, and updating maps as the robot explores and revisits areas of the environment.

7) Quality Metrics and Evaluation: The package offers tools for evaluating the quality and performance of SLAM algorithms and generated maps. It provides metrics to assess accuracy, precision, and consistency of the estimated poses and the resulting maps.

The SLAM_toolbox package provides a comprehensive and modular solution for SLAM in ROS, catering to a wide range of robotic platforms and sensor configurations. It simplifies the development and integration of SLAM functionality into ROS-based robotic systems, enabling accurate mapping and localization capabilities. The package can be utilized in various applications, such as autonomous navigation, robot exploration, or mapping for robotic perception tasks.

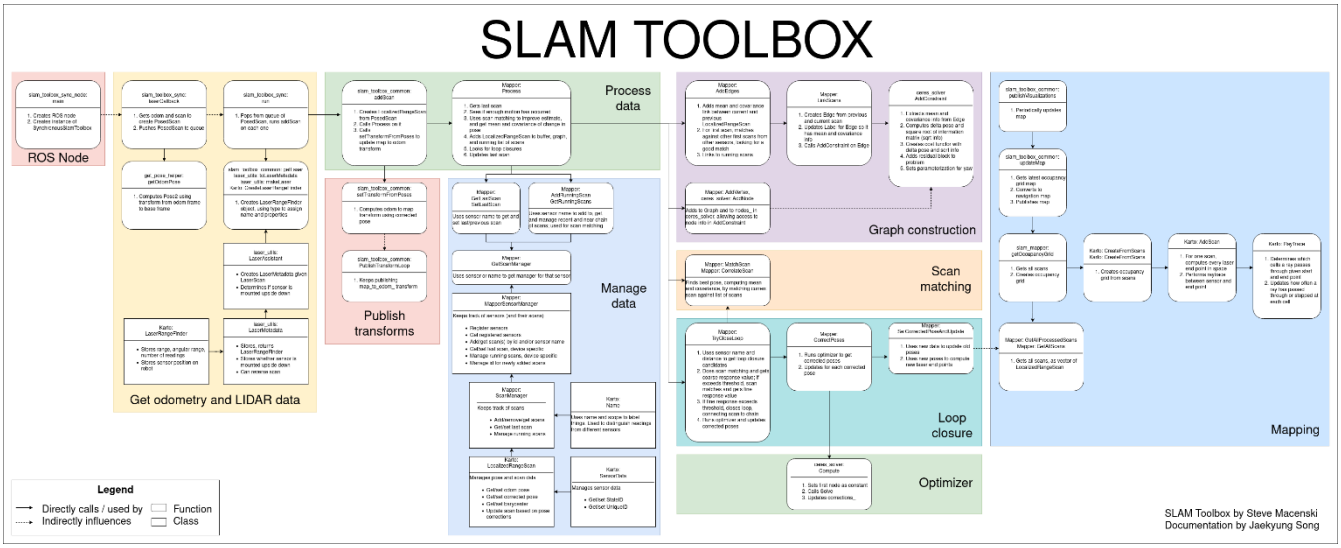


Figure 95 (2.5.4.6.5.3.6) SLAM_toolbox Structure

2.5.4.6.6. Path Planning [57]

The path planning problem of mobile robots is a hot spot in the field of mobile robot navigation research: mobile robots can find an optimal or near-optimal path from the starting state to the target state that avoids obstacles based on one or some performance indicators (such as the lowest working cost, the shortest walking route, the shortest walking time, etc.) in the motion space.

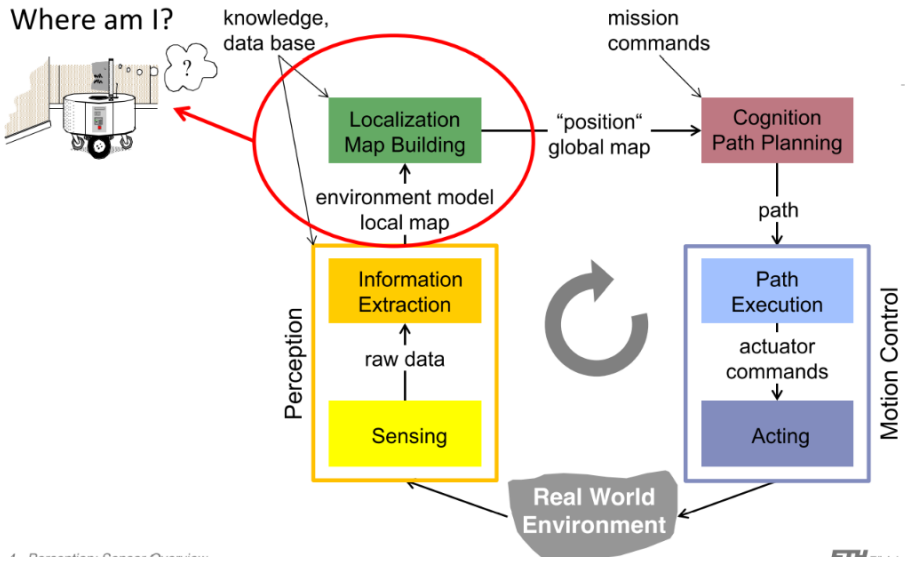


Figure 96 (2.5.4.6.6.a) Path Planning Process

The path planning selects the shortest distance of the path: that is, the shortest length of the path from the starting point to the target point as a performance indicator. The mobile robot path planning method can be divided into two types according to the known degree of environmental information: path planning based on global map information or local map information. These two path planning methods are referred to as global path planning and local path planning. The global path planning method can generate the path under the completely known environment (the position and shape of the obstacle are predetermined). With the global map model of the environment where the mobile

robots are located, the search is performed on the established global map model. The optimal algorithm can obtain the optimal path. Therefore, global path planning involves two parts: establishment of the environmental model and the path planning strategy.

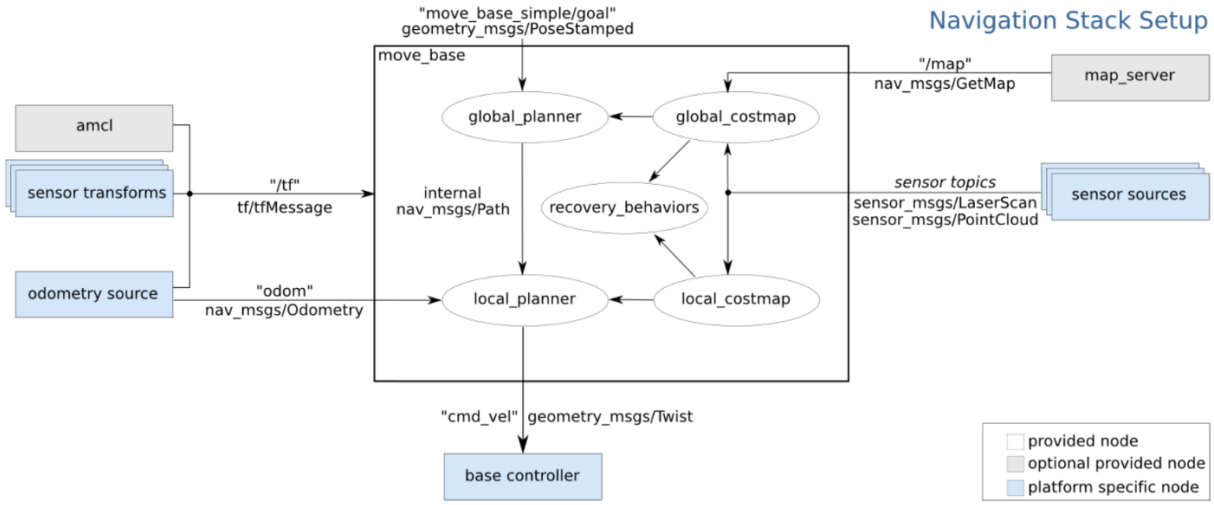


Figure 97 (2.5.4.6.6.b) ROS Navigation Stack setup

The Navigation Stack is fairly simple on a conceptual level. It takes in information from odometry and sensor streams and outputs velocity commands to send to a mobile base. Use of the Navigation Stack on an arbitrary robot, however, is a bit more complicated. As a pre-requisite for navigation stack use, the robot must be running ROS, have a TF transform tree in place, and publish sensor data using the correct ROS Message types. Also, the Navigation Stack needs to be configured for the shape and dynamics of a robot to perform at a high level. To help with this process, this manual is meant to serve as a guide to typical Navigation Stack setup and configuration.

2.5.4.6.6.1 Dijkstra's Algorithm [58]

Dijkstra's algorithm is a popular graph search algorithm used to find the shortest path between two nodes in a weighted graph. It was developed by Dutch computer scientist Edsger W. Dijkstra in 1956. The algorithm works for both directed and undirected graphs with non-negative edge weights. Here's an overview of how Dijkstra's algorithm works:

1) Initialization:

- Assign a tentative distance value to every node in the graph, usually set to infinity except for the starting node, which is set to 0.
- Set the starting node as the current node.

2) Iteration:

- For the current node, consider all of its neighboring nodes.
- Calculate the tentative distance from the starting node to each neighboring node through the current node.
- If the calculated distance is smaller than the previously assigned tentative distance, update the tentative distance.

3) Selection of the next node:

- After considering all the neighbors of the current node, mark the current node as visited.
- From the unvisited nodes, select the one with the smallest tentative distance as the next current node and repeat the iteration step.
- If the destination node is marked as visited or if the smallest tentative distance among the unvisited nodes is infinity, the algorithm terminates.

4) Path reconstruction:

- Once the destination node is marked as visited, the algorithm can terminate.
- To find the shortest path, backtrack from the destination node to the starting node, following the path with the lowest cumulative weight.

Dijkstra's algorithm guarantees finding the shortest path from the starting node to all other nodes in the graph. It is an efficient algorithm for small to medium-sized graphs, but its time complexity can be high for large graphs. Various optimizations, such as using a priority queue to efficiently select the node with the smallest tentative distance, can be applied to improve its performance.

It's important to note that Dijkstra's algorithm assumes non-negative edge weights. If negative weights are present, other algorithms like Bellman-Ford or the A* algorithm should be considered.

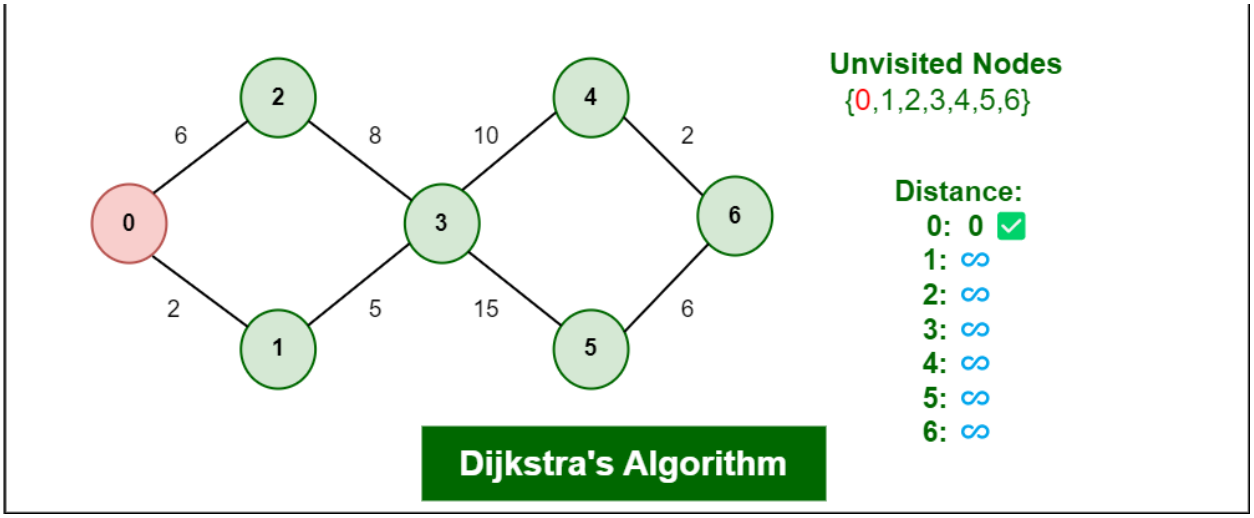


Figure 98 (2.5.4.6.6.1.a) Dijkstra's Algorithm

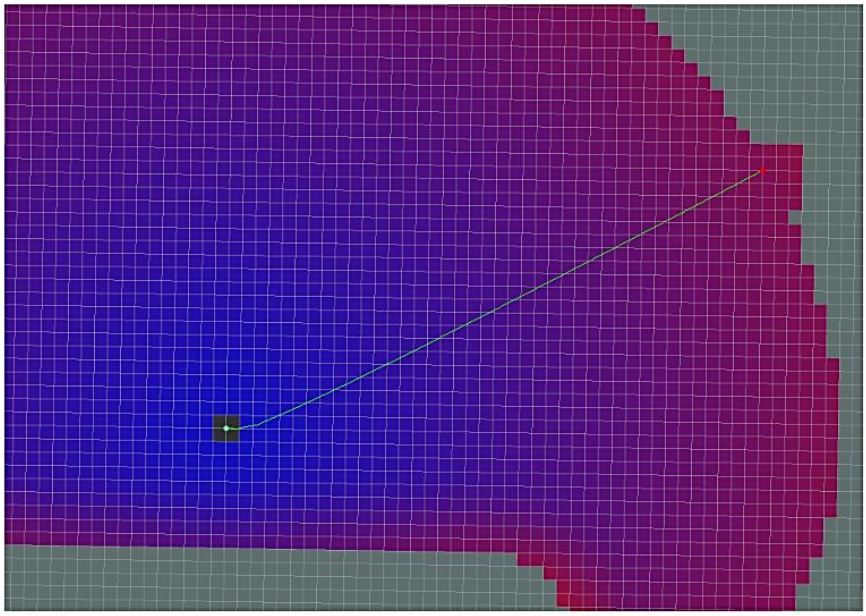


Figure 99 (2.5.4.6.6.1.b) Dijkstra's Algorithm Global Path

2.5.4.6.6.2 Global Planner [59]

The global planner requires a map of the environment to calculate the best route. Depending on the analysis of the map, some methods are based on Roadmaps like Silhouette proposed by Canny in 1987 or Voronoi in 2007. Some of them solve the problem by assigning a value to each region of the roadmap in order to find the path with minimum cost. Some examples are the Dijkstra algorithm, Best First, and. Another approach is by dividing the map into small regions (cells) called cell decomposition as. A similar approach by using potential fields, the most extended algorithm used few years ago rapidly exploring random trees or the new approach based on neural networks. Some solutions combine the algorithms improving the outcome at the cost of high.

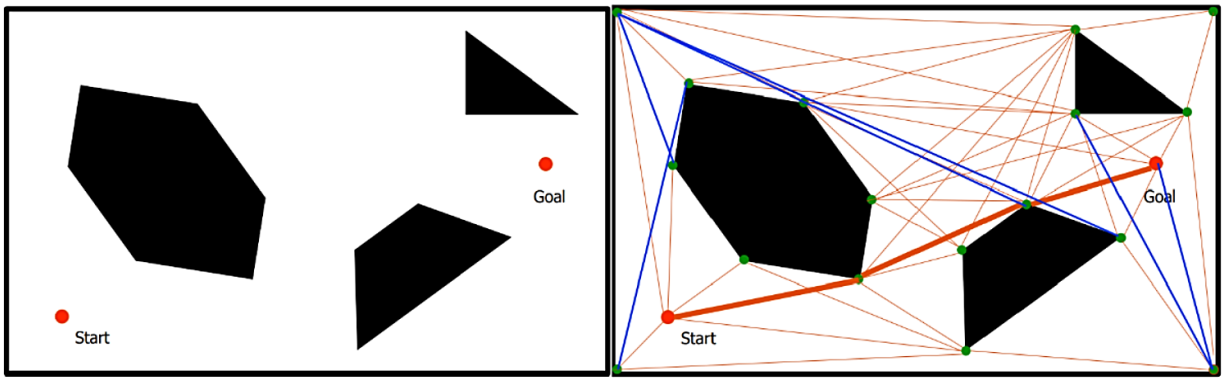


Figure 100 (2.5.4.6.6.2) Graph Search Representation

2.5.4.6.6.2.1 A* Graph Search Algorithm [59]

A* Search attempts to improve upon the performance of grassfire and Dijkstra by incorporating a heuristic function that guides the path planner Heuristic function.

H(n) takes a node n and returns a non-negative real number that is an estimation of the path cost from node n to the goal.

```
For each node n in the graph
n.f = Infinity, n.g = Infinity
Create an empty list
start.g = 0 , start.f = H(start) add start to list
While list not empty
Let current = node in the list with the smallest f value, remove current from list
If (current == goal node) report success
For each node, n that is adjacent to current
If (n.g > (current.g + cost of edge from n to current))
n.g = current.g + cost of edge from n to current
n.f = n.g + H(n)
n.parent = current add n to list if it isn't there already
```

Figure 101 (2.5.4.6.6.2.1.a) A Graph Search Algorithm*

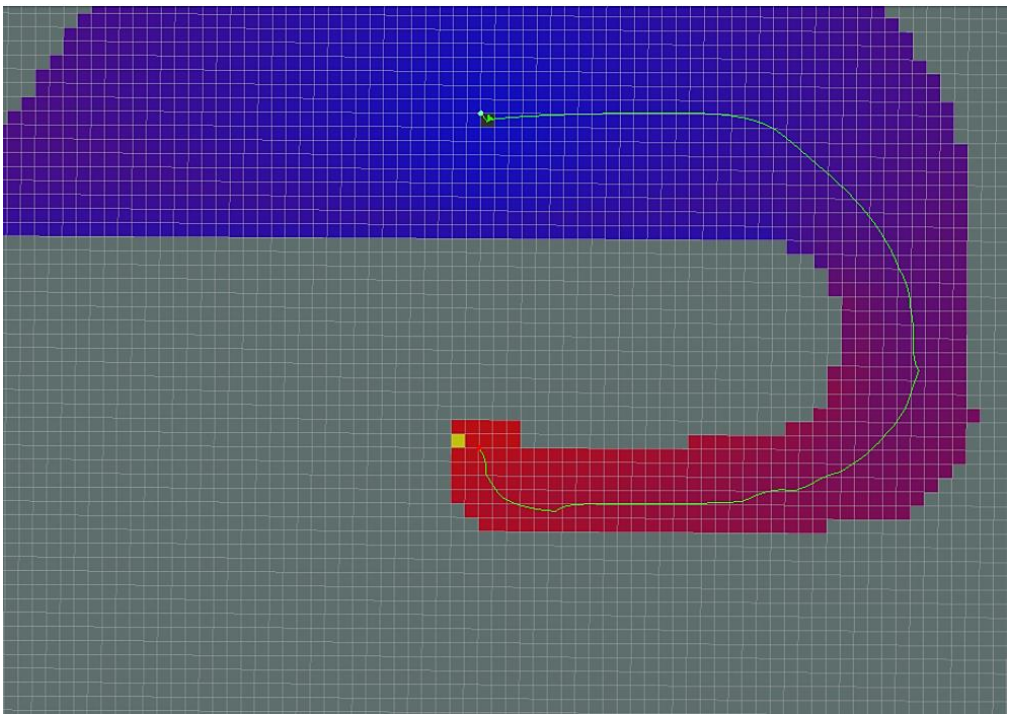


*Figure 102 (2.5.4.6.6.2.1.b) Global Path Planning Using A**

2.5.4.6.6.3 Local Planner [59]

In order to transform the global path into suitable waypoints, the local planner creates new waypoints taking into consideration the dynamic obstacles and the vehicle constraints. So, to recalculate the path at a specific rate, the map is reduced to the surroundings of the vehicle and is updated as the vehicle is moving around. It is not possible to use the whole map because the sensors are unable to update the map in all regions and a large number of cells would raise the computational cost. Therefore, with the updated local map and the global waypoints, the local planning generates avoidance strategies for dynamic obstacles and tries to match the trajectory as much as possible to the provided waypoints from the global planner. Different approaches are based on trajectory generation using Clothoids lines, Bezier lines, arcs and segments, or splines. All of them create intermediate waypoints following the generated trajectory.

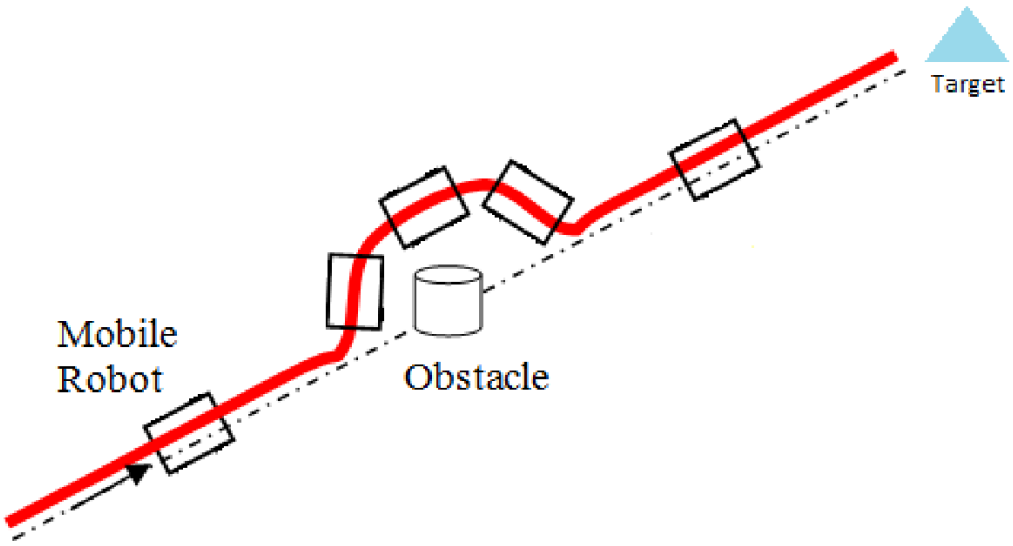
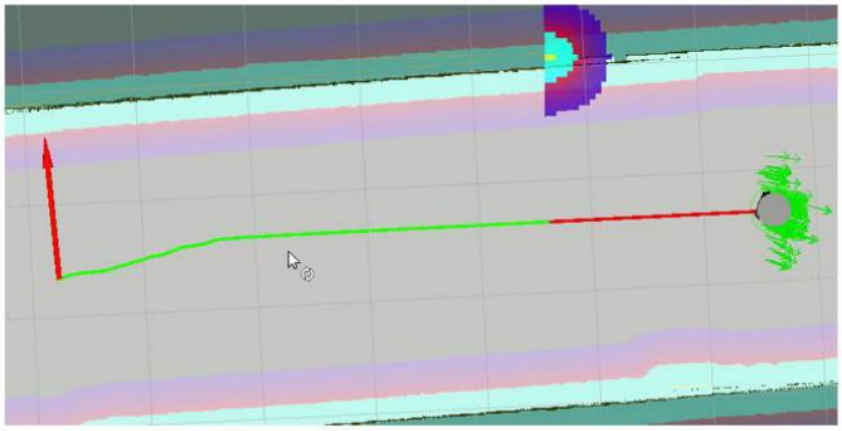


Figure 103 (2.5.4.6.6.3.a) Local Planning a path to avoid the obstacle.



