An-Najah National University

Faculty of Engineering & Information Technology

Presented in partial fulfillment of the requirements

for a bachelor’s degree in computer engineering

RoboMealMate: Innovating Restaurant Service Hospitality with AI (Artificial Intelligence)

Students:

Osama Mansour

Ahmad Rasheed

Supervisors:

Dr.Bahaa Shaqour

Dr.Muhaned Al-jabi

22 January 2025

Acknowledgment

We would like to express our sincere gratitude to Dr. Bahaa Shaqour and Dr. Muhaned Al-jabi, our supervisors, for their invaluable guidance, continuous support, and insightful feedback throughout the development of this project. Their expertise and encouragement have been instrumental in the successful completion of RoboMealMate.

We extend our appreciation to An-Najah National University, Faculty of Engineering & Information Technology, for providing us with the resources and knowledge that made this project possible.

A special thank you to our families and friends for their unwavering support, patience, and motivation during this journey.

Lastly, we acknowledge all the researchers and developers whose work in robotics, artificial intelligence, and automation has inspired and contributed to the realization of our project.

Disclaimer

DISCLAIMER

This report was written by student(s) at the Engineering Department, Faculty of Engineering, An-Najah National University. It has not been altered or corrected, other than editorial corrections, as a result of assessment and it may contain language as well as content errors. The views expressed in it together with any outcomes and recommendations are solely those of the student(s). An-Najah National University accepts no responsibility or liability for the consequences of this report being used for a purpose other than the purpose for which it was commissioned.

Contents

List of Figures

List of tables

Abstract

The RoboMealMate project is designed to harness the technological revolution in artificial intelligence and robotics in restaurant automation, customer service, and smart hospitality. The project aims to enhance customer service efficiency, reduce operational costs, and provide customers with a unique experience. The robot will serve as a waiter. It will navigate autonomously in restaurant environments, interacting with customers, taking orders, and conversing with them.

- **Importance aspects to cover:**
    1. **Robot interaction**: implement a friendly design that seamlessly interacts with customers the design will include a screen for signs and emotions and a microphone for speech recognition.
    2. **Navigation and Obstacle Avoidance:** implement LiDAR-based SLAM (Simultaneous Localization and Mapping) and image processing for accurate movement and obstacle avoidance.
    3. **AI:** using speech recognition, order processing, and image-based learning to enhance adaptability and add human-like behavior.
    4. **Safety and reliability:** ensure that the robot is secure for restaurant environments, especially crowded environments by choosing the right components, sensors, and design that is compatible with the work environment.
- **Objectives:**
    1. Enhance customer service.
    2. Increase productivity and accuracy in the work environment.
    3. Reduce operational costs.
    4. Harnessing of AI and robotics revolution in serving people and RoboMealMate is just the beginning.
- Methodology:

1. **Hardware process:**
   Create a design for the robot body, then set up and connect the components, such as the Raspberry Pi, Arduino Mega, sensors(LiDAR, ultrasonic, etc.), camera, Screen, motors, and power supplies. Make sure that all components are connected correctly.
2. **Software process:**
   In this process, we will implement the algorithms for AI parts like OpenAI API, speech-to-text and text-to-speech APIs, image processing, LiDAR-base SLAM, ...etc.
3. **Testing and debugging process:** Test the robot in different situations, ensure that the functionalities are working correctly, and handle the errors.

- **Similar Projects:**
  While other restaurant robots exist, such as Pepper and Bear Robotics' Servi, RoboMealMate aims to stand out by incorporating advanced AI for interactive communication, adaptable learning, and seamless integration in small to medium-sized restaurant spaces.

Chapter 1

1 Introduction

1.1 Background:

When we talk about restaurants, customer satisfaction is one of the most important elements for the success of restaurants, as customer satisfaction means more sales, more sales means more profit, and more profit means achieving financial success. But there are many problems facing the restaurant sector, especially in the field of customer service.

One of the main problems in restaurants is the problem of taking orders, taking orders problem manifests in several ways, common issues are miscommunication between staff and customers manual entry errors slow service, inflexibility, and lack of personalization.

The increasing demand for autonomous servicing in restaurants where efficiency, accuracy, and customer experience are paramount, leads to a gap in how a machine or robot achieves these goals without humans. Before the AI revolution that was hard to accomplish, but these days with AI tools available around us and easy access to the network the mission has become possible.

RoboMealMate came to solve most of the taking orders issues, it is an AI-driven robot that navigates autonomy in a restaurant environment engages with customers takes orders, and makes conversations with them, then places the orders for the kitchen.

1.2 Objectives:

RoboMealMate aims to add a distinctive experience to customers through their interaction with a human-like machine that meets their needs with the required speed and accuracy. It also aims to solve problems related to taking orders and making customer service more accurate and fast in the restaurant environment. Harnessing technological developments in the field of AI and robotics in customer service, design a cost-effective and scalable solution that can be deployed in diverse restaurant environments.