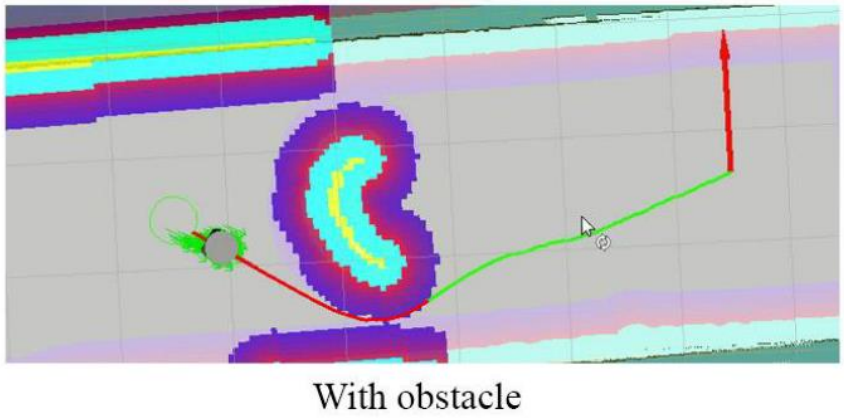


Figure 104 (2.5.4.6.6.3.b) ROS Local Planning Using Cost Map

2.5.4.6.6.4 Recovery Behavior [59]

It is a package that provides a recovery behavior for the navigation stack that attempts to clear space by performing a 360-degree rotation of the robot. It is activated when surrounded by obstacles that prevent the robot to track the global planar path or the given goal is invalid due to robot geometry.

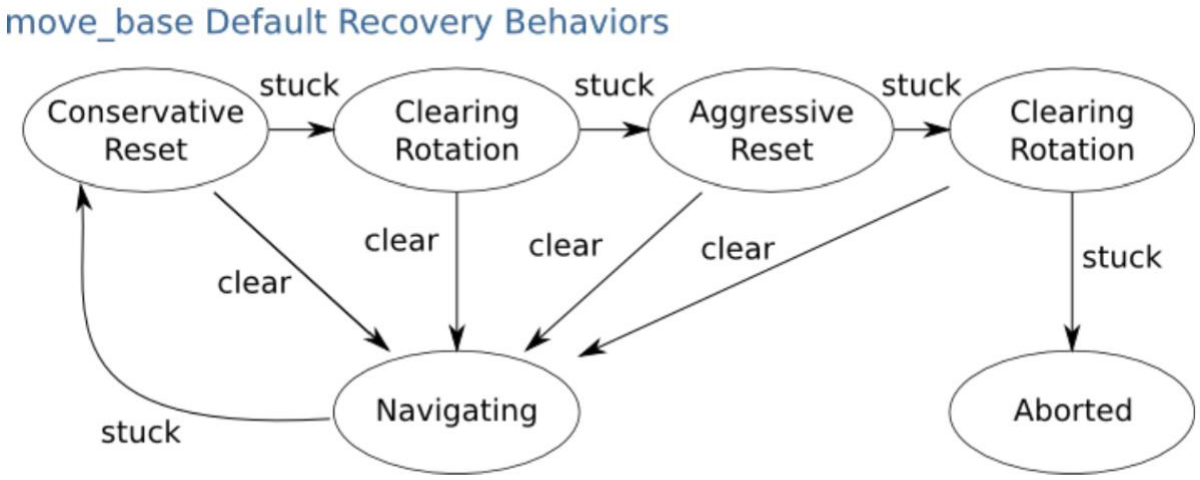


Figure 105 (2.5.4.6.6.4) Recovery Behavior Flow Chart

2.5.4.6.6.4. Navigation2 Stack [60]

Nav2 is the professionally supported spiritual successor of the ROS Navigation Stack. This project seeks to find a safe way to have a mobile robot move to complete complex tasks through many types of environments and classes of robot kinematics. 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 and more. Nav2 is a production-grade and high-quality navigation framework trusted by 50+ companies worldwide.

It provides perception, planning, control, localization, visualization, and much more to build highly reliable autonomous systems. This will complete

environmental modeling from sensor data, dynamic path planning, compute velocities for motors, avoid obstacles, represent semantic regions and objects, and structure higher-level robot behaviors. To learn more about this project, such as related projects, robots using, ROS1 comparison, and maintainers, see About and Contact. To learn more about navigation and ROS concepts, see Navigation Concepts.

Nav2 uses behavior trees to create customized and intelligent navigation behavior via orchestrating many independent modular servers. A task server can be used to compute a path, control effort, recovery, or any other navigation related task. These separate servers communicate with the behavior tree (BT) over a ROS interface such as an action server or service. A robot may utilize potentially many different behavior trees to allow a robot to perform many types of unique tasks.

The diagram below **(figure 106)** will give you a good first-look at the structure of Nav2. Note: It is possible to have multiple plugins for controllers, planners, and recoveries in each of their servers with matching BT plugins. This can be used to create contextual navigation behaviors. If you would like to see a comparison between this project and ROS (1) Navigation, see ROS to ROS 2 Navigation.

The expected inputs to Nav2 are TF transformations conforming to REP-105, a map source if utilizing the Static Costmap Layer, a BT XML file, and any relevant sensor data sources. It will then provide valid velocity commands for the motors of a holonomic or non-holonomic robot to follow. We currently support all of the major robot types: holonomic, differential-drive, legged, and ackermann

(car-like) base types! We support them uniquely with both circular and arbitrarily-shaped robots for SE2 collision checking.

It has tools to:

- Load, serve, and store maps (Map Server)
- Localize the robot on the map (AMCL)
- Plan a path from A to B around obstacles (Nav2 Planner)
- Control the robot as it follows the path (Nav2 Controller)
- Smooth path plans to be more continuous and feasible (Nav2 Smoother)
- Convert sensor data into a costmap representation of the world (Nav2 Costmap 2D)
- Build complicated robot behaviors using behavior trees (Nav2 Behavior Trees and BT Navigator)
- Compute recovery behaviors in case of failure (Nav2 Recoveries)
- Follow sequential waypoints (Nav2 Waypoint Follower)
- Manage the lifecycle and watchdog for the servers (Nav2 Lifecycle Manager)
- Plugins to enable your own custom algorithms and behaviors (Nav2 Core)
- Monitor raw sensor data for imminent collision or dangerous situation (Collision Monitor)
- Python3 API to interact with Nav2 in a pythonic manner (Simple Commander)
- A smoother on output velocities to guarantee dynamic feasibility of commands (Velocity Smoother)

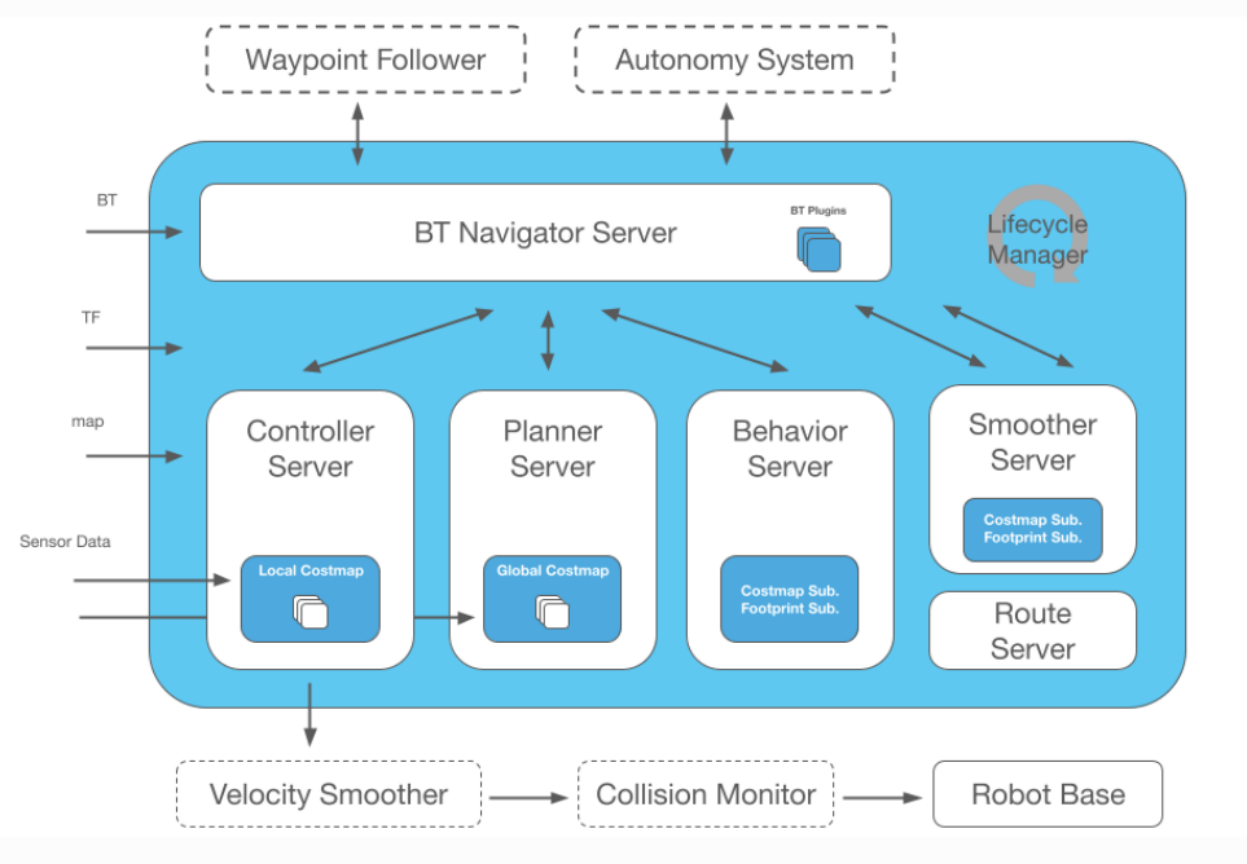


Figure 106 (2.5.4.6.6.4.5) Navigation2 Stack Flowchart

The nav2 library in ROS2 (Robot Operating System 2) is a powerful navigation stack that provides a set of components and tools for autonomous robot navigation. It is designed to assist in the development of navigation capabilities in ROS2-based robots. The nav2 library builds upon the navigation2 package and introduces several enhancements and improvements. Here are the key roles and functionalities of the nav2 library:

1) Path Planning: The nav2 library offers path planning algorithms that enable robots to generate optimal paths from their current location to a target location. It supports various algorithms such as Dijkstra's algorithm, A*, and

potential fields, which can be used to plan paths based on different criteria like distance, time, or avoiding obstacles.

2) Local Planning: It includes local planning modules that help the robot navigate safely in its immediate environment. These modules utilize sensor data, such as lidar or depth cameras, to generate collision-free trajectories. The local planner ensures that the robot smoothly follows the global path while avoiding obstacles and staying within its dynamic limits.

3) Costmap Generation: nav2 provides tools for generating costmaps, which are grid-based representations of the environment where each cell's value indicates the traversal cost. Costmaps are crucial for path planning and obstacle avoidance. The library supports both static and dynamic costmaps, allowing the robot to react to changes in the environment.

4) Sensor Integration: nav2 facilitates integration with various sensors commonly used in autonomous navigation, such as lidar, cameras, and range finders. It provides sensor drivers, data processing pipelines, and filters to process and fuse sensor data effectively.

5) Behavior Tree-Based Architecture: The nav2 library adopts a behavior tree-based architecture for defining and managing robot behaviors. Behavior trees allow the robot's navigation behavior to be organized into modular and reusable components, making it easier to specify complex navigation strategies.

6) Configuration and Parameter Management: nav2 provides tools and utilities to manage and configure navigation-related parameters. It supports loading configuration files, dynamic reconfiguration of parameters, and parameter management through the ROS2 parameter server.

7) Plug-and-Play Components: nav2 is designed with modularity in mind. It offers a set of plug-and-play components that can be customized and combined to fit specific robot navigation requirements. This modular approach enables developers to select and integrate the components that best suit their robot's capabilities and environment.

2.6. Design alternatives and justification

2.6.1. DC Motor choice

We were limited to the availability in the Egyptian market with the DC motors with encoders, it is a very limited range in terms of choice of the torque and speed. We got a 200 RPM, 4.5 kg.cm (stall torque) motor, that has a rated torque of 1.43 kg.cm. If we had the choice, we would trade a built in gearbox with lower speed and a higher torque such as 50 RPM and 6 kg.cm torque which is available but without a built-in encoder. Having more torque would have helped in overcoming more of the slip improving odometry.

Figure 107 (2.6.1) DC Motor

2.6.2. Battery

2.6.2.1. Sealed Lead Acid (SLA)

SLA batteries have lead-based electrodes, and electrolytes composed of sulfuric acid. Each electrode inside the battery contributes about 2V. SLAs are usually available in up to 12V. They are the cheapest type of battery; however, they are the heaviest ones, therefore it is usually better to replace them with NiCd, NiMH, or lithium batteries. Another disadvantage is that most of them take several hours to charge [66].

Figure 108 (2.6.2.1) Sealed Lead Acid Battery

2.6.2.2. Nickel-Cadmium (NiCd)

NiCd batteries use nickel as a cathode, and cadmium as an anode. They supply high currents without significant voltage drops. They are more expensive than SLAs, however, they can last several years if properly handled, returning their investment. Each cell provides about 1.2V. The cells are usually soldered in series to form battery packs, with voltages that are a multiple of 1.2V. The packs used in

combat usually have 12V, 18V, 24V and 36V, with respectively 10, 15, 20 and 30 cells [66].

Figure 109 (2.6.2.2) NiCd Battery

2.6.2.3. Nickel-Metal Hydride (NiMH)

NiMH batteries also use nickel as a cathode; however, the anode is composed of a metallic alloy capable of absorbing hydrates, replacing cadmium, which is poisonous. NiMH batteries store 30% more energy per weight than NiCd, however, they can consistently supply about half the peak currents of a NiCd with the same capacity. A significant problem is that these batteries lose naturally about 30% of their charge every month (self-discharge), therefore they are not appropriate for applications with sporadic use [66].

Figure 110 (2.6.2.3) NiMH Battery

2.6.3.4. Lithium-Ion

A lithium-ion battery is a family of rechargeable battery types in which lithium ions move from the negative electrode to the positive electrode during discharge and back when charging. Chemistry, performance, cost, and safety characteristics vary across lithium-ion battery types. Unlike lithium primary batteries (which are disposable), lithium-ion batteries use an intercalated lithium compound as the electrode material instead of metallic lithium [68].

Figure 111 (2.6.2.4) Lithium Ion Battery

2.6.2.5. Lithium-polymer

A lithium-polymer battery (LiPo) is a rechargeable battery that uses solid polymer for electrolytes and lithium for one of the electrodes. Commercially available LiPo are hybrids: gel polymer or liquid electrolyte in a pouch format more accurately termed a lithium-ion polymer battery. True LiPo batteries have not reached commercial viability. The batteries referred to as LiPo in commercial use offer reduced thickness, flexibility, and weight. Their qualities make LiPo batteries suited to thin smartphones, tablets, and wearables. While LiPo made a splash in radio-controlled hobbies and remains an option, they still require external casing to prevent expansion that would otherwise become a performance and safety issue [67].

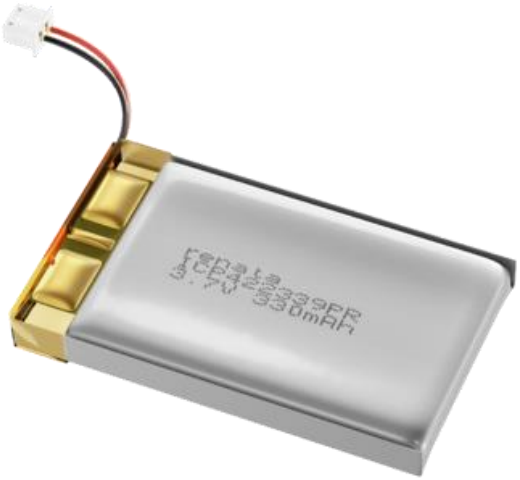


Figure 112 (2.6.2.5) Lithium Polymer Battery

2.7. Block diagram and functions of the subsystems

2.7.1. PID Controller for Robot Movement

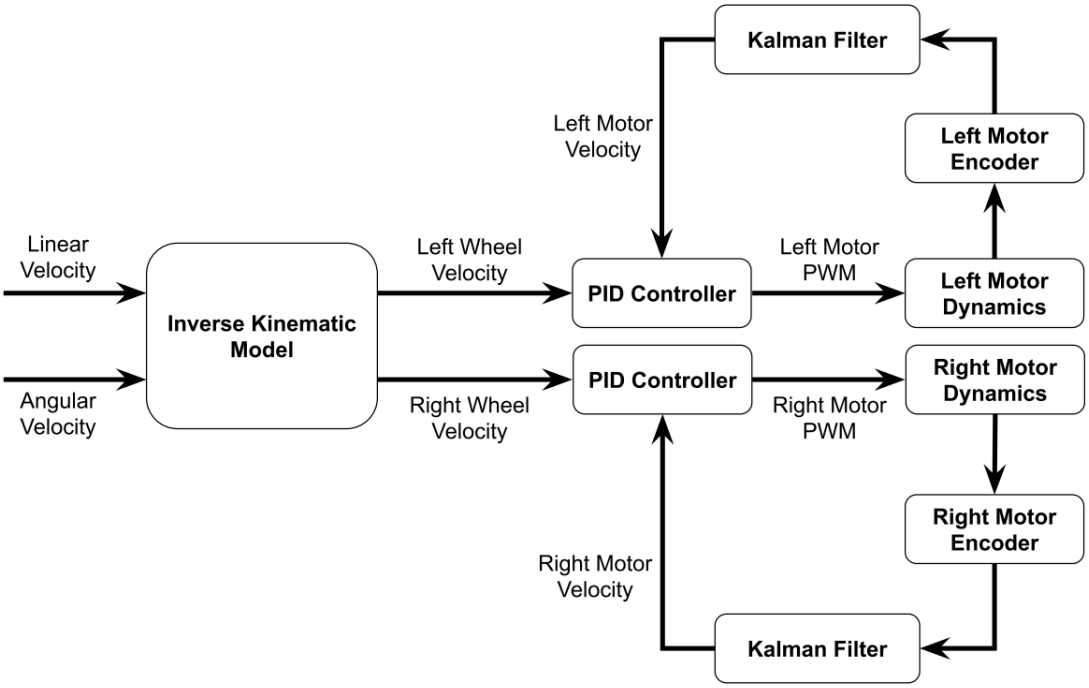


Figure 113 (2.7.1.a) Diagram of the controller used for closed-loop velocity control

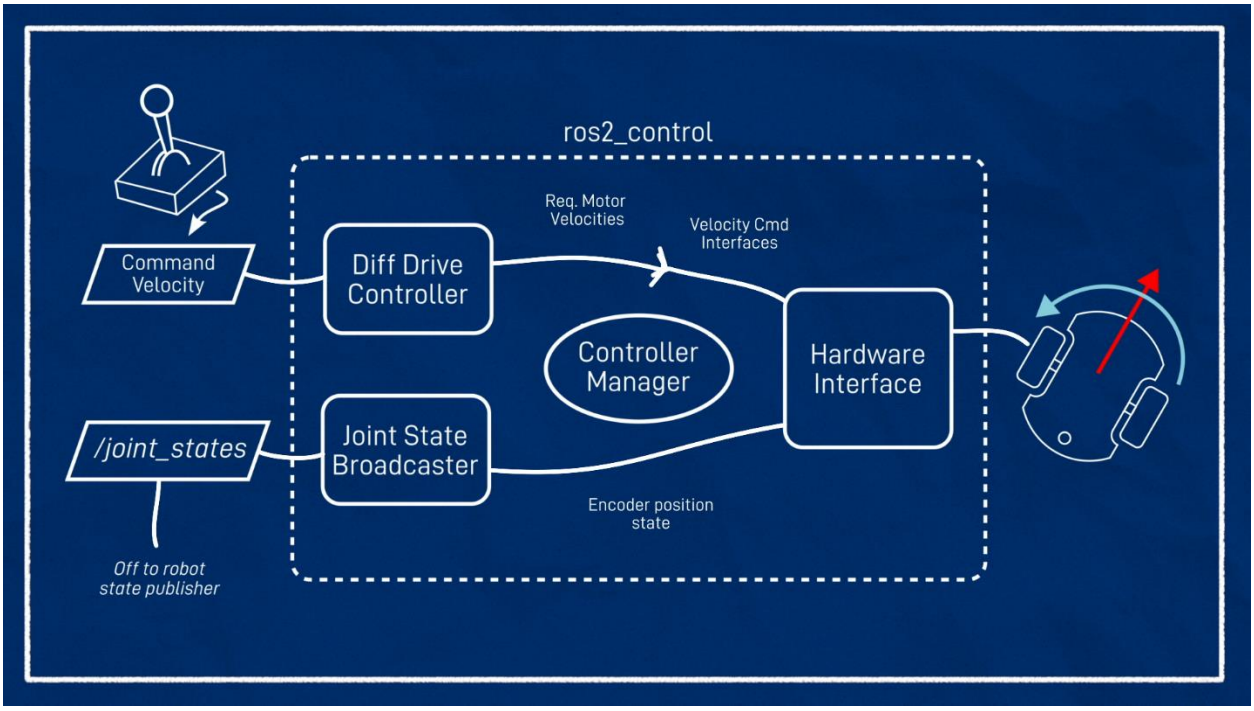


Figure 114 (2.7.1.b) ros2_control block diagram

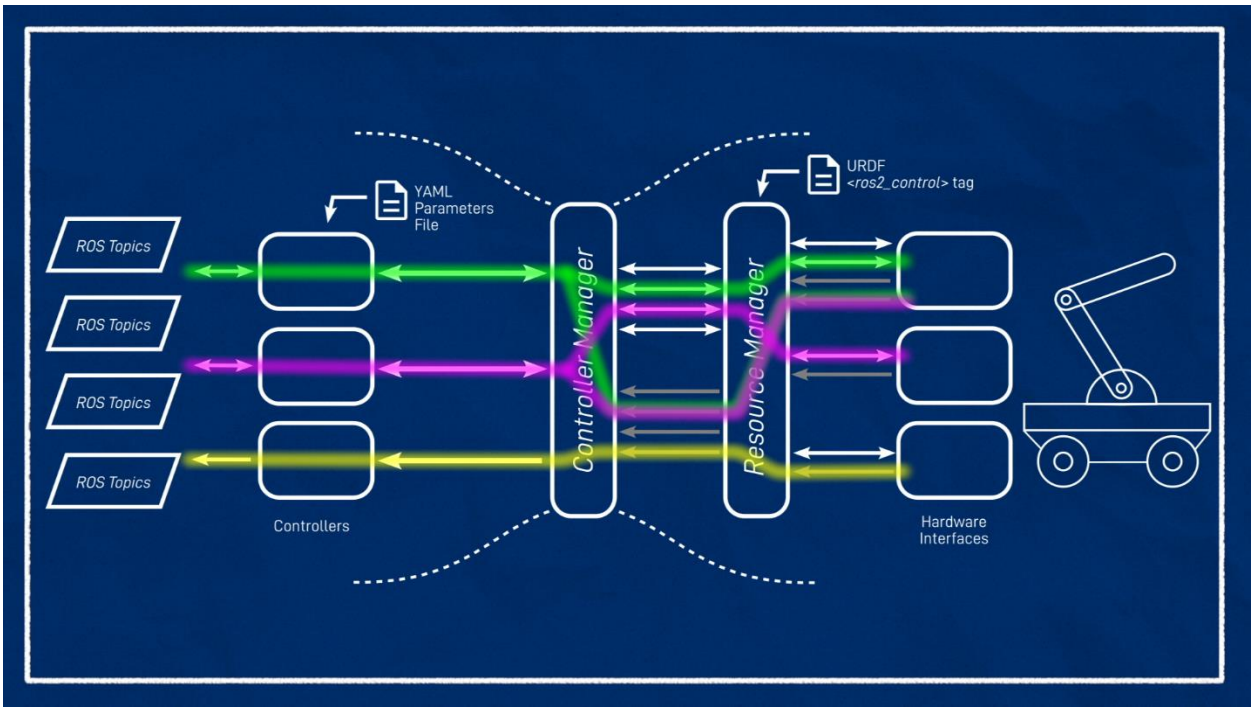


Figure 115 (2.7.1.c) Controller manager block diagram

2.7.2. SLAM Flow chart

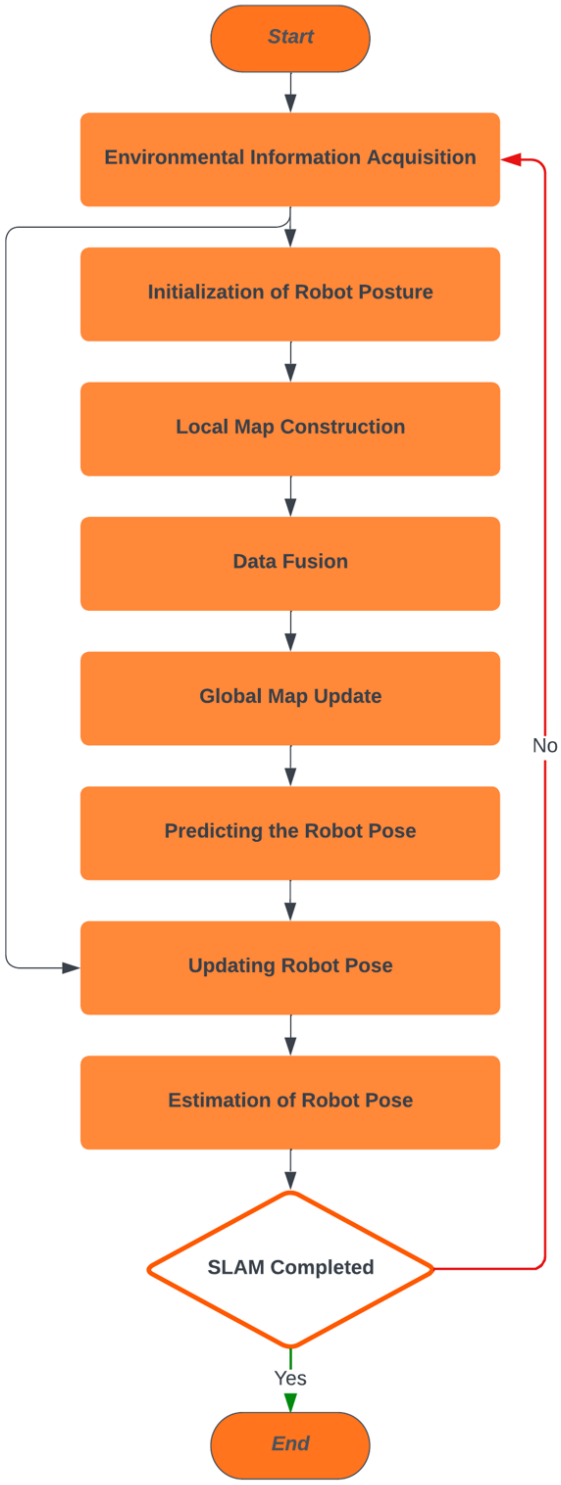


Figure 116 (2.7.2) SLAM Flowchart

2.7.3. Path Planning

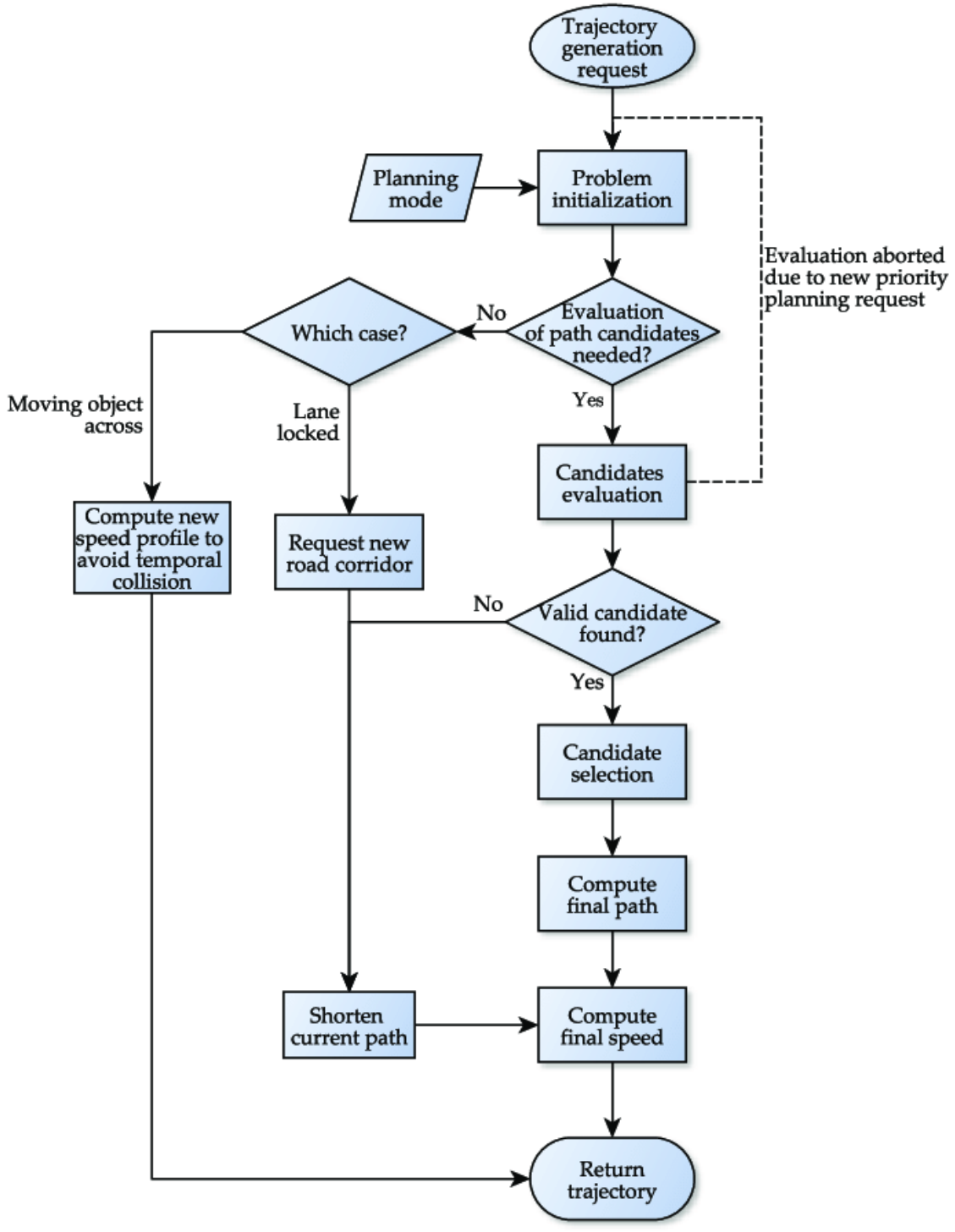


Figure 117 (2.7.3.a) Path Planning Flow Chart

```
For each node n in the graph
n.f = Infinity, n.g = Infinity
Create an empty list
start.g = 0 , start.f = H(start) add start to list
While list not empty
Let current = node in the list with the smallest f value, remove current from list
If (current == goal node) report success
For each node, n that is adjacent to current
If (n.g > (current.g + cost of edge from n to current))
n.g = current.g + cost of edge from n to current
n.f = n.g + H(n)
n.parent = current add n to list if it isn't there already
```

Figure 118 (2.7.3.b) A Search Algorithm*

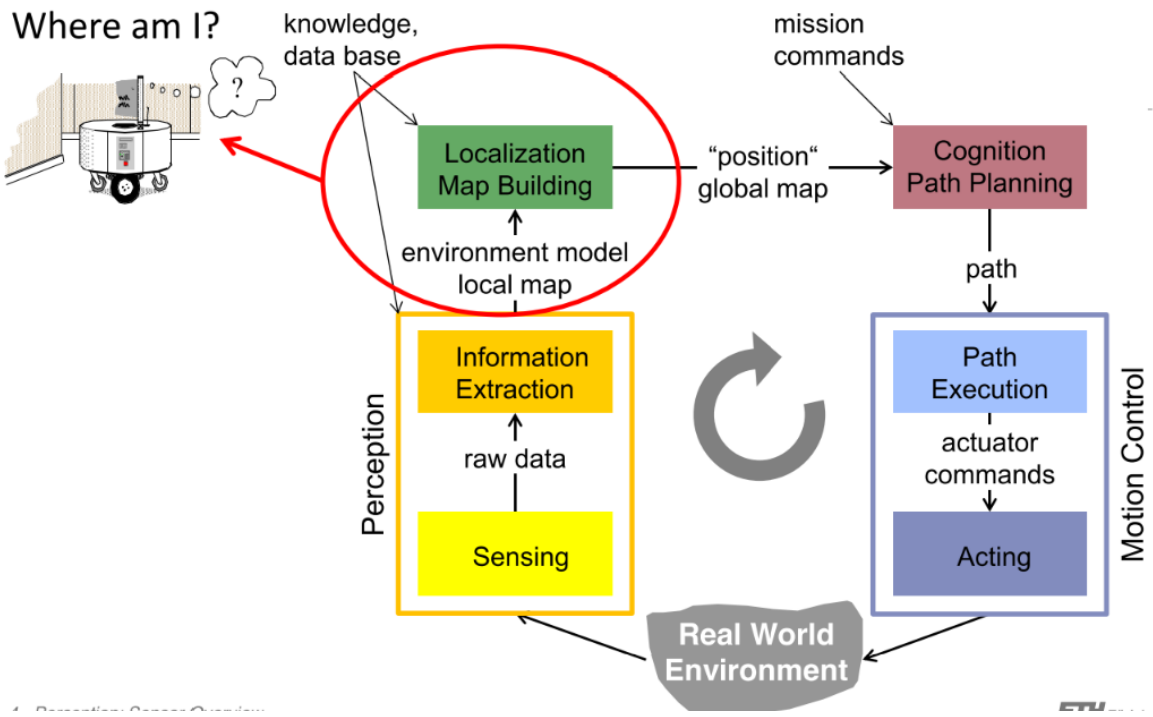


Figure 119 (2.7.3.c) Path Planning Process

2.7.4. Navigation Stack

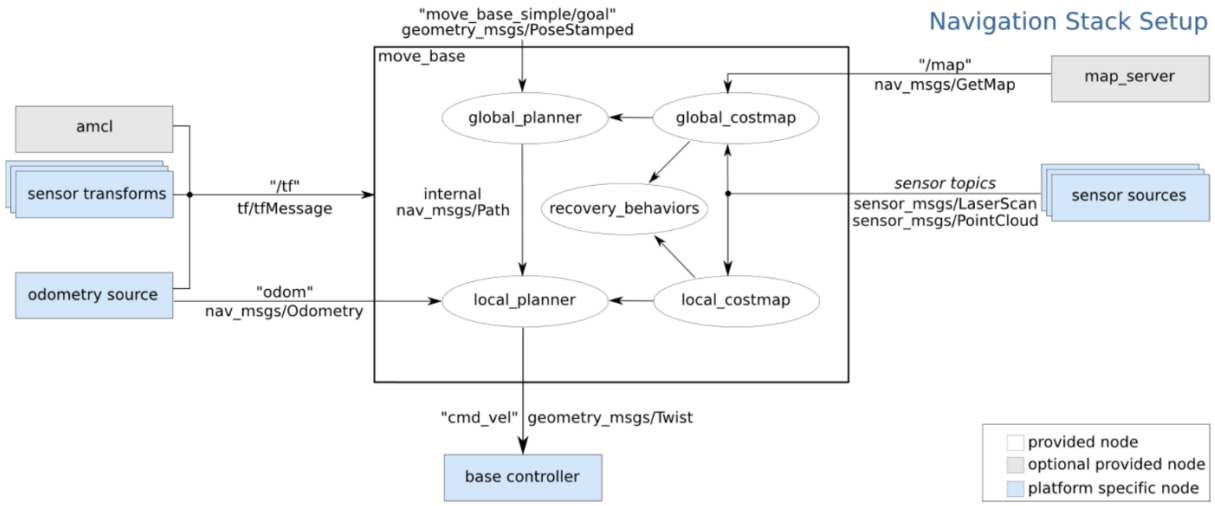


Figure 120 (2.7.4) ROS Navigation Stack

2.7.5. Electric Circuit

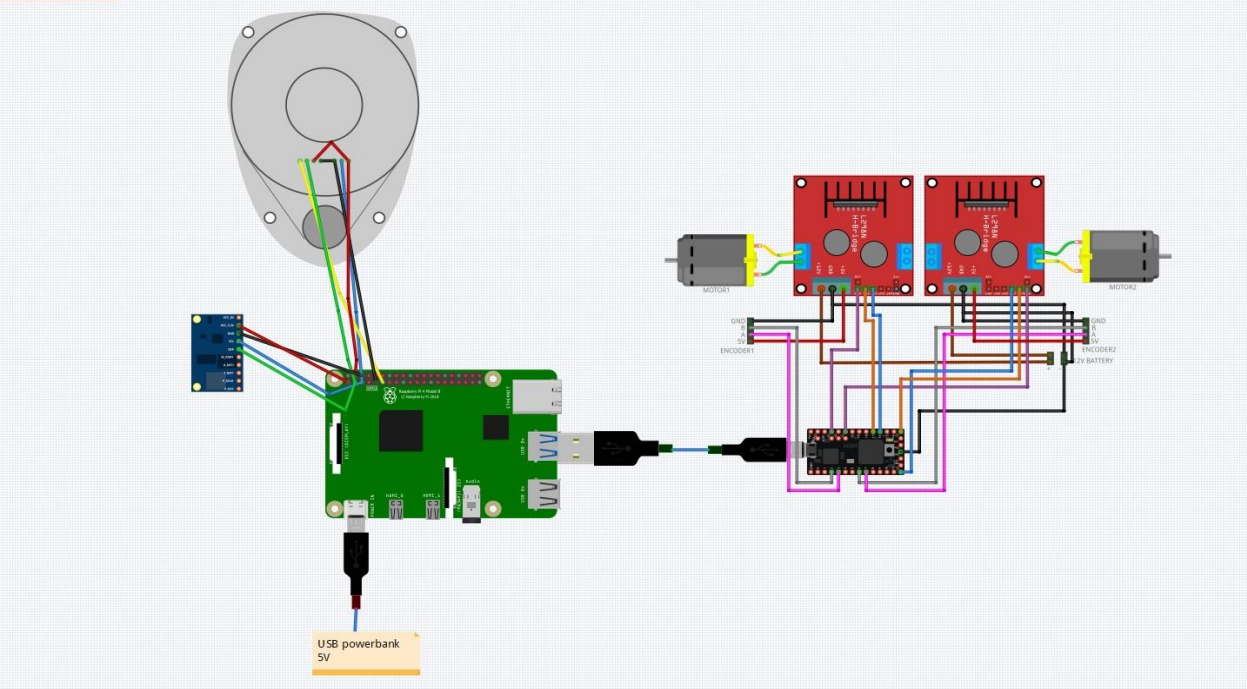


Figure 121 (2.7.5.a) Control Circuit

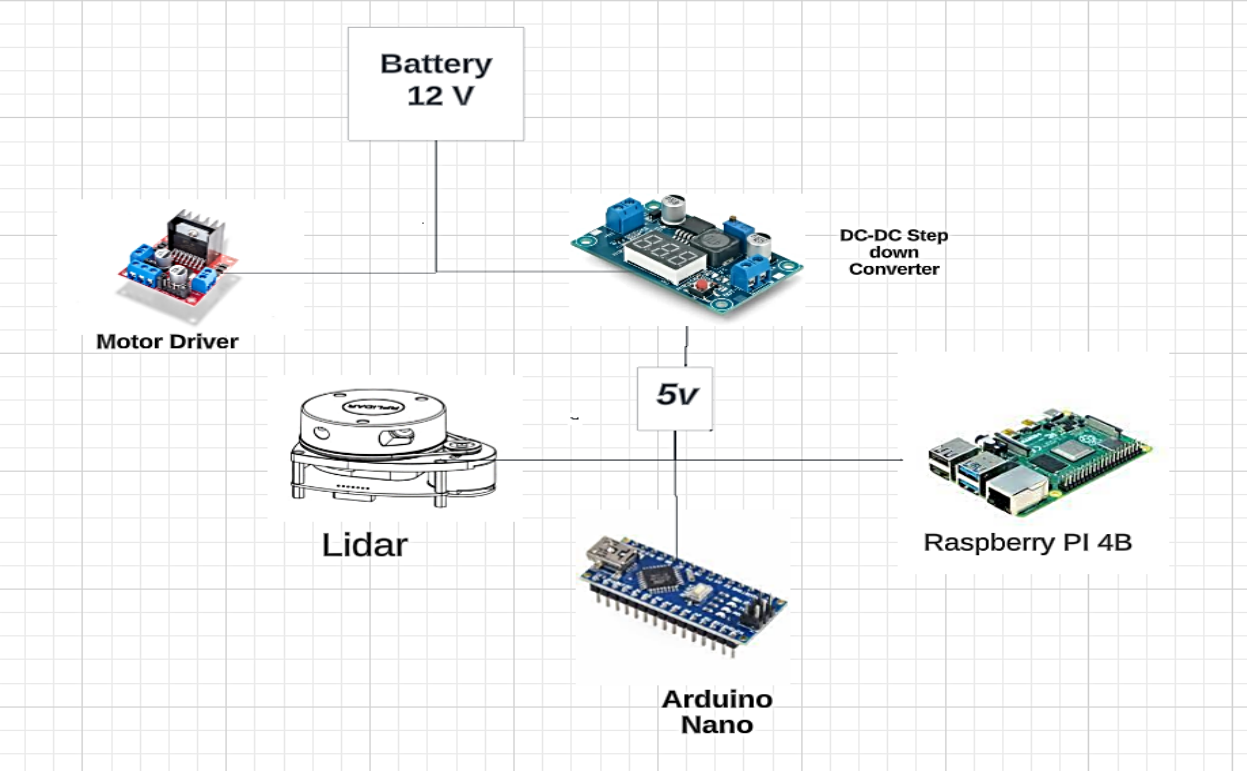


Figure 122 (2.7.5.b) Power Circuit

3.Chapter Three: Project Execution

3.1. Timeline and Tasks

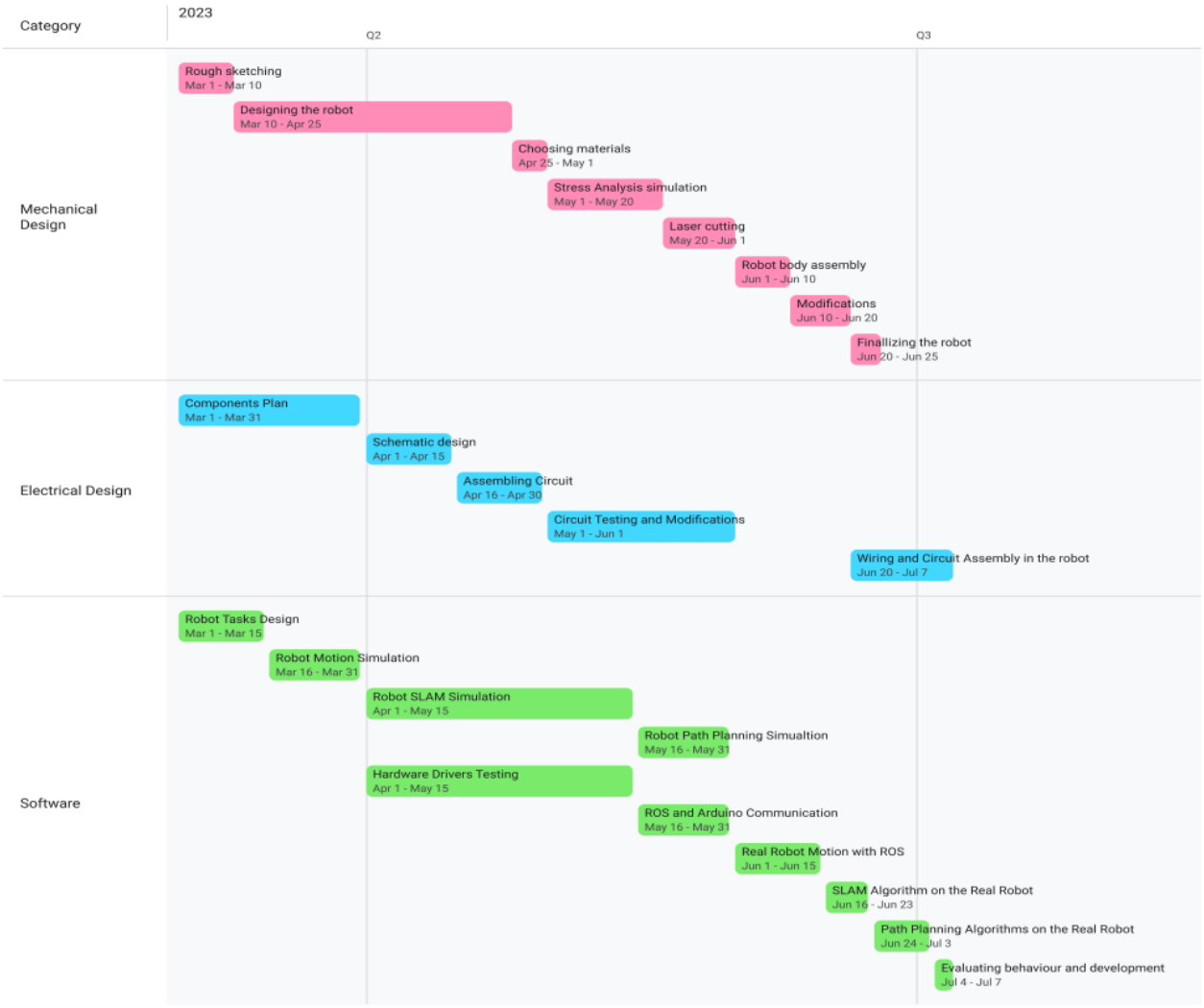


Figure 123 (3.1) Project Timeline

3.2. Mechanical subsystem

3.2.1. SolidWorks Analysis

- The main parts of the robot are laser cut, as we knew before, the material is acrylic, both didn't exist on SolidWorks, so we had to add them manually.