1.3 Importance and Significance:

The importance of RoboMealMate lies in meeting the global market’s demands for self-service and harnessing technology in business. The field of restaurant automation is expected to grow significantly as companies strive to achieve efficiency and customer satisfaction.

Inaccurate ordering, slow service, and excessive labor expenses are just a few of the inefficiencies that plague traditional restaurant operations. In order to solve these problems, RoboMealMate offers a dependable and automated solution that raises operational consistency and service quality. Additionally, the robot's capacity to connect with patrons through individualized interactions and natural language processing improves the eating experience and distinguishes it from conventional service techniques.

Additionally, RoboMealMate is an advancement in scalable and sustainable technology. Its design may be modified to fit various restaurant settings, giving it a flexible option for businesses with varying clientele and sizes. RoboMealMate not only satisfies present market demands but also establishes itself as a pioneer in restaurant technology by satisfying the increasing demand for intelligent automation.

1.4 Report Overview:

This report provides a comprehensive overview of the RoboMealMat**e** project, detailing the design, development, and implementation of an AI-driven robotic waiter for restaurant environments. The document is structured as follows:

- Chapter 1: Introduction Introduces the project, its objectives, and significance in the restaurant industry.
- Chapter 2: Theoretical Background and Previous Work – Reviews relevant technologies, similar projects, and advancements in AI and robotics.
- Chapter 3: Methodology Covers the physical and hardware design, software architecture, simulation process, ROS2 integration, and AI components.
- Chapter 4: Implementation & Testing Details the integration of AI and machine learning for real-time customer interaction, speech recognition, and autonomous navigation.
- Chapter 5: Results & Discussion Presents testing results, challenges encountered, and potential improvements.
- Chapter 6: Conclusion & Future Work Summarizes findings and suggests future enhancement

Chapter 2

2 Theoretical Background and Previous Work

This chapter provides an overview of the theoretical concepts and previous research that underpin the RoboMealMate project. It aims to establish a foundation for understanding the key technologies used in the project and to highlight relevant work in the fields of robotics, artificial intelligence (AI), computer vision, natural language processing (NLP),and autonomous navigation.

2.1 Robotic Service Systems in Hospitality

Robotics in the hospitality sector has garnered significant interest in recent years, particularly in food service. Robots are increasingly used to automate repetitive tasks, improve customer experience, and enhance operational efficiency. In restaurants, robots are used for a variety of functions, such as serving food, greeting customers, and even preparing meals. These systems leverage technologies like computer vision, machine learning, speech recognition, and autonomous navigation.

One notable example is the Pepper robot, developed by SoftBank Robotics, which has been used in multiple industries for customer service applications, including restaurants. Pepper uses facial recognition and natural language processing to interact with people, offering a friendly and intuitive experience. However, despite these advancements, many existing robotic systems still face challenges in terms of mobility, adaptability to dynamic environments, and the complexity of human-robot interactions.

2.2 Key Technologies in RoboMealMate

2.2.1 Artificial Intelligence (AI) in Robotics

AI plays a pivotal role in modern robotics, enabling machines to learn from their environment, make decisions, and improve performance over time. In the RoboMealMate project, AI technologies are used to enhance the robot's ability to interact with humans, understand commands, and make autonomous decisions.

2.2.2 Natural Language Processing (NLP)

NLP is a crucial component for enabling effective human-robot interaction in the RoboMealMate project. NLP algorithms are used to interpret voice commands and process spoken language, allowing the robot to take orders from customers and respond accordingly. Technologies like Google Cloud Speech-to-Text and OpenAI GPT are used to transcribe speech into text and generate natural language responses, respectively. These tools enhance the robot's ability to understand and interact in a conversational manner, making the user experience more intuitive and engaging.

Incorporating NLP into the system allows for multilingual capabilities, enabling the robot to serve a broader customer base by processing commands in different languages. This feature is particularly valuable in globalized restaurant environments where diverse language speakers interact with the service system.

2.2.3 SLAM(Simultaneous Localization and Mapping)

The robot uses SLAM (Simultaneous Localization and Mapping) techniques to navigate its environment autonomously. SLAM allows the robot to build a map of its surroundings while simultaneously determining its position within that map, which is essential for safe and efficient movement in dynamic environments, such as busy restaurant floors. The use of Slamtec RPLIDAR A1M8 provides high-precision scanning capabilities for this task.

2.2.4 Autonomous Navigation and Obstacle Avoidance

Autonomous navigation is central to the functionality of the RoboMealMate robot. The robot is equipped with ultrasonic sensors, and LiDAR, to detect obstacles and move through its environment efficiently. Path planning and obstacle avoidance are achieved through sophisticated algorithms that integrate data from these sensors to determine the best routes and avoid collisions with both static and dynamic obstacles.