- There are no external loads. So, the analysis was on the weight of the main parts, and we added the other parts as weight loads.

3.2.1.1. Stress analysis

Name	Type	Min	Max
Stress1	VON: von Mises Stress	0 N/mm^2 (MPa) Node: 47794	54 N/mm^2 (MPa) Node: 8603

Table 12 (3.2.1.1) stress analysis

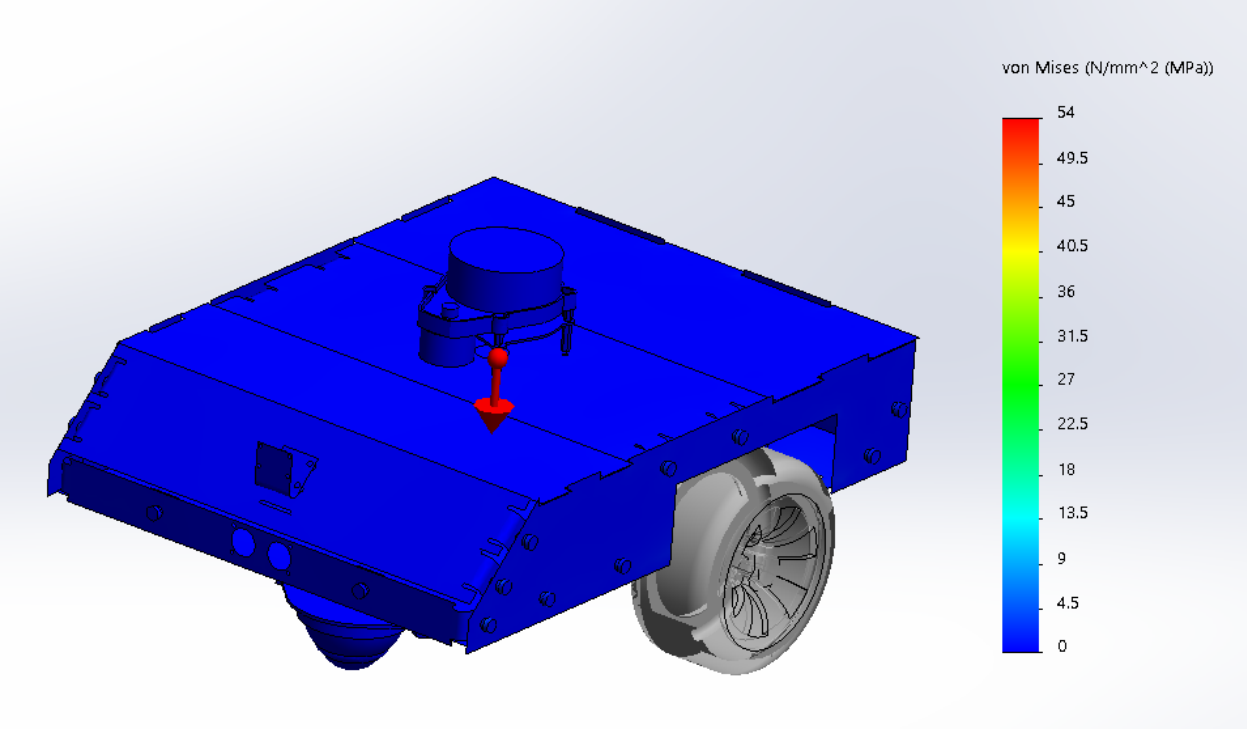


Figure 124 (3.2.1.1) stress analysis graph

3.2.1.2. displacement analysis

Name	Type	Min	Max
Displacement1	URES: Resultant Displacement	0.000e+000mm Node: 47794	8.993e-001mm Node: 94044

Table 13 (3.2.1.2) displacement analysis

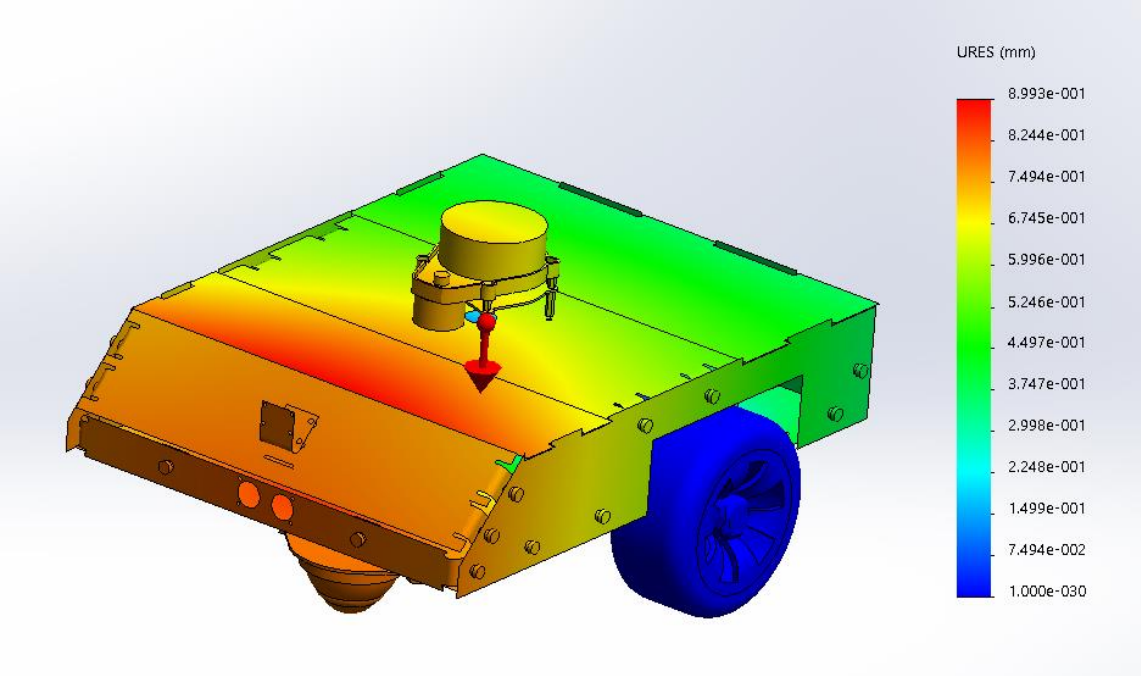


Figure 125 (3.2.1.2) Strain analysis graph

3.2.1.3. Strain analysis

Name	Type	Min	Max
Strain1	ESTRN: Equivalent Strain	0.000e+000 Element: 25237	7.005e-004 Element: 33195

Table 14 (3.2.1.3) Strain analysis

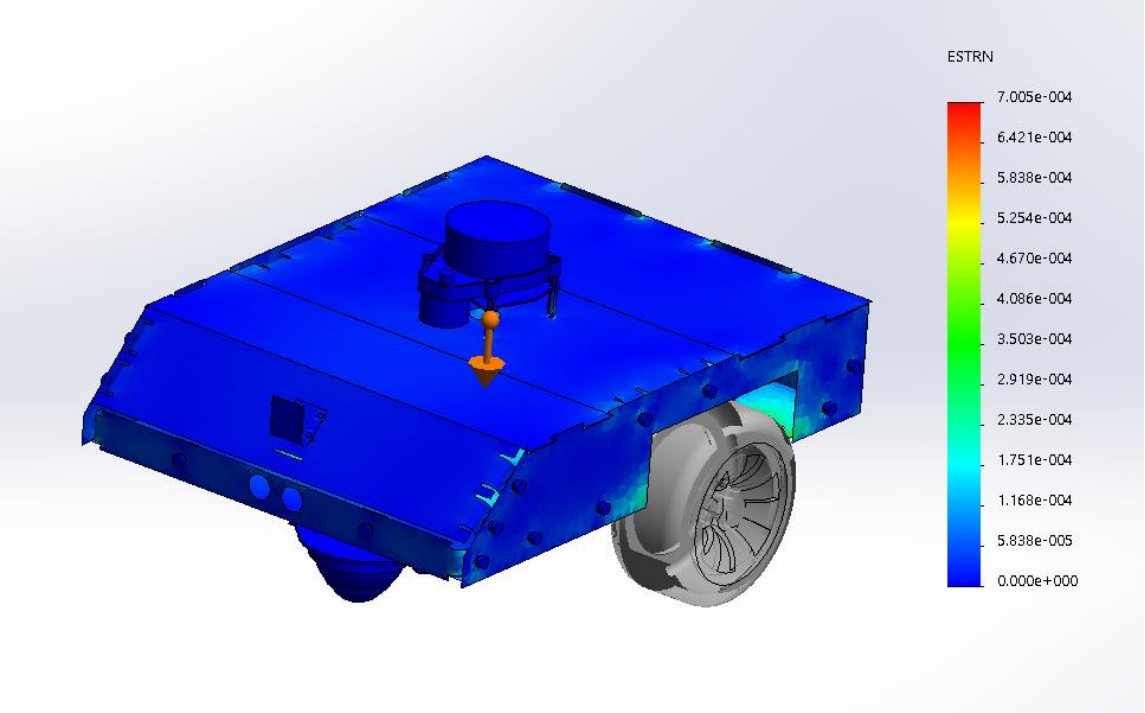


Figure 126 (3.2.1.3) strain analysis graph

3.2.1.4. Fatigue check

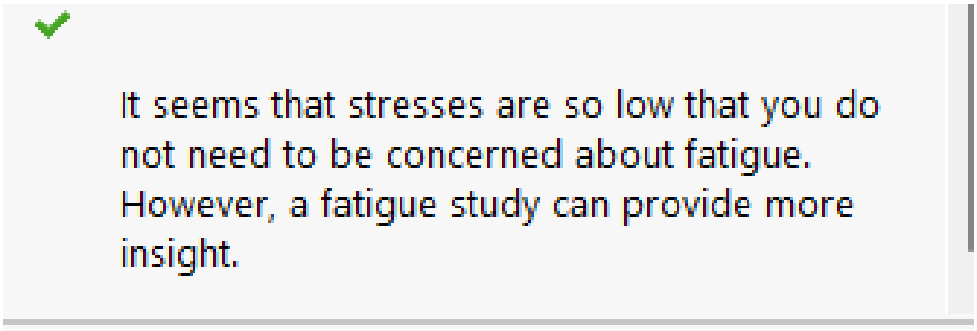


Figure 127 (3.2.1.4) Fatigue check

3.2.2. Mechanical Design

After we designed our CAD model, and checked the stress analysis, we moved on to manufacturing. Our Body is made from 5mm Acrylic black sheets, that are laser cut, then being bent 90 degrees bends for assembly with M6 screws.

Figure 128 (3.2.2.a) Mechanical Design pre-assembly

We designed the top part as three pieces for easy maintenance.

Figure 129 (3.2.2.b) Semi Assembled Mechanical Design

We used 13cm diameter rubber wheels mounted with a coupling to the motors, initially using a different type of caster wheels, that suffered from a slipping problem, so we switched to a ball caster wheel.

Figure 130 (3.2.2.c) Semi Assembled Mechanical Design

Then we assembled the body using spacers for mounting the lidar.

Figure 131 (3.2.2.d) Front view of the Mechanical Body

Figure 132 (3.2.2.e) Top View of the mechanical body

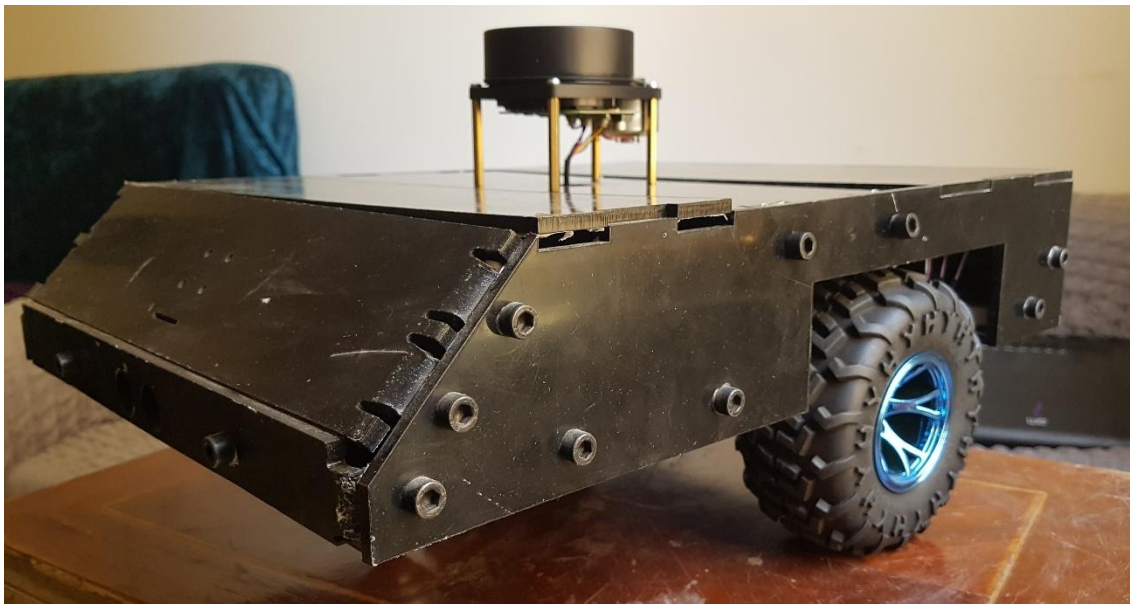


Figure 133 (3.2.2.f) Side view of the mechanical Body

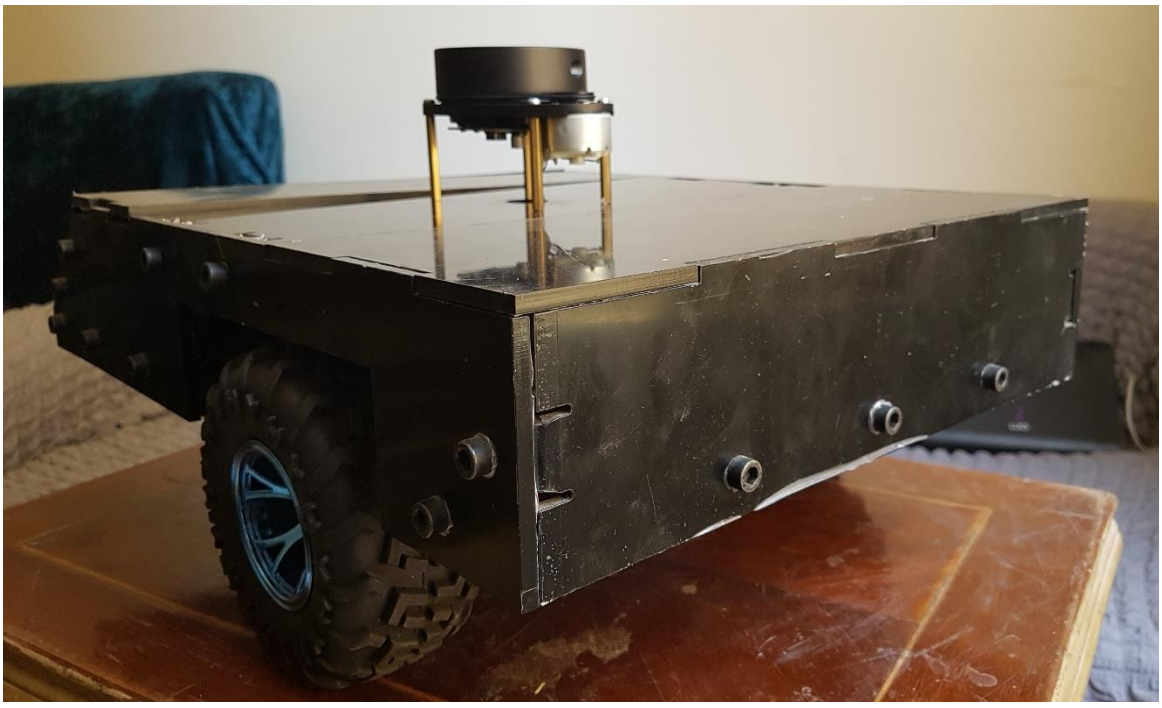


Figure 134 (3.2.2.g) Back view of the mechanical body

3.3. Electrical design

According to our Schematic, we connected our components and made the wiring using terminals so that we can have a common Vcc and ground. We used two buck converters for current safety, one is used for the 5V of the raspberry pi and the other for a powered USB hub to power the Arduino and LIDAR.

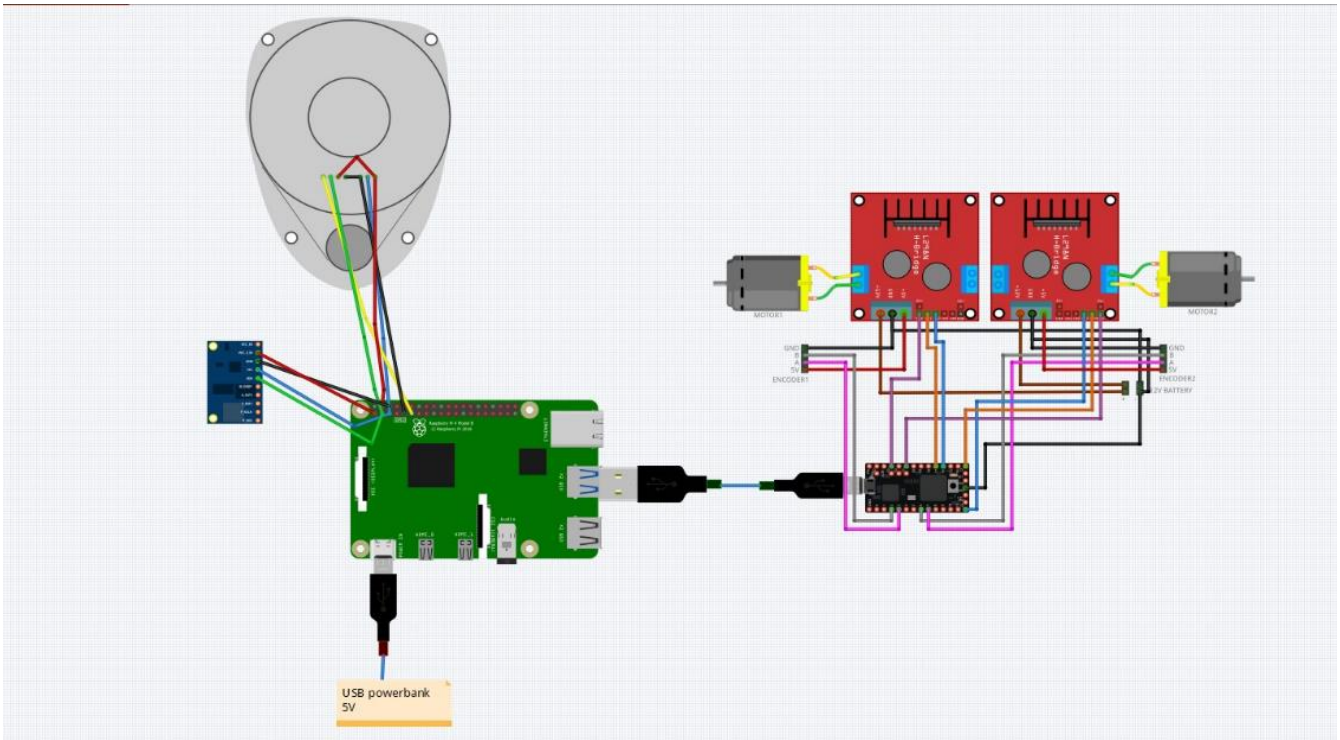


Figure 135 (3.3.a) Circuit Schematic

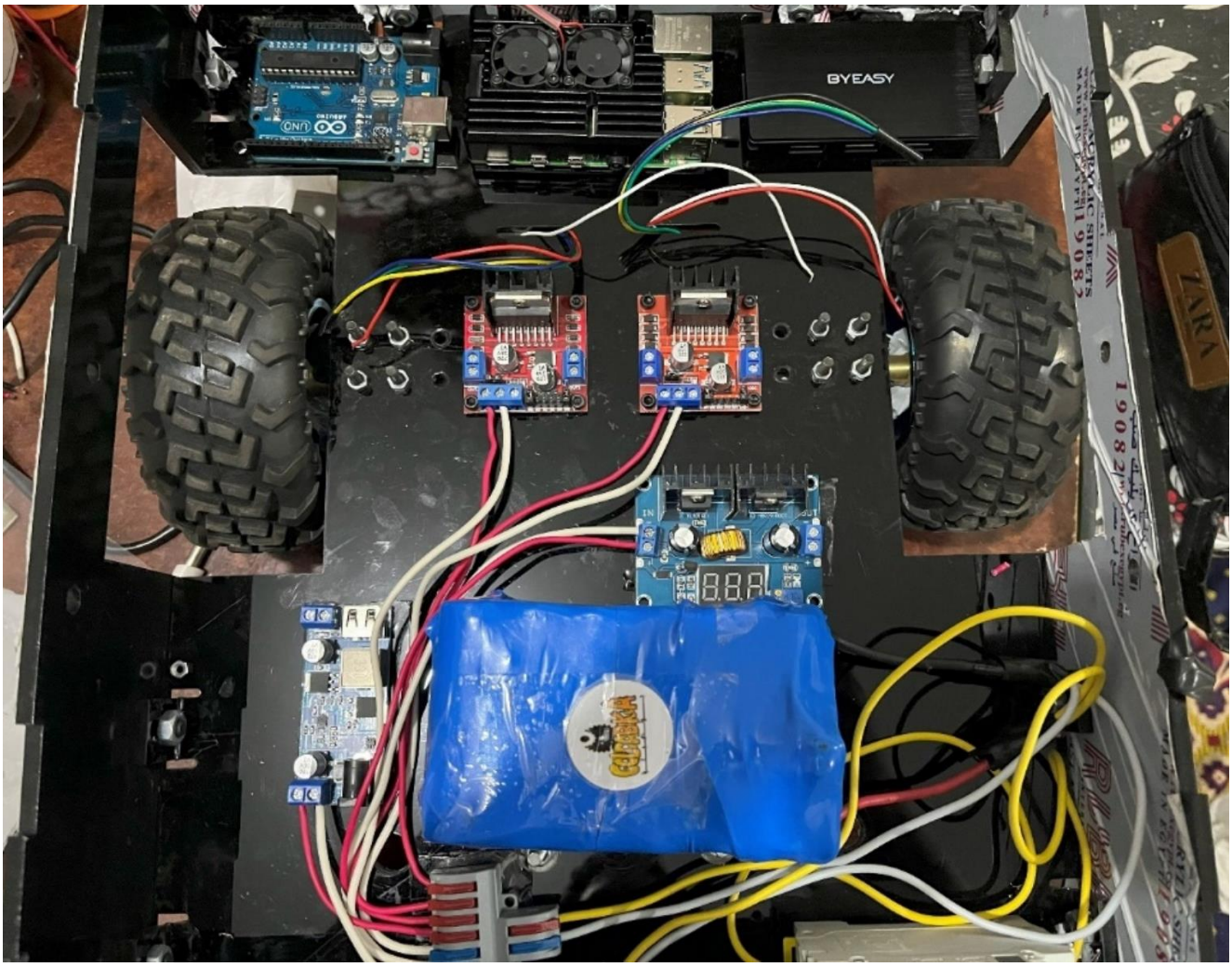


Figure 136 (3.2) Electric Circuit in the vehicle

3.4. Software

3.4.1 Unified Robot Description Format (URDF)

URDF (Unified Robot Description Format) is an XML-based file format used to describe the physical properties and structure of a robot in the Robot Operating System (ROS). It is commonly used to represent the kinematics, dynamics, and visual aspects of a robot in a standardized manner. URDF files provide a way to define the robot's physical components, such as links, joints, sensors, and visual models.

We made our URDF to be used in the simulation and RViZ, added to the robot base is the LIDAR and camera, and we made our origin at the center of the wheels so that we can make the other transformation easier.

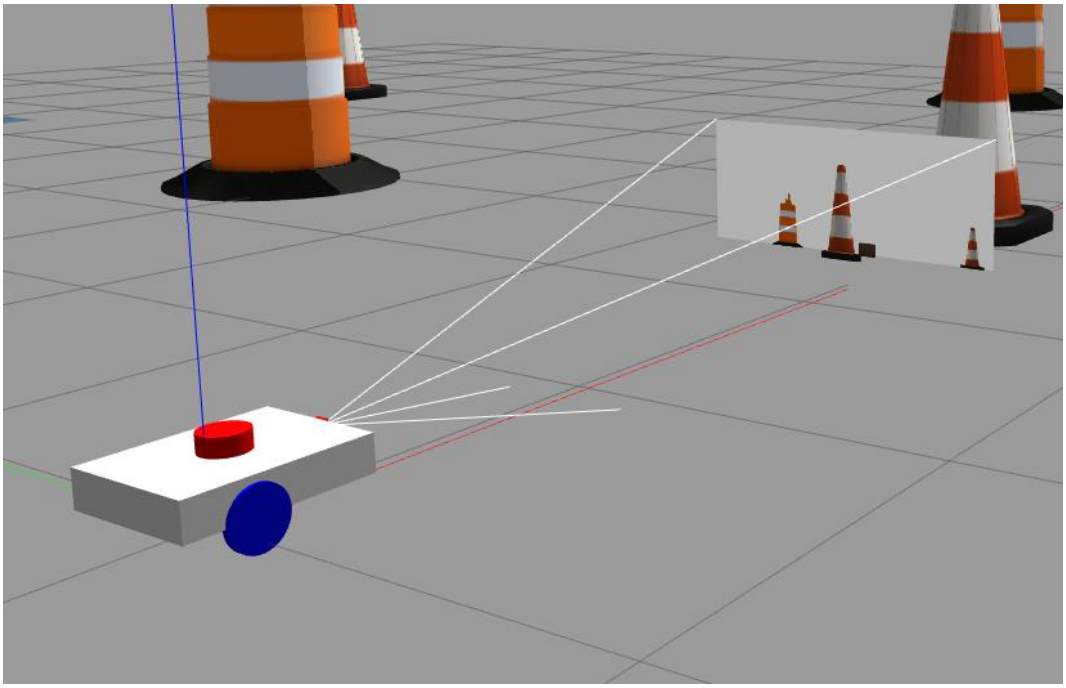


Figure 137 (3.4.1.a) URDF with the Camera view

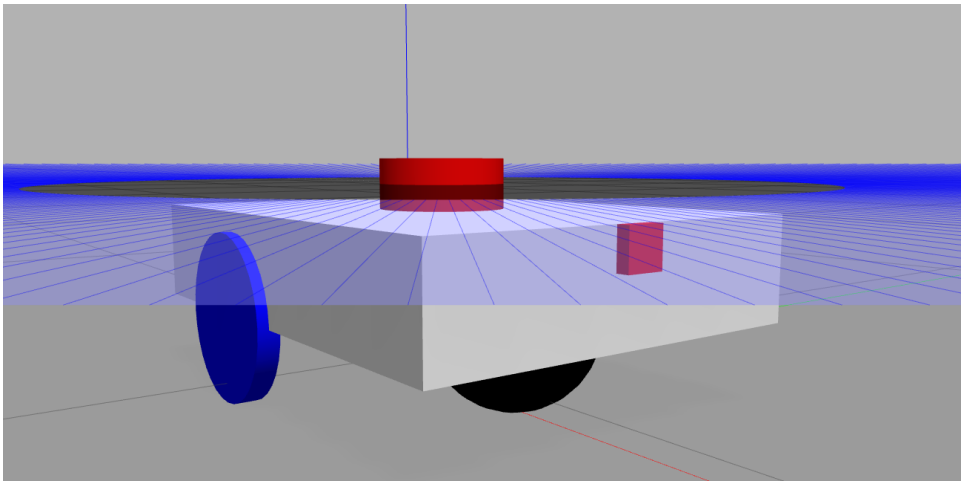


Figure 138 (3.4.1.b) URDF with the LIDAR rays

3.4.2 Robot Files

3.4.2.1 ros2_control

In the robot package have a description folder where we have the URDF files and ros2_control xacro that works in simulation and real robot mode to detect the Arduino port knowing the Wheel diameter and wheel Separation **(the full code in appendix: 2.1)**

```
<xacro:unless value="$(arg sim_mode)">
        <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/ttyACM0</param>
                <param name="baud_rate">57600</param>
                <param name="timeout">1000</param>
                <param name="enc_counts_per_rev">1910</param>
            </hardware>
```

Figure 139 (3.4.2.1) ros2_control hardware detection

3.4.2.2 Launch files

The first launch file gets all the urdf files to be shown (rsp.launch), **(the full code in appendix: 2.2).**

```
def generate_launch_description():

    # Check if we're told to use sim time
    use_sim_time = LaunchConfiguration('use_sim_time')
    use_ros2_control = LaunchConfiguration('use_ros2_control')

    # Process the URDF file
    pkg_path = os.path.join(get_package_share_directory('hero_robot'))
    xacro_file = os.path.join(pkg_path,'description','robot.urdf.xacro')
    # robot_description_config = xacro.process_file(xacro_file).toxml()
    robot_description_config = Command(['xacro ', xacro_file, ' use_ros2_control:=',
use_ros2_control, ' sim_mode:=', use_sim_time])

    # Create a robot_state_publisher node
    params = {'robot_description': robot_description_config, 'use_sim_time':
use_sim_time}
    node_robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        parameters=[params]
    )
```

Figure 140 (3.4.2.2.a) The launch file that gets the urdf files.

The 2nd launch file is the robot launch file that triggers the ros2_control files and lidar files and gazebo and rviz files **(full code in the appendix 2.3)**.

```
package_name='hero_robot'

    rsp = IncludeLaunchDescription(
                PythonLaunchDescriptionSource([os.path.join(
                    get_package_share_directory(package_name),'launch','rsp.launch.py'
                )]), launch_arguments={'use_sim_time': 'false', 'use_ros2_control':
'true'}.items()
    )


    twist_mux_params =
os.path.join(get_package_share_directory(package_name),'config','twist_mux.yaml')
    twist_mux = Node(
            package="twist_mux",
            executable="twist_mux",
            parameters=[twist_mux_params],
            remappings=[('/cmd_vel_out','/diff_cont/cmd_vel_unstamped')]
        )



    robot_description = Command(['ros2 param get --hide-type /robot_state_publisher
robot_description'])

    controller_params_file =
os.path.join(get_package_share_directory(package_name),'config','my_controllers.yaml')

    controller_manager = Node(
        package="controller_manager",
        executable="ros2_control_node",
        parameters=[{'robot_description': robot_description},
                    controller_params_file]
    )
```

Figure 141 (3.4.2.2.b) Robot launch File

3.4.2.3 Navigation Launch files

3.4.2.3.1. Mapping Launch File

This is used to launch the slam algorithm to start constructing a map, we used the cartographer package which is the same as slam_toolbox.

```
import os
from launch import LaunchDescription
from ament_index_python.packages import get_package_share_directory
from launch_ros.actions import Node



def generate_launch_description():

    cartographer_config_dir =
os.path.join(get_package_share_directory('dhero_navigation'), 'config')
    configuration_basename = 'cartographer.lua'

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

    grid = Node(
            package='cartographer_ros',
            executable='occupancy_grid_node',
            output='screen',
            name='occupancy_grid_node',
            parameters=[{'use_sim_time': True}],
            arguments=['-resolution', '0.05', '-publish_period_sec', '1.0'])
    return LaunchDescription([
        cartographer, grid
    ])
```

Figure 142 (3.4.2.3.1) SLAM algorithm launch file

3.4.2.3.2. Localization Launch File

This file is used to launch the created map in rviz with the robot to make sure it is accurate when navigating the saved map compared to the real environment.

```
import os

from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.substitutions import LaunchConfiguration
from launch.conditions import IfCondition

def generate_launch_description():

    package_name = "hro_navigation"

    nav2_yaml = os.path.join(get_package_share_directory(package_name), 'config',
'amcl_config.yaml')
    map_file = os.path.join(get_package_share_directory(package_name), 'config',
'obs_map.yaml') #change to your map
    rviz_config_file = os.path.join(get_package_share_directory(package_name),
"rviz", "main.rviz")
    use_rviz = LaunchConfiguration("rviz", default=False)

    return LaunchDescription([
        Node(
            package='nav2_map_server',
            executable='map_server',
            name='map_server',
            output='screen',
            parameters=[{'use_sim_time': True},
                        {'yaml_filename':map_file}]
        ),

        Node(
            package='nav2_amcl',
            executable='amcl',
            name='amcl',
            output='screen',
            parameters=[nav2_yaml]
        ),

        Node(
            package='nav2_lifecycle_manager',
            executable='lifecycle_manager',
            name='lifecycle_manager_localization',
            output='screen',
            parameters=[{'use_sim_time': True},
                        {'autostart': True},
                        {'node_names': ['map_server', 'amcl']}]
        ),
        Node(
        package= "rviz2",
        executable= "rviz2",
        arguments=["-d", rviz_config_file],
        output= "screen",
        condition=IfCondition(use_rviz)
    )

    ])
```

Figure 143 (3.4.2.3.2) Localization Launch File

3.4.2.3.3. Path Planning Launch File

After Creating the map and launching the localization file, we use this file to trigger the path planning algorithm to trigger the desired pose in rviz so the robot can reach it.

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():

    controller_yaml = os.path.join(get_package_share_directory('hero_navigation'), 'config',
'controller.yaml')
    bt_navigator_yaml = os.path.join(get_package_share_directory('hero_navigation'), 'config',
'bt_navigator.yaml')
    planner_yaml = os.path.join(get_package_share_directory('hero_navigation'), 'config',
'planner_server.yaml')
    recovery_yaml = os.path.join(get_package_share_directory('hero_navigation'), 'config',
'recovery.yaml')


    return LaunchDescription([
        Node(
            package='nav2_controller',
            executable='controller_server',
            name='controller_server',
            output='screen',
            parameters=[controller_yaml]),

        Node(
            package='nav2_planner',
            executable='planner_server',
            name='planner_server',
            output='screen',
            parameters=[planner_yaml]),

        Node(
            package='nav2_recoveries',
            executable='recoveries_server',
            name='recoveries_server',
            parameters=[recovery_yaml],
            output='screen'),

        Node(
            package='nav2_bt_navigator',
            executable='bt_navigator',
            name='bt_navigator',
            output='screen',
            parameters=[bt_navigator_yaml]),

        Node(
            package='nav2_lifecycle_manager',
            executable='lifecycle_manager',
            name='lifecycle_manager_pathplanner',
            output='screen',
            parameters=[{'autostart': True},
                        {'node_names': ['planner_server',
                                        'controller_server',
                                        'recoveries_server',
                                        'bt_navigator']}])
    ])
```

Figure 144 (3.4.2.3.3) Path Planning Launch file

3.5. Simulation

3.5.1 Camera

The camera Module got added to the URDF and described in Gazebo. It shows the environment while it is being navigated, and this can be added o rviz to use different compression methods.

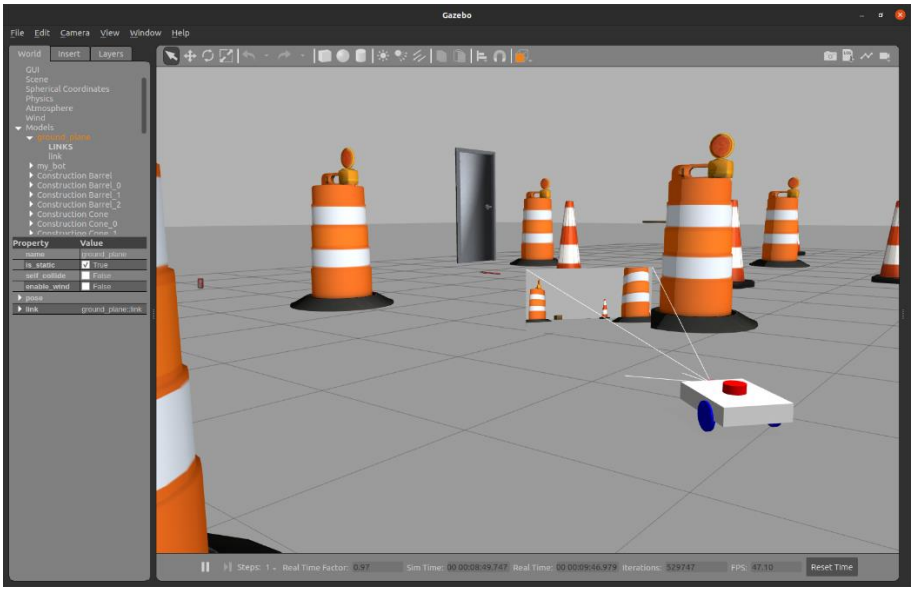


Figure 145 (3.5.1.a) Camera view in Gazebo

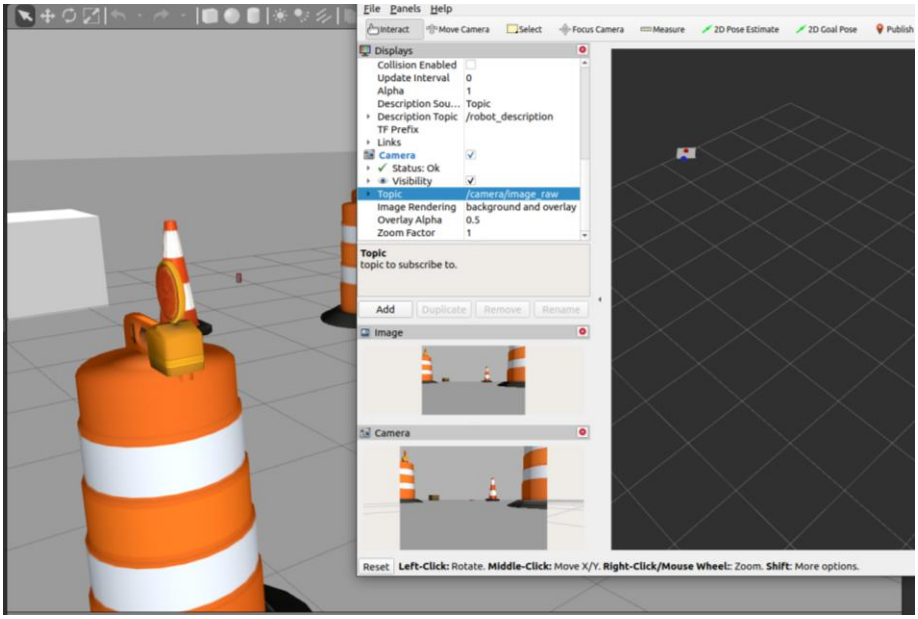


Figure 146 (3.5.1.b) Camera View in Rviz

3.5.2. LIDAR

The Lidar is attached to URDF at the top of the robot sensing the environment and used in the SLAM process in Rviz.

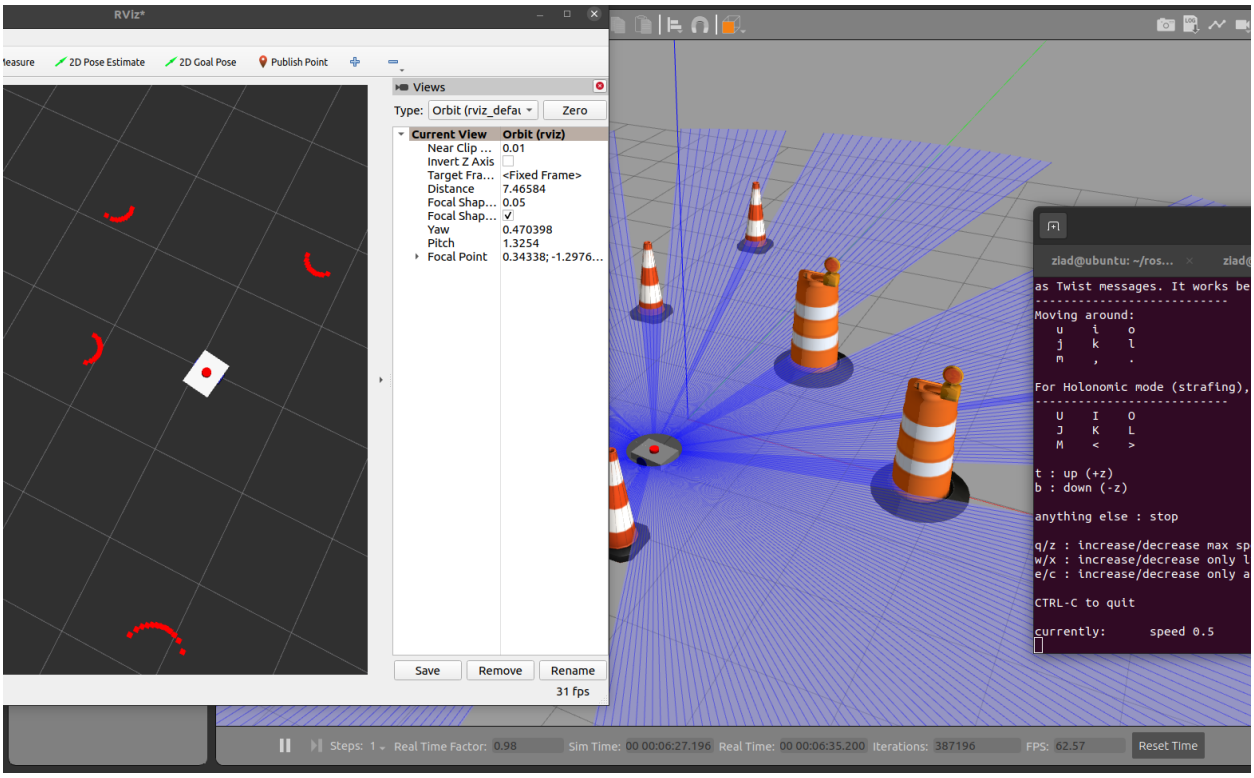


Figure 147 (3.5.2) Lidar view in Gazebo and Rviz

3.5.3. Links

The URDF shows different link origins, and an overview on the transformations between the different links.

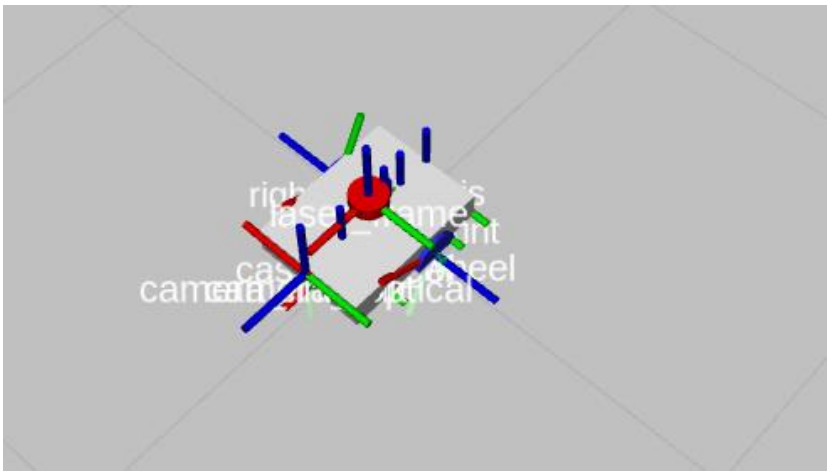


Figure 148 (3.5.3.a) URDF links

The rqt_tree shows the different links and topics talking to each other with the parent-child relation, to make sure that everything is working.

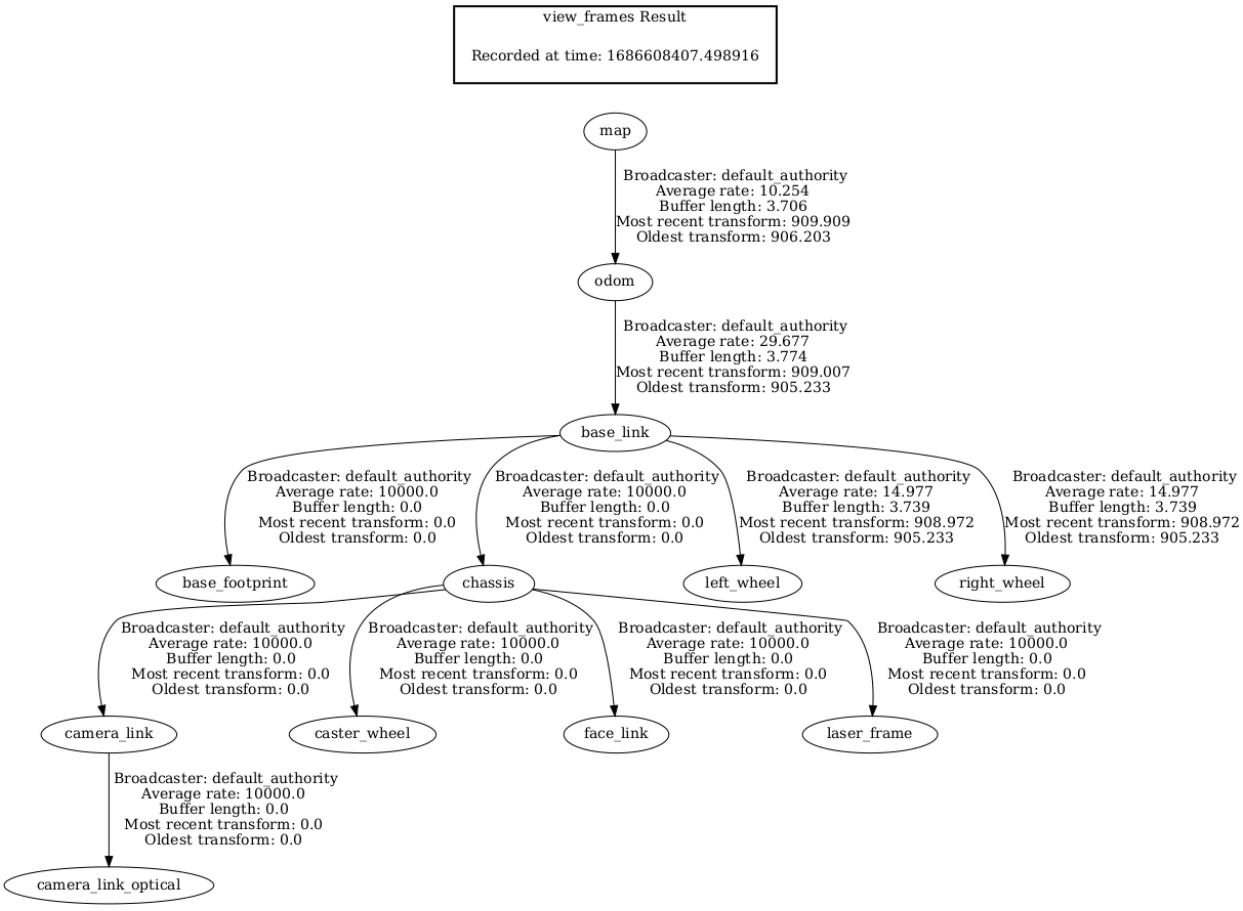


Figure 149 (3.5.3.b) rqt-tree showing links and topics of the robot

3.5.4. SLAM

The process starts by running the robot launch files and the teleop node to run the robot manually, so it discovers its environment and construct the map using the cartographer SLAM package. It is preferred to be exploring the environment in low speeds so we can minimize the errors.

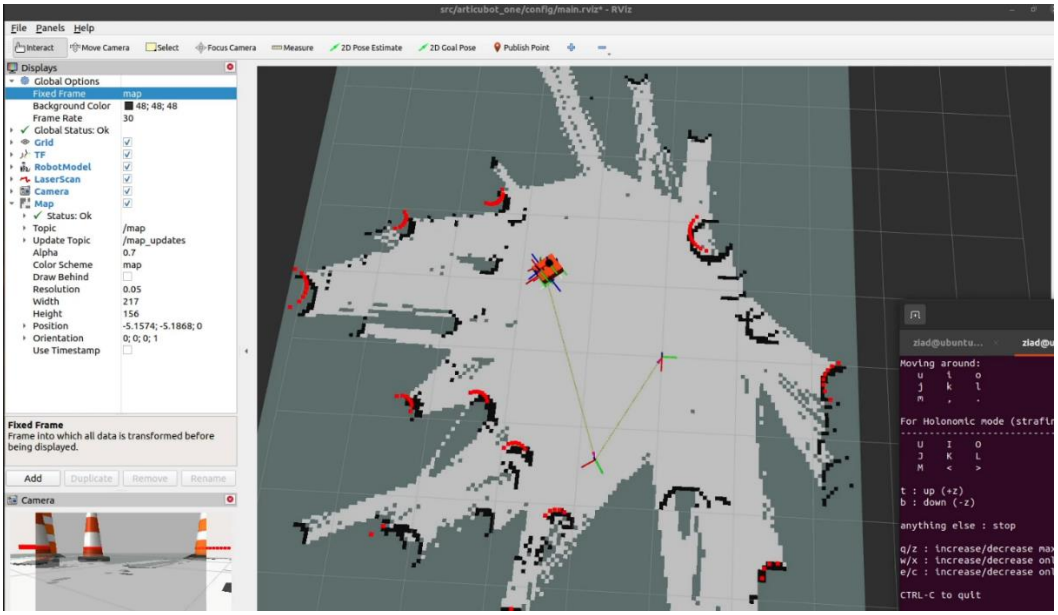


Figure 150 (3.5.4.a) SLAM Process Starting

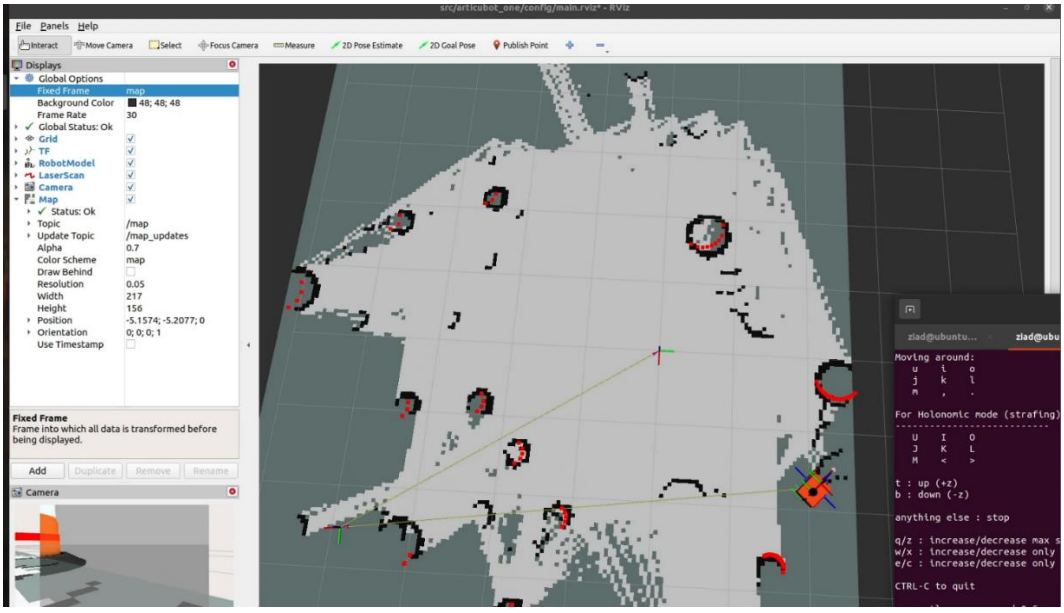


Figure 151 (3.5.4.b) SLAM Process After some exploration

3.5.5. Localization

After constructing the map, it is summoned via the localization that uses nav2_map_server to publish the saved map from the SLAM process. We can notice that the Lidar readings almost overlap the created map, but due to small errors in the odometry there may be a small shift.

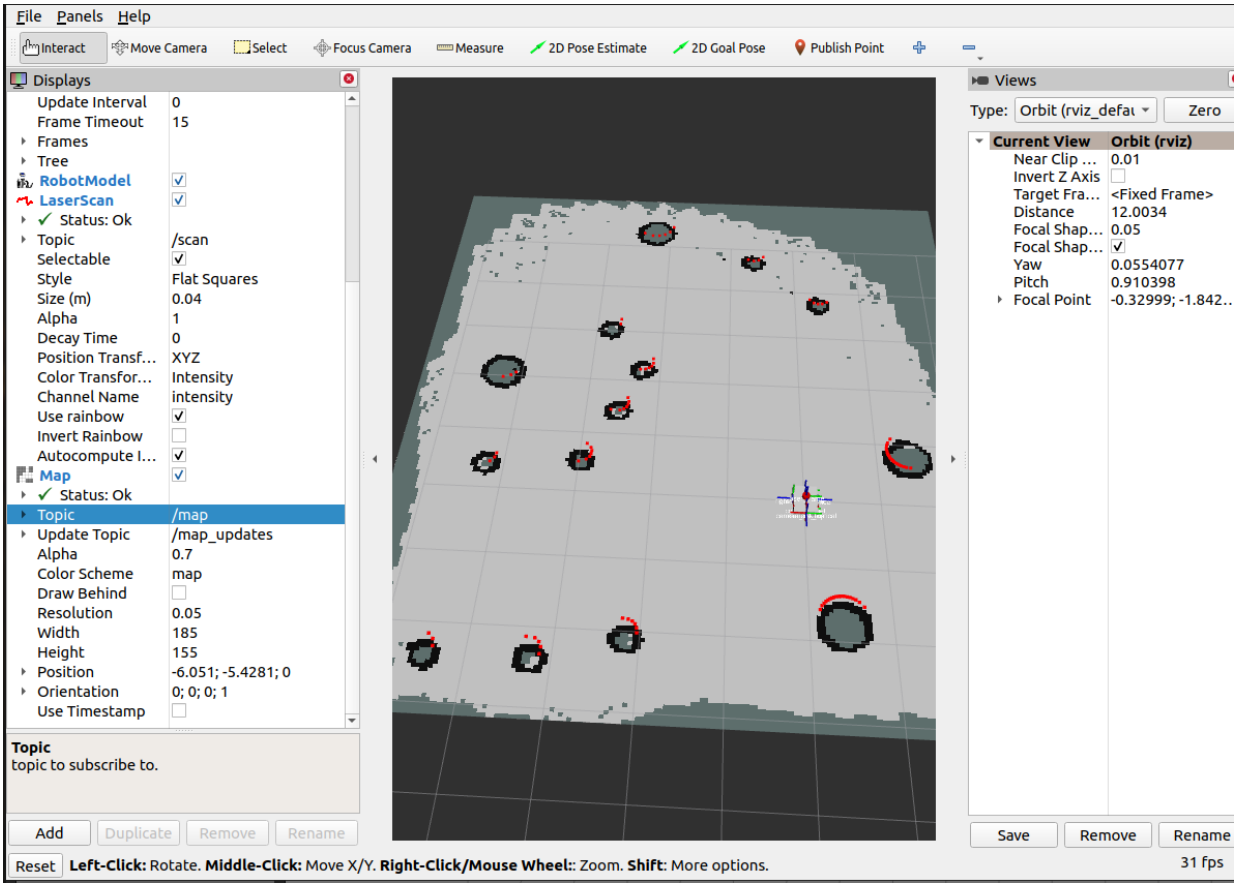


Figure 152 (3.5.5) Robot Localization Process

3.5.6. Path Planning

After the map being published to visualize how the path planning works we can see the details in the global costmap. The global costmap in Nav2 is responsible for representing the occupancy of the environment and the associated costs for path planning at a global scale. It provides a bird's-eye view of the environment, allowing the robot to plan its path by considering obstacles, terrain, and other factors that might influence navigation. Each cell in the costmap is associated with a cost value that represents the desirability of traversing that cell. Costs can be based on factors like distance, terrain difficulty, or any other criteria defined by the user. Cells with higher costs are less desirable for path planning.

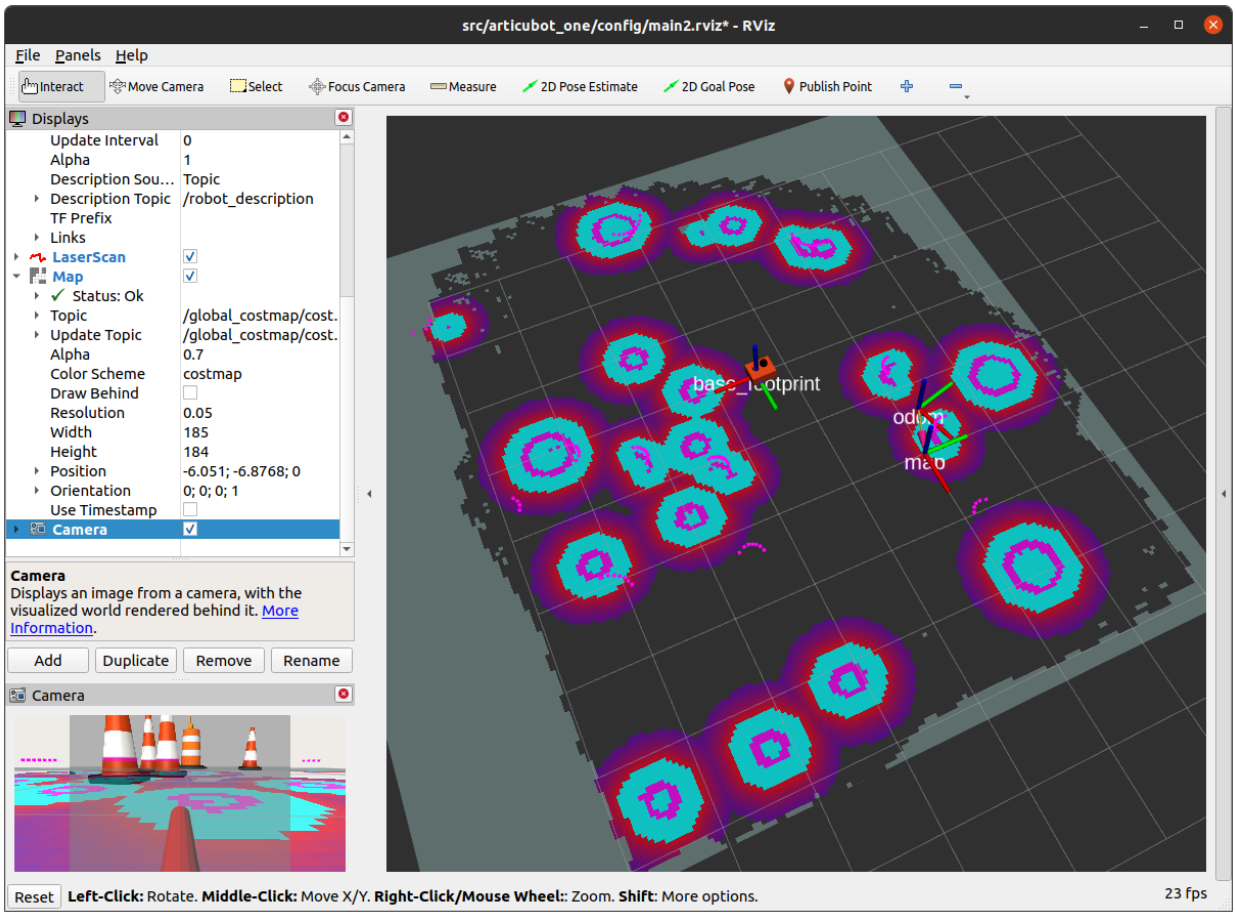


Figure 153 (3.5.6.a) Global costmap after the map being created.

In the path planning process, you use the 2D goal Pose in Rviz to choose the desired position and orientation. The robot then starts moving to the desired pose taking in consideriation its global costmap and considering the local planner in the case that an obstacle not in the map initially showed up.

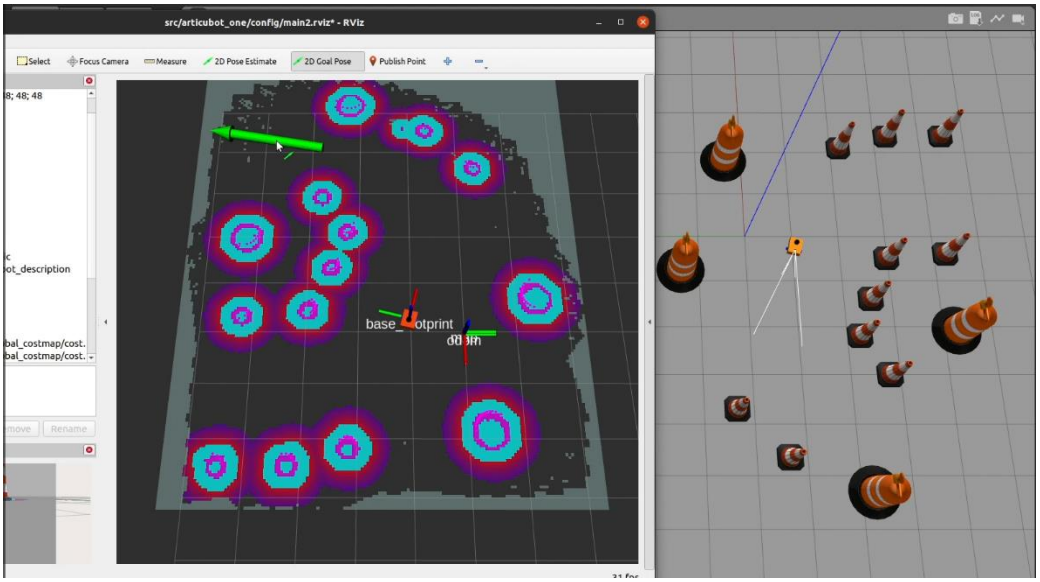


Figure 154 (3.5.6.b) Path Planning - Choosing the desired pose.

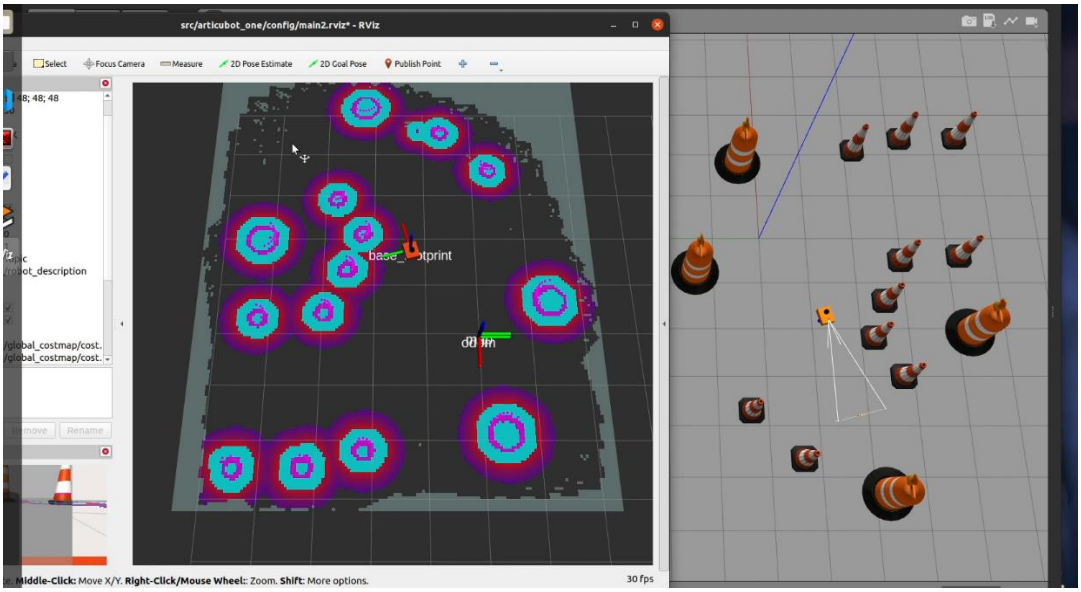


Figure 155 (3.5.6.c) Path Planning - robot moving to the desired pose

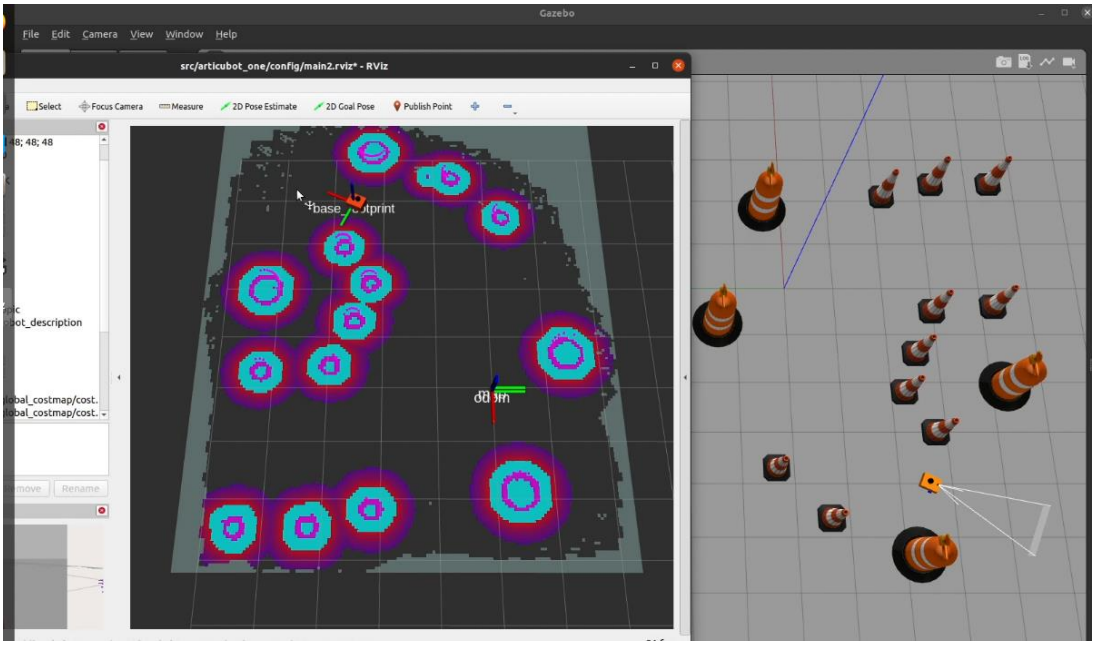


Figure 156 (3.5.6.d) Path Planning - Robot reached its goal.

4. Chapter Four: Testing and evaluation of the project

4.1. Field Test

In the first SLAM trial, the map was not accurate and has a lot of noise mostly due to odometry errors due to slip.

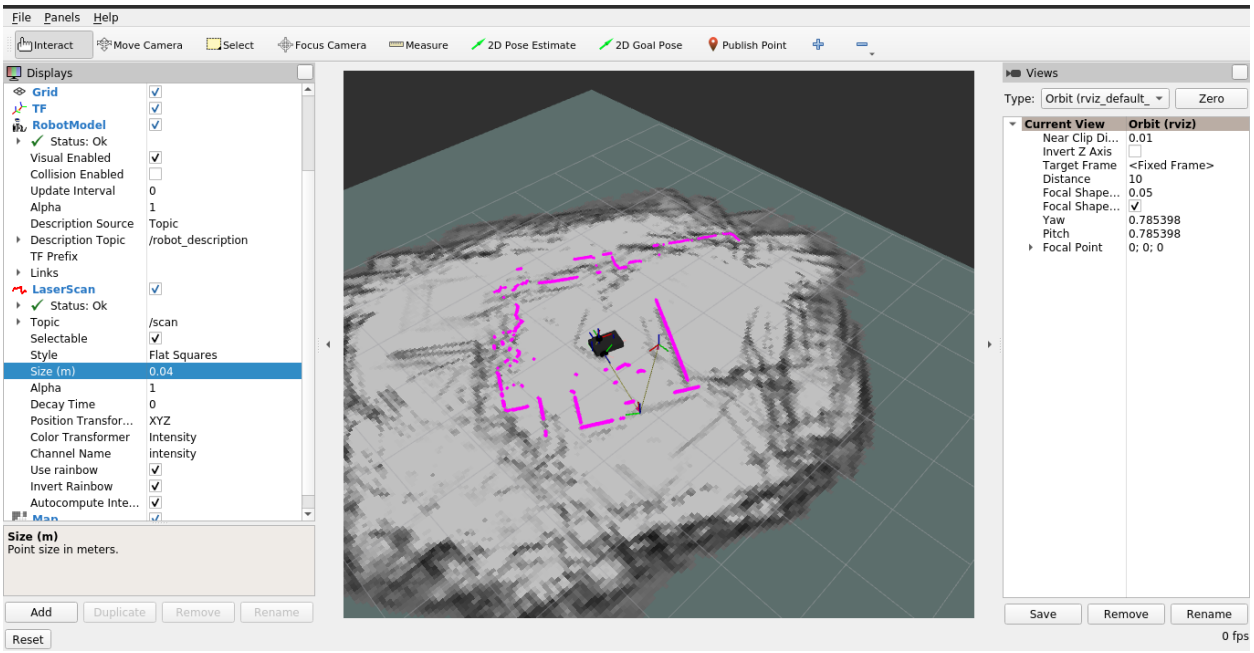


Figure 157 (4.1.a) Frist trial of SLAM on the real robot

After several experiments and tunning of parameters, while also lowering the exploration speed, we got a better result of the map. We still have some noise, but this may be as a result of wheel slip or friction caused by little vibrations of the coupling with the motor, making the encoder readings less accurate.

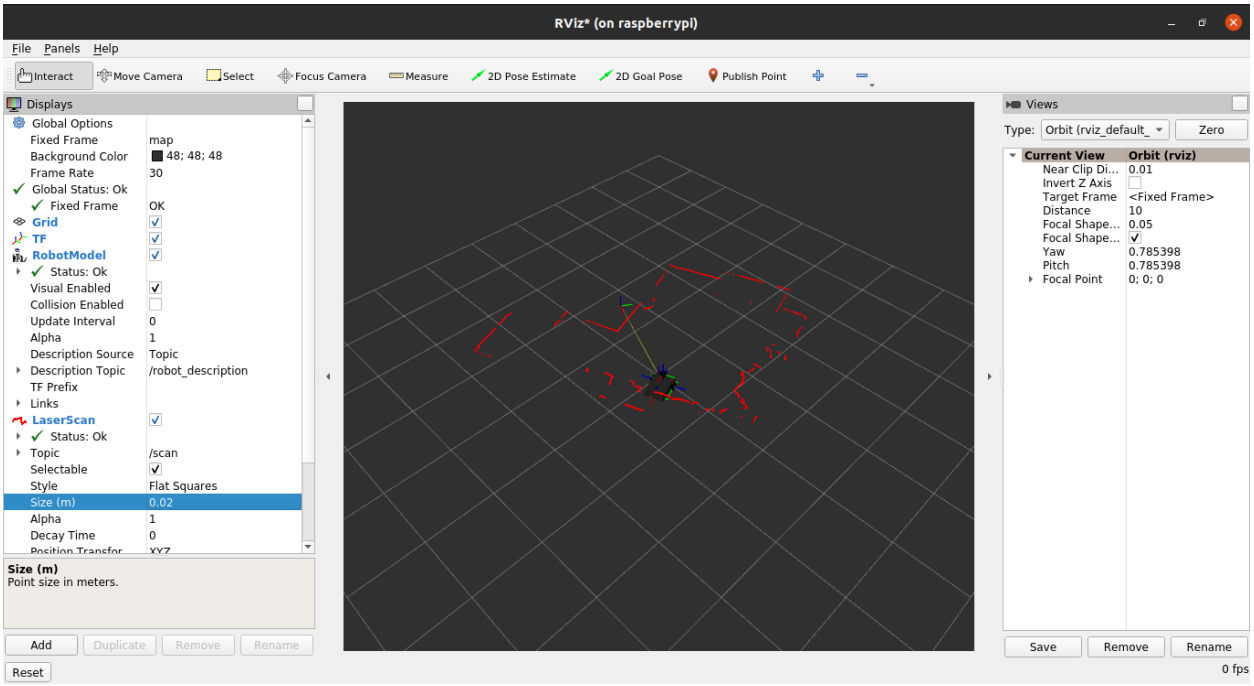


Figure 158 (4.1.b) Real Robot Lidar Perception

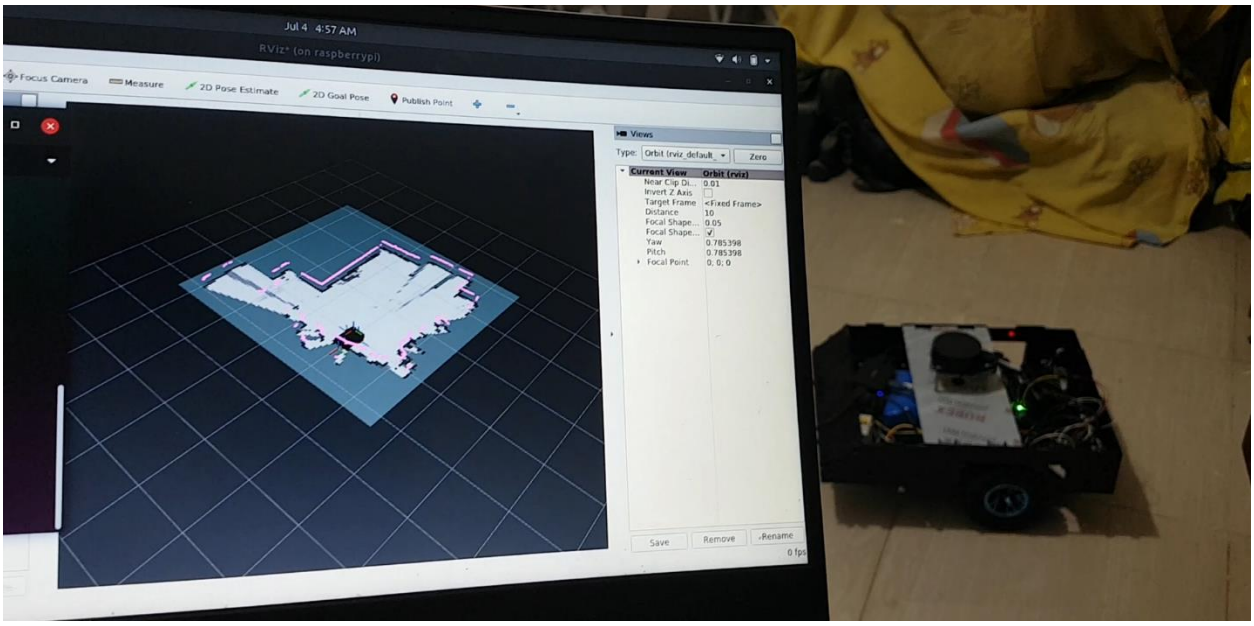


Figure 159 (4.1.c) Showing the robot in ground and its feedback in Rviz

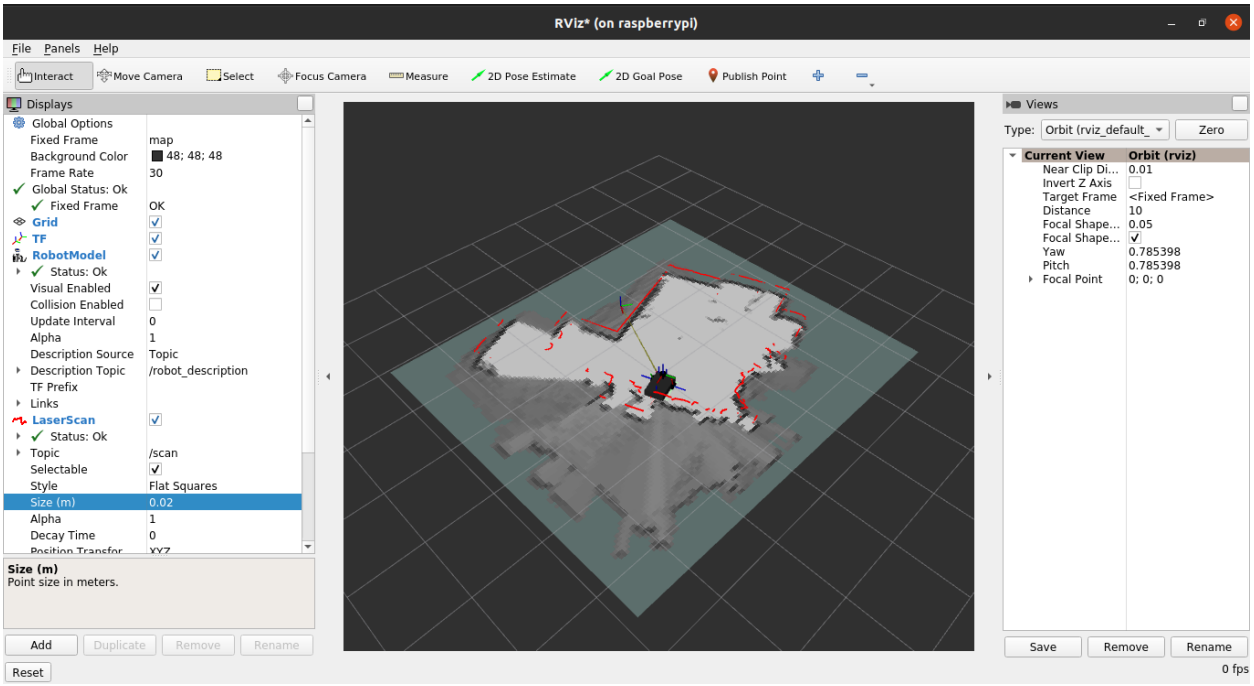


Figure 160 (4.1.d) SLAM Process building the map on the real robot.

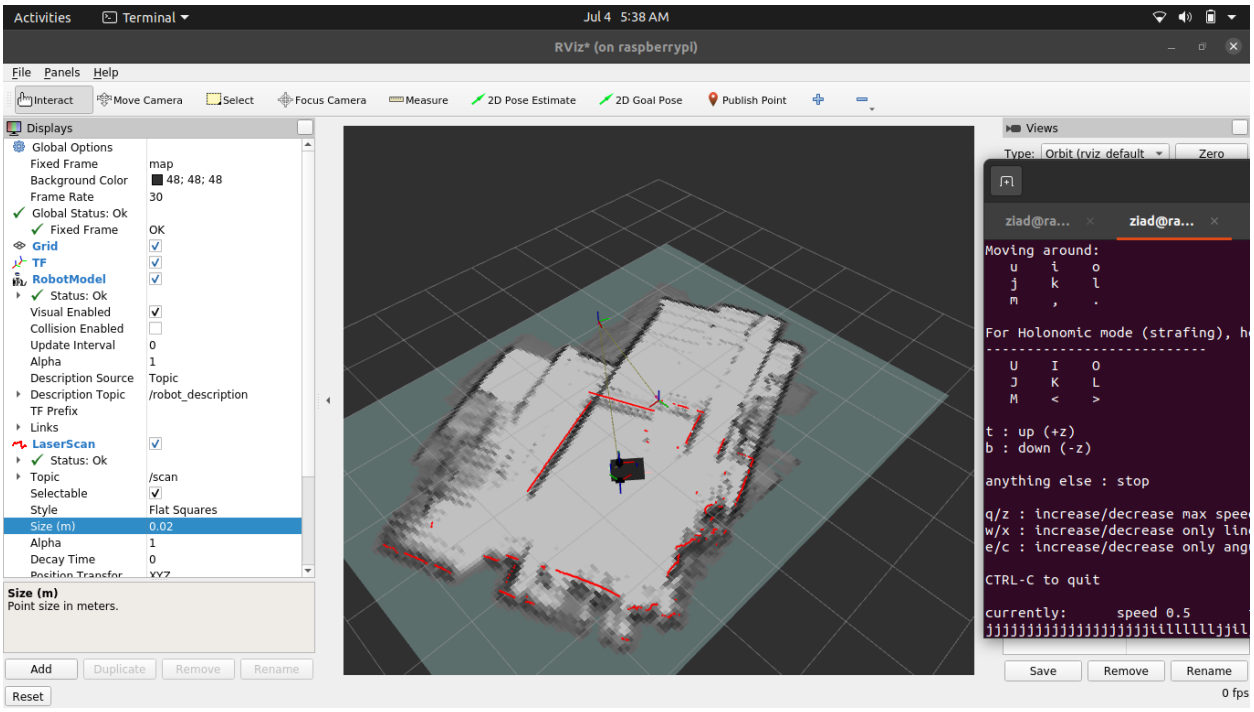


Figure 161 (4.1.e) A map built with SLAM process on the real robot.

4.2. Cost

Compared to the cost of the labor, specially in the repetitive tasks, these types of vehicles have an initial high capital cost (~15,000 L.E.), but saves a lot when comparing even with a 1 worker with the Egyptian minimum wage of 2700 L.E., the vehicle would pay its price in less than 6 months.

4.3. Environmental impact

The energy source and efficiency of the ground navigation robot play a significant role in its environmental impact. Having an electrical drivetrain enables us to have the ability to charge the batteries via renewable energy, which reduces the emissions.

4.4. Manufacturability

Our robot can be easily manufactured as laser cutting is currently automated at very high speed. The Assembly Bends can be done using bending machines with a high process volume, as the bends are 90 degrees, which makes them easier to accomplish.

4.5. Ethics

Ground navigation robots can automate repetitive and labor-intensive tasks, leading to increased efficiency and productivity. By taking over mundane or physically demanding activities, robots free up human resources for more creative and complex tasks. This can enhance job satisfaction and promote a healthier work-life balance.

They can also improve accessibility and inclusivity by providing assistance to individuals with disabilities or limited mobility. These robots can help with daily activities, transportation, or social interaction, promoting independence and inclusivity for people who may face mobility challenges.

Ground navigation robot developers and manufacturers can establish ethical guidelines and ensure transparency in the design, deployment, and operation of these robots. This includes considerations for privacy protection, data security, and adherence to legal and ethical standards. Openness about the capabilities, limitations, and potential risks of ground navigation robots fosters trust and responsible use.

4.6. Social & economic Impact

The development, production, deployment, and maintenance of ground navigation robots can create new job opportunities. These jobs range from manufacturing and assembly to robot operation, maintenance, and software development. Moreover, as automation frees up human workers from repetitive or physically demanding tasks, it allows them to focus on higher-level responsibilities, leading to skills development and more fulfilling work.

In industries facing labor shortages or demographic challenges, ground navigation robots can help fill the gaps and ensure continuity of operations. These robots can perform tasks that may be difficult for staff due to factors such as remote locations, labor market conditions, or population aging, ensuring business continuity and economic stability. This besides making the process autonomous that can be done remotely, economically has a high impact of reducing costs due to labor while being efficient at the specified tasks, saving time and money.

4.7. Health and safety

Ground navigation robots can be designed with advanced sensing technologies, obstacle avoidance systems, and safety protocols to enhance safety in various applications. By autonomously navigating their surroundings, these robots can reduce the risk of accidents, particularly in hazardous environments or situations where human presence may be unsafe.

4.8. Sustainability

Implementing a circular economy approach involves designing ground navigation robots with the aim of maximizing their lifespan, promoting repairability, and facilitating component reuse or recycling. This reduces waste, conserves resources, and minimizes the need for new production.

Conclusion:

Autonomous Navigation robots had been such an essential part of robotics due to their use in most fields, from the warehouses to minesweeping to delivering packages to field survey. Working with ROS2, ros2_control, slam_toolbox and nav2 helped a lot in tuning the parameters to overcome many of the non-ideal situations with the real robot, while being portable that it can work on many types or sizes of robots with minimal configurations. Our robot achieves its task of localization and mapping, to be used in path planning of the built in map. This opens the door for many developments in the future such as add-ones as a robot arm or a specific sensor to collect needed data in different fields.

References

1. 7 benefits of autonomous cars (thalesgroup.com)
2. How driverless cars will change our world - BBC Future
3. The future of autonomous vehicles (AV) | McKinsey
4. Regency style | art | Britannica
5. 7 Early Robots and Automatons | HISTORY
6. Automation - Digital Computers, Sensor Technology, and Artificial Intelligence | Britannica
7. Automation - Computer Process Control, Production Control, and Data Gathering and Analysis | Britannica
8. The Future of Artificial Intelligence and Cybernetics | OpenMind (bbvaopenmind.com)
9. Return of cybernetics | Nature Machine Intelligence
10. https://robotnik.eu/history-of-robots-and-robotics/
11. https://history-computer.com/shakey-the-robot/
12. https://www.researchgate.net/publication/323913016_Advances_in_sensing_and_processing_methods_for_three-dimensional_robot_vision
13. https://www.researchgate.net/publication/325249481_Autonomous_robots_for_harsh_environments_A_holistic_overview_of_current_solutions_and_ongoing_challenges
14. https://www.sciencedirect.com/topics/earth-and-planetary-sciences/autonomous-underwater-vehicle
15. https://www.sciencedirect.com/science/article/pii/S2667241323000113
16. https://news.stanford.edu/2019/01/16/stanfords-robotics-legacy/
17. https://nssdc.gsfc.nasa.gov/planetary/mesur.html

18. A Brief History of Military Robots Including Autonomous Systems (interestingengineering.com)
19. Automated guided vehicles and autonomous mobile robots for recognition and tracking in civil engineering - ScienceDirect
20. https://robotsguide.com/robots/packbot
21.https://www.researchgate.net/publication/4092514_The_DARPA_grand_challenge_-_development_of_an_autonomous_vehicle
22. https://www.bostondynamics.com/legacy
23. https://mars.nasa.gov/mars-exploration/missions/mars-exploration-rovers/
24. https://www.bostondynamics.com/products/spot
25. http://www.clearpathrobotics.com/assets/guides/noetic/husky/
26. https://www.eltis.org/in-brief/news/trust-and-safety-among-top-priorities-autonomous-vehicles
27. https://www.munciejournal.com/2022/11/ball-state-university-introduces-starship-food-delivery-robots-on-campus/
28. https://batteryindustry.tech/autonomous-agricultural-robot-rosie-by-eth-zurich-uses-ecovolta-batteries/
29. https://www.mdpi.com/1424-8220/21/23/7898
30. https://link.springer.com/10.1007%2F978-3-540-30301-5_49
31. https://www.mdpi.com/1424-8220/20/2/490
32. https://www.plainconcepts.com/autonomous-robots-security/
33. https://www.sciencedirect.com/science/article/abs/pii/S1366554522002150

34. https://ec.europa.eu/research-and-innovation/en/horizon-magazine/robot-rescuers-help-save-lives-after-disasters

35. https://www.carvilleplastics.com/latest_news/key-properties-acrylic/

36. https://www.creativemechanisms.com/blog/injection-mold-3d-print-cnc-acrylic-plastic-pmma

37. https://www.carvilleplastics.com/latest_news/key-properties-cast-acrylic-pmma/

38. https://3dfilaprint.com/product/esun-lcd-standard-resin-various-colours-1000gms/

39. https://www.theconstructsim.com/

40. https://docs.ros.org/en/foxy/index.html

41. https://gazebosim.org/home

42. http://wiki.ros.org/rviz

43. https://www.guru99.com/tcp-ip-model.html

44. https://control.ros.org/master/index.html

45. https://articulatedrobotics.xyz/mobile-robot-13-ros2-control-real/

46. https://www.mathworks.com/discovery/slam.html

47. https://thekalmanfilter.com/kalman-filter-explained-simply/

48. https://www.mathworks.com/videos/autonomous-navigation-part-3-understanding-slam-using-pose-graph-optimization-1594984678407.html

49. https://www.unravel.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaabebgwr&download=1

50. https://github.com/SteveMacenski/slam_toolbox

51. https://www.intechopen.com/chapters/62978

52. https://github.com/linorobot/linorobot

53. https://www.researchgate.net/figure/Block-diagram-of-Madgwicks-filter-in-basic-IMU-inertial-measurement-unit-application_fig3_340347515

54. https://towardsdatascience.com/extended-kalman-filter-43e52b16757d

55. https://arxiv.org/pdf/1809.02989.pdf

56. http://wiki.ros.org/amcl

57. Gayathri Rajendran, Uma V, Bettina O'Brien, Unified robot task and motion planning with extended planner using ROS simulator, Journal of King Saud University - Computer and Information Sciences, 2021.

58. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

59. https://navigation.ros.org/

60. https://kaiyuzheng.me/documents/navguide.pdf

61. https://www.elprocus.com/an-overview-of-arduino-nano-board/

62. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

63. https://components101.com/sensors/mpu6050-module

64. https://www.generationrobots.com/media/rplidar-a1m8-360-degree-laser-scanner-development-kit-datasheet-1.pdf

65. https://github.com/joshnewans/ros_arduino_bridge

66. http://meggi.usuarios.rdc.puc-rio.br/docs/riobotz_combot_tutorial.pdf

67. https://www.techtarget.com/whatis/definition/lithium-polymer-battery-LiPo

68. https://www.sciencedirect.com/science/article/pii/B9780857090690500100

1. ESP8266ex: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf
2. N-channel Enhancement-mode Power MOSFET: https://datasheetspdf.com/pdf/839826/AdvancedPowerElectronics/AP2312GN-HF-3/1
3. 8-Channel Analog Multiplexer/Demultiplexer: https://www.st.com/resource/en/datasheet/hcf4051.pdf
4. IR rectifier: https://www.alldatasheet.com/view.jsp?Searchword=Irlml2060trpbf&gclid=CjwKCAiAzKqdBhAnEiwAePEjkk7mEFCtCBFwUW1fy47vqLt4ttP95RWNm5xDQI9PnFitHKmb17wwHhoCuHsQAvD_BwE
5. MPU 6000: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf
6. Reflective Optical Sensor with Transistor Output: https://www.vishay.com/docs/83760/tcrt5000.pdf
7. Li-lon Battery Charger: https://www.mikrocontroller.net/attachment/273612/TP4056.pdf
8. Intelligent control LED integrated light source: https://media.digikey.com/pdf/Data%20Sheets/Seeed%20Technology/WS2813B_Ver.V5_10-20-19.pdf

Appendices

1. Microcontroller code [65]

1.1 Motor Driver

```
/*************************************************************
   Motor driver definitions

   Add a "#elif defined" block to this file to include support
   for a particular motor driver.  Then add the appropriate
   #define near the top of the main ROSArduinoBridge.ino file.

   *************************************************************/

#ifdef USE_BASE

#ifdef POLOLU_VNH5019
  /* Include the Pololu library */
  #include "DualVNH5019MotorShield.h"

  /* Create the motor driver object */
  DualVNH5019MotorShield drive;

  /* Wrap the motor driver initialization */
  void initMotorController() {
    drive.init();
  }

  /* Wrap the drive motor set speed function */
  void setMotorSpeed(int i, int spd) {
    if (i == LEFT) drive.setM1Speed(spd);
    else drive.setM2Speed(spd);
  }

  // A convenience function for setting both motor speeds
  void setMotorSpeeds(int leftSpeed, int rightSpeed) {
    setMotorSpeed(LEFT, leftSpeed);
    setMotorSpeed(RIGHT, rightSpeed);
  }
#elif defined POLOLU_MC33926
  /* Include the Pololu library */
```

```
37    #include "DualMC33926MotorShield.h"
38
39    /* Create the motor driver object */
40    DualMC33926MotorShield drive;
41
42    /* Wrap the motor driver initialization */
43    void initMotorController() {
44      drive.init();
45    }
46
47    /* Wrap the drive motor set speed function */
48    void setMotorSpeed(int i, int spd) {
49      if (i == LEFT) drive.setM1Speed(spd);
50      else drive.setM2Speed(spd);
51    }
52
53    // A convenience function for setting both motor speeds
54    void setMotorSpeeds(int leftSpeed, int rightSpeed) {
55      setMotorSpeed(LEFT, leftSpeed);
56      setMotorSpeed(RIGHT, rightSpeed);
57    }
58  #elif defined L298_MOTOR_DRIVER
59    void initMotorController() {
60      digitalWrite(RIGHT_MOTOR_ENABLE, HIGH);
61      digitalWrite(LEFT_MOTOR_ENABLE, HIGH);
62    }
63
64    void setMotorSpeed(int i, int spd) {
65      unsigned char reverse = 0;
66
67      if (spd < 0)
68      {
69        spd = -spd;
70        reverse = 1;
71      }
72      if (spd > 255)
73        spd = 255;
74
75      if (i == LEFT) {
76        if      (reverse == 0) { analogWrite(LEFT_MOTOR_FORWARD, spd);
    analogWrite(LEFT_MOTOR_BACKWARD, 0); }
77        else if (reverse == 1) { analogWrite(LEFT_MOTOR_BACKWARD, spd);
    analogWrite(LEFT_MOTOR_FORWARD, 0); }
78      }
```

```
79      else /*if (i == RIGHT) //no need for condition*/ {
80        if      (reverse == 0) { analogWrite(RIGHT_MOTOR_FORWARD, spd);
    analogWrite(RIGHT_MOTOR_BACKWARD, 0); }
81        else if (reverse == 1) { analogWrite(RIGHT_MOTOR_BACKWARD, spd);
    analogWrite(RIGHT_MOTOR_FORWARD, 0); }
82      }
83    }
84
85    void setMotorSpeeds(int leftSpeed, int rightSpeed) {
86      setMotorSpeed(LEFT, leftSpeed);
87      setMotorSpeed(RIGHT, rightSpeed);
88    }
89  #else
90    #error A motor driver must be selected!
91  #endif
92
93  #endif
94
```

1.2 Encoder Driver

```
#ifdef USE_BASE

#ifdef ROBOGAIA
  /* The Robogaia Mega Encoder shield */
  #include "MegaEncoderCounter.h"

  /* Create the encoder shield object */
  MegaEncoderCounter encoders = MegaEncoderCounter(4); // Initializes the Mega
Encoder Counter in the 4X Count mode

  /* Wrap the encoder reading function */
  long readEncoder(int i) {
    if (i == LEFT) return encoders.YAxisGetCount();
    else return encoders.XAxisGetCount();
  }

  /* Wrap the encoder reset function */
  void resetEncoder(int i) {
    if (i == LEFT) return encoders.YAxisReset();
    else return encoders.XAxisReset();
  }
#elif defined(ARDUINO_ENC_COUNTER)
```

```
  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)];
  }

  /* Interrupt routine for RIGHT encoder, taking care of actual counting */
  ISR (PCINT1_vect){
        static uint8_t enc_last=0;

  enc_last <<=2; //shift previous state two places
  enc_last |= (PINC & (3 << 4)) >> 4; //read the current state into lowest 2 bits

    right_enc_pos += ENC_STATES[(enc_last & 0x0f)];
  }

  /* Wrap the encoder reading function */
  long readEncoder(int i) {
    if (i == LEFT) return left_enc_pos;
    else return right_enc_pos;
  }

  /* Wrap the encoder reset function */
  void resetEncoder(int i) {
    if (i == LEFT){
      left_enc_pos=0L;
      return;
    } else {
      right_enc_pos=0L;
      return;
    }
  }
#else
  #error A encoder driver must be selected!
#endif
```

```
/* Wrap the encoder reset function */
void resetEncoders() {
  resetEncoder(LEFT);
  resetEncoder(RIGHT);
}

#endif
```

1.3 PID Controller

```
/* PID setpoint info For a Motor */
typedef struct {
  double TargetTicksPerFrame;    // target speed in ticks per frame
  long Encoder;                  // encoder count
  long PrevEnc;                  // last encoder count

  /*
  * Using previous input (PrevInput) instead of PrevError to avoid derivative
kick,
  * see http://brettbeauregard.com/blog/2011/04/improving-the-beginner%E2%80%99s-
pid-derivative-kick/
  */
  int PrevInput;                // last input
  //int PrevErr;                   // last error

  /*
  * Using integrated term (ITerm) instead of integrated error (Ierror),
  * to allow tuning changes,
  * see http://brettbeauregard.com/blog/2011/04/improving-the-beginner%E2%80%99s-
pid-tuning-changes/
  */
  //int Ierror;
  int ITerm;                    //integrated term

  long output;                    // last motor setting
}
SetPointInfo;

SetPointInfo leftPID, rightPID;
```

```
/* PID Parameters */
int Kp = 20;
int Kd = 12;
int Ki = 0;
int Ko = 50;

unsigned char moving = 0; // is the base in motion?

/*
* Initialize PID variables to zero to prevent startup spikes
* when turning PID on to start moving
* In particular, assign both Encoder and PrevEnc the current encoder value
* See http://brettbeauregard.com/blog/2011/04/improving-the-beginner%E2%80%99s-
pid-initialization/
* Note that the assumption here is that PID is only turned on
* when going from stop to moving, that's why we can init everything on zero.
*/
void resetPID(){
   leftPID.TargetTicksPerFrame = 0.0;
   leftPID.Encoder = readEncoder(LEFT);
   leftPID.PrevEnc = leftPID.Encoder;
   leftPID.output = 0;
   leftPID.PrevInput = 0;
   leftPID.ITerm = 0;

   rightPID.TargetTicksPerFrame = 0.0;
   rightPID.Encoder = readEncoder(RIGHT);
   rightPID.PrevEnc = rightPID.Encoder;
   rightPID.output = 0;
   rightPID.PrevInput = 0;
   rightPID.ITerm = 0;
}

/* PID routine to compute the next motor commands */
void doPID(SetPointInfo * p) {
  long Perror;
  long output;
  int input;

  //Perror = p->TargetTicksPerFrame - (p->Encoder - p->PrevEnc);
  input = p->Encoder - p->PrevEnc;
  Perror = p->TargetTicksPerFrame - input;
```

```
  /*
  * Avoid derivative kick and allow tuning changes,
  * see http://brettbeauregard.com/blog/2011/04/improving-the-beginner%E2%80%99s-
pid-derivative-kick/
  * see http://brettbeauregard.com/blog/2011/04/improving-the-beginner%E2%80%99s-
pid-tuning-changes/
  */
  //output = (Kp * Perror + Kd * (Perror - p->PrevErr) + Ki * p->Ierror) / Ko;
  // p->PrevErr = Perror;
  output = (Kp * Perror - Kd * (input - p->PrevInput) + p->ITerm) / Ko;
  p->PrevEnc = p->Encoder;

  output += p->output;
  // Accumulate Integral error *or* Limit output.
  // Stop accumulating when output saturates
  if (output >= MAX_PWM)
    output = MAX_PWM;
  else if (output <= -MAX_PWM)
    output = -MAX_PWM;
  else
  /*
  * allow turning changes, see http://brettbeauregard.com/blog/2011/04/improving-
the-beginner%E2%80%99s-pid-tuning-changes/
  */
    p->ITerm += Ki * Perror;

  p->output = output;
  p->PrevInput = input;
}

/* Read the encoder values and call the PID routine */
void updatePID() {
  /* Read the encoders */
  leftPID.Encoder = readEncoder(LEFT);
  rightPID.Encoder = readEncoder(RIGHT);

  /* If we're not moving there is nothing more to do */
  if (!moving){
    /*
    * Reset PIDs once, to prevent startup spikes,
    * see http://brettbeauregard.com/blog/2011/04/improving-the-
beginner%E2%80%99s-pid-initialization/
    * PrevInput is considered a good proxy to detect
    * whether reset has already happened
```

```
    */
    if (leftPID.PrevInput != 0 || rightPID.PrevInput != 0) resetPID();
    return;
  }

  /* Compute PID update for each motor */
  doPID(&rightPID);
  doPID(&leftPID);

  /* Set the motor speeds accordingly */
  setMotorSpeeds(leftPID.output, rightPID.output);
}
```

1.4 ROS-Arduino Bridge

```
#define USE_BASE      // Enable the base controller code
//#undef USE_BASE     // Disable the base controller code

/* Define the motor controller and encoder library you are using */
#ifdef USE_BASE
   /* The Pololu VNH5019 dual motor driver shield */
   //#define POLOLU_VNH5019

   /* The Pololu MC33926 dual motor driver shield */
   //#define POLOLU_MC33926

   /* The RoboGaia encoder shield */
   //#define ROBOGAIA

   /* Encoders directly attached to Arduino board */
   #define ARDUINO_ENC_COUNTER

   /* L298 Motor driver*/
   #define L298_MOTOR_DRIVER
#endif

//#define USE_SERVOS  // Enable use of PWM servos as defined in servos.h
#undef USE_SERVOS     // Disable use of PWM servos

/* Serial port baud rate */
```

```
#define BAUDRATE     57600

/* Maximum PWM signal */
#define MAX_PWM        255

#if defined(ARDUINO) && ARDUINO >= 100
#include "Arduino.h"
#else
#include "WProgram.h"
#endif

/* Include definition of serial commands */
#include "commands.h"

/* Sensor functions */
#include "sensors.h"

/* Include servo support if required */
#ifdef USE_SERVOS
   #include <Servo.h>
   #include "servos.h"
#endif

#ifdef USE_BASE
  /* Motor driver function definitions */
  #include "motor_driver.h"

  /* Encoder driver function definitions */
  #include "encoder_driver.h"

  /* PID parameters and functions */
  #include "diff_controller.h"

  /* Run the PID loop at 30 times per second */
  #define PID_RATE           30     // Hz

  /* Convert the rate into an interval */
  const int PID_INTERVAL = 1000 / PID_RATE;

  /* Track the next time we make a PID calculation */
  unsigned long nextPID = PID_INTERVAL;

  /* Stop the robot if it hasn't received a movement command
   in this number of milliseconds */
```

```
  #define AUTO_STOP_INTERVAL 2000
  long lastMotorCommand = AUTO_STOP_INTERVAL;
#endif

/* Variable initialization */

// A pair of varibles to help parse serial commands (thanks Fergs)
int arg = 0;
int index = 0;

// Variable to hold an input character
char chr;

// Variable to hold the current single-character command
char cmd;

// Character arrays to hold the first and second arguments
char argv1[16];
char argv2[16];

// The arguments converted to integers
long arg1;
long arg2;

/* Clear the current command parameters */
void resetCommand() {
  cmd = NULL;
  memset(argv1, 0, sizeof(argv1));
  memset(argv2, 0, sizeof(argv2));
  arg1 = 0;
  arg2 = 0;
  arg = 0;
  index = 0;
}

/* Run a command.  Commands are defined in commands.h */
int runCommand() {
  int i = 0;
  char *p = argv1;
  char *str;
  int pid_args[4];
  arg1 = atoi(argv1);
  arg2 = atoi(argv2);
```

```
  switch(cmd) {
  case GET_BAUDRATE:
    Serial.println(BAUDRATE);
    break;
  case ANALOG_READ:
    Serial.println(analogRead(arg1));
    break;
  case DIGITAL_READ:
    Serial.println(digitalRead(arg1));
    break;
  case ANALOG_WRITE:
    analogWrite(arg1, arg2);
    Serial.println("OK");
    break;
  case DIGITAL_WRITE:
    if (arg2 == 0) digitalWrite(arg1, LOW);
    else if (arg2 == 1) digitalWrite(arg1, HIGH);
    Serial.println("OK");
    break;
  case PIN_MODE:
    if (arg2 == 0) pinMode(arg1, INPUT);
    else if (arg2 == 1) pinMode(arg1, OUTPUT);
    Serial.println("OK");
    break;
  case PING:
    Serial.println(Ping(arg1));
    break;
#ifdef USE_SERVOS
  case SERVO_WRITE:
    servos[arg1].setTargetPosition(arg2);
    Serial.println("OK");
    break;
  case SERVO_READ:
    Serial.println(servos[arg1].getServo().read());
    break;
#endif

#ifdef USE_BASE
  case READ_ENCODERS:
    Serial.print(readEncoder(LEFT));
    Serial.print(" ");
    Serial.println(readEncoder(RIGHT));
    break;
   case RESET_ENCODERS:
```

```
    resetEncoders();
    resetPID();
    Serial.println("OK");
    break;
  case MOTOR_SPEEDS:
    /* Reset the auto stop timer */
    lastMotorCommand = millis();
    if (arg1 == 0 && arg2 == 0) {
      setMotorSpeeds(0, 0);
      resetPID();
      moving = 0;
    }
    else moving = 1;
    leftPID.TargetTicksPerFrame = arg1;
    rightPID.TargetTicksPerFrame = arg2;
    Serial.println("OK");
    break;
  case MOTOR_RAW_PWM:
    /* Reset the auto stop timer */
    lastMotorCommand = millis();
    resetPID();
    moving = 0; // Sneaky way to temporarily disable the PID
    setMotorSpeeds(arg1, arg2);
    Serial.println("OK");
    break;
  case UPDATE_PID:
    while ((str = strtok_r(p, ":", &p)) != '\0') {
       pid_args[i] = atoi(str);
       i++;
    }
    Kp = pid_args[0];
    Kd = pid_args[1];
    Ki = pid_args[2];
    Ko = pid_args[3];
    Serial.println("OK");
    break;
#endif
  default:
    Serial.println("Invalid Command");
    break;
  }
}

/* Setup function--runs once at startup. */
```

```
void setup() {
  Serial.begin(BAUDRATE);

// Initialize the motor controller if used */
#ifdef USE_BASE
  #ifdef ARDUINO_ENC_COUNTER
    //set as inputs
    DDRD &= ~(1<<LEFT_ENC_PIN_A);
    DDRD &= ~(1<<LEFT_ENC_PIN_B);
    DDRC &= ~(1<<RIGHT_ENC_PIN_A);
    DDRC &= ~(1<<RIGHT_ENC_PIN_B);

    //enable pull up resistors
    PORTD |= (1<<LEFT_ENC_PIN_A);
    PORTD |= (1<<LEFT_ENC_PIN_B);
    PORTC |= (1<<RIGHT_ENC_PIN_A);
    PORTC |= (1<<RIGHT_ENC_PIN_B);

    // tell pin change mask to listen to left encoder pins
    PCMSK2 |= (1 << LEFT_ENC_PIN_A)|(1 << LEFT_ENC_PIN_B);
    // tell pin change mask to listen to right encoder pins
    PCMSK1 |= (1 << RIGHT_ENC_PIN_A)|(1 << RIGHT_ENC_PIN_B);

    // enable PCINT1 and PCINT2 interrupt in the general interrupt mask
    PCICR |= (1 << PCIE1) | (1 << PCIE2);
  #endif
  initMotorController();
  resetPID();
#endif

/* Attach servos if used */
  #ifdef USE_SERVOS
    int i;
    for (i = 0; i < N_SERVOS; i++) {
      servos[i].initServo(
          servoPins[i],
          stepDelay[i],
          servoInitPosition[i]);
    }
  #endif
}

/* Enter the main loop.  Read and parse input from the serial port
   and run any valid commands. Run a PID calculation at the target
```

```
   interval and check for auto-stop conditions.
*/
void loop() {
  while (Serial.available() > 0) {

    // Read the next character
    chr = Serial.read();

    // Terminate a command with a CR
    if (chr == 13) {
      if (arg == 1) argv1[index] = NULL;
      else if (arg == 2) argv2[index] = NULL;
      runCommand();
      resetCommand();
    }
    // Use spaces to delimit parts of the command
    else if (chr == ' ') {
      // Step through the arguments
      if (arg == 0) arg = 1;
      else if (arg == 1)  {
        argv1[index] = NULL;
        arg = 2;
        index = 0;
      }
      continue;
    }
    else {
      if (arg == 0) {
        // The first arg is the single-letter command
        cmd = chr;
      }
      else if (arg == 1) {
        // Subsequent arguments can be more than one character
        argv1[index] = chr;
        index++;
      }
      else if (arg == 2) {
        argv2[index] = chr;
        index++;
      }
    }
  }
```

```
// If we are using base control, run a PID calculation at the appropriate
intervals
#ifdef USE_BASE
  if (millis() > nextPID) {
    updatePID();
    nextPID += PID_INTERVAL;
  }

  // Check to see if we have exceeded the auto-stop interval
  if ((millis() - lastMotorCommand) > AUTO_STOP_INTERVAL) {;
    setMotorSpeeds(0, 0);
    moving = 0;
  }
#endif

// Sweep servos
#ifdef USE_SERVOS
  int i;
  for (i = 0; i < N_SERVOS; i++) {
    servos[i].doSweep();
  }
#endif
}
```

2. Robot Files

2.1 ros2 control.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <xacro:unless value="$(arg sim_mode)">
        <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/ttyACM0</param>
                <param name="baud_rate">57600</param>
                <param name="timeout">1000</param>
                <param name="enc_counts_per_rev">1910</param>
            </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="velocity"/>
                <state_interface name="position"/>
            </joint>
            <joint name="right_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10</param>
                    <param name="max">10</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
        </ros2_control>
    </xacro:unless>

    <xacro:if value="$(arg sim_mode)">
        <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="velocity"/>
                <state_interface name="position"/>
            </joint>
            <joint name="right_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10</param>
                    <param name="max">10</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
        </ros2_control>
    </xacro:if>

    <gazebo>
        <plugin name="gazebo_ros2_control" filename="libgazebo_ros2_control.so">
            <parameters>$(find
hero_robot)/config/my_controllers.yaml</parameters>
            <parameters>$(find
hero_robot)/config/gaz_ros2_ctl_use_sim.yaml</parameters>
        </plugin>
    </gazebo>

</robot>
```

2.2 rsp.launch.py

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.substitutions import LaunchConfiguration, Command
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node

import xacro
```

```
def generate_launch_description():

    # Check if we're told to use sim time
    use_sim_time = LaunchConfiguration('use_sim_time')
    use_ros2_control = LaunchConfiguration('use_ros2_control')

    # Process the URDF file
    pkg_path = os.path.join(get_package_share_directory('hero_robot'))
    xacro_file = os.path.join(pkg_path,'description','robot.urdf.xacro')
    # robot_description_config = xacro.process_file(xacro_file).toxml()
    robot_description_config = Command(['xacro ', xacro_file, '
use_ros2_control:=', use_ros2_control, ' sim_mode:=', use_sim_time])

    # Create a robot_state_publisher node
    params = {'robot_description': robot_description_config, 'use_sim_time':
use_sim_time}
    node_robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        parameters=[params]
    )


    # Launch!
    return LaunchDescription([
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use sim time if true'),
        DeclareLaunchArgument(
            'use_ros2_control',
            default_value='true',
            description='Use ros2_control if true'),

        node_robot_state_publisher
    ])
```

2.3 launch_robot.launch.py

```
import os

from ament_index_python.packages import get_package_share_directory


from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription, TimerAction
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import Command
from launch.actions import RegisterEventHandler
from launch.event_handlers import OnProcessStart
from launch.substitutions import LaunchConfiguration
from launch.conditions import IfCondition
from launch_ros.actions import Node



def generate_launch_description():




    package_name='hero_robot'

    rsp = IncludeLaunchDescription(
                PythonLaunchDescriptionSource([os.path.join(

get_package_share_directory(package_name),'launch','rsp.launch.py'
                )]), launch_arguments={'use_sim_time': 'false',
'use_ros2_control': 'true'}.items()
    )


    twist_mux_params =
os.path.join(get_package_share_directory(package_name),'config','twist_mux.yaml')
    twist_mux = Node(
            package="twist_mux",
            executable="twist_mux",
            parameters=[twist_mux_params],
            remappings=[('/cmd_vel_out','/diff_cont/cmd_vel_unstamped')]
        )
```

```
    robot_description = Command(['ros2 param get --hide-type
/robot_state_publisher robot_description'])

    controller_params_file =
os.path.join(get_package_share_directory(package_name),'config','my_controllers.y
aml')

    controller_manager = Node(
        package="controller_manager",
        executable="ros2_control_node",
        parameters=[{'robot_description': robot_description},
                    controller_params_file]
    )

    delayed_controller_manager = TimerAction(period=3.0,
actions=[controller_manager])

    diff_drive_spawner = Node(
        package="controller_manager",
        executable="spawner.py",
        arguments=["diff_cont"],
    )

    delayed_diff_drive_spawner = RegisterEventHandler(
        event_handler=OnProcessStart(
            target_action=controller_manager,
            on_start=[diff_drive_spawner],
        )
    )

    joint_broad_spawner = Node(
        package="controller_manager",
        executable="spawner.py",
        arguments=["joint_broad"],
    )

    delayed_joint_broad_spawner = RegisterEventHandler(
        event_handler=OnProcessStart(
            target_action=controller_manager,
            on_start=[joint_broad_spawner],
        )
    )
```

```
    rviz_config_file = os.path.join(get_package_share_directory(package_name),
"rviz", "view_bot.rviz")
    use_rviz = LaunchConfiguration("rviz", default=False)
    rviz = Node(
        package= "rviz2",
        executable= "rviz2",
        arguments=["-d", rviz_config_file],
        output= "screen",
        condition=IfCondition(use_rviz)
    )

    lidar = Node(
            package='rplidar_ros',
            node_executable='rplidar_composition',
            node_name='rplidar_composition',
            output='screen',
            parameters=[{
                'serial_port': '/dev/ttyUSB0',
                'serial_baudrate': 115200,  # A1 / A2
                # 'serial_baudrate': 256000, # A3
                'frame_id': 'laser_frame',
                'inverted': False,
                'angle_compensate': True,}])



    # Launch them all!
    return LaunchDescription([
        rsp,
        twist_mux,
        delayed_controller_manager,
        delayed_diff_drive_spawner,
        delayed_joint_broad_spawner,
        lidar,
        rviz,
    ])
```