The integration of omni wheels in the robot's design allows it to move in any direction, providing greater flexibility in movement and improving its ability to navigate tight spaces in crowded restaurant environments.

2.3 Previous Work in Service Robotics

Several studies have explored the use of robots in service-oriented environments, including restaurants. For example, Savioke's Relay robot is designed to deliver items in hotels, while Bear Robotics' Servi robot is employed in restaurants to serve food to customers autonomously. These robots rely on AI, computer vision, and autonomous navigation to perform their tasks. However, there are still gaps in handling more complex interactions, such as understanding diverse customer preferences, dynamically adjusting to busy environments, and providing a personalized experience.

Another notable example is the Karakuri robot, which specializes in personalized food service. It combines robotics with AI to create dishes according to customer preferences, enhancing the dining experience. This type of service personalization, along with real-time adaptive behavior, is an area of focus in the RoboMealMate project.

2.4 Challenges and Opportunities

The integration of multiple advanced technologies into the RoboMealMate project introduces both challenges and opportunities for future research and development. One of the primary challenges is the dynamic nature of restaurant environments. Human movement, changing obstacles, and unpredictable interactions create difficulties in navigation and object recognition. Additionally, while current NLP technologies are impressive, they still struggle with accents, noisy environments, and understanding complex or ambiguous speech.

Another challenge is the robot's ability to understand and react to social cues, such as body language or emotional states, in a way that feels natural to human users. Research into emotion-aware robotics and personalized service delivery continues to evolve, and the

RoboMealMate project aims to push the boundaries in this area by incorporating more sophisticated emotion recognition and response systems.

On the other hand, these challenges present exciting opportunities for innovation. The RoboMealMate project stands at the intersection of AI, robotics, and human-computer interaction, providing a platform for further advancements in service robots, customer experience, and AI-driven automation.

Chapter 3

3 Methodology

In this chapter, we will provide detailed steps we have done to accomplish this work

3.1 Design:

This section is divided into 3 subsections as follows

I. Physical design: in this subsection, we will talk about the shape, dimensions, material, and parts.
II. Hardware design: In this subsection, we will discuss hardware components, their uses, and their costs.
III. Software design: This subsection will discuss the software approach, ROS2(robot operating system), ROS2 topics, ROS2 nodes, ROS2 packages & plugins.

3.1.1 physical design:

When we say the robot will serve as a waiter We must take into consideration that the shape, dimensions, and design must be consistent with the task it will perform, so the shape and dimensions should be human-like, we achieved this by following:

A. Base: It will be the main part to which the rest of the body will be attached and will be primarily responsible for movement and holding hardware components for movement.
   - ❖ Dimensions:
     - ✓ Height: 15 cm.
     - ✓ Width: 60 cm.
     - ✓ Length: 30 cm.

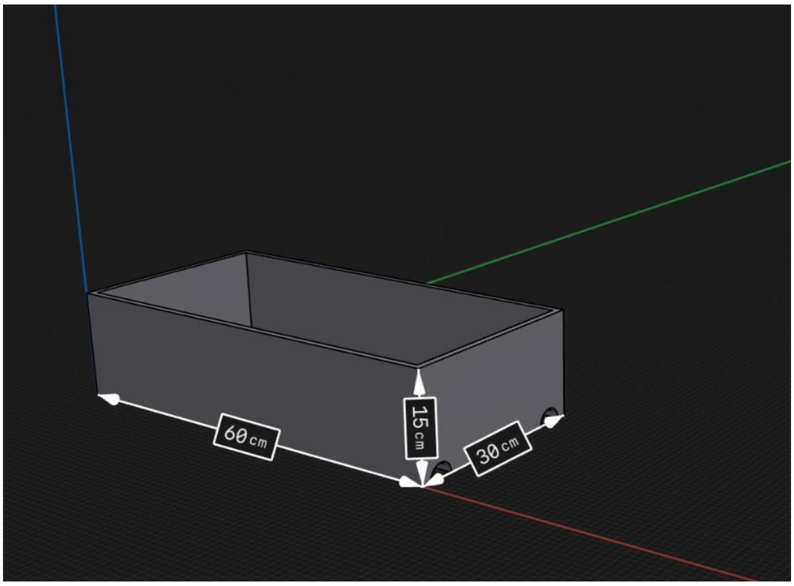


Figure 3.1Base dimensions

B. Left and right feet: it will be the middle part between the base and body and will be a wire tunnel between the base and the body.
  - ❖ Dimensions:
    - ✓ Height: 50 cm.
    - ✓ Width: 15 cm.
    - ✓ Length: 15 cm.
    - ✓ Distance between: 10 cm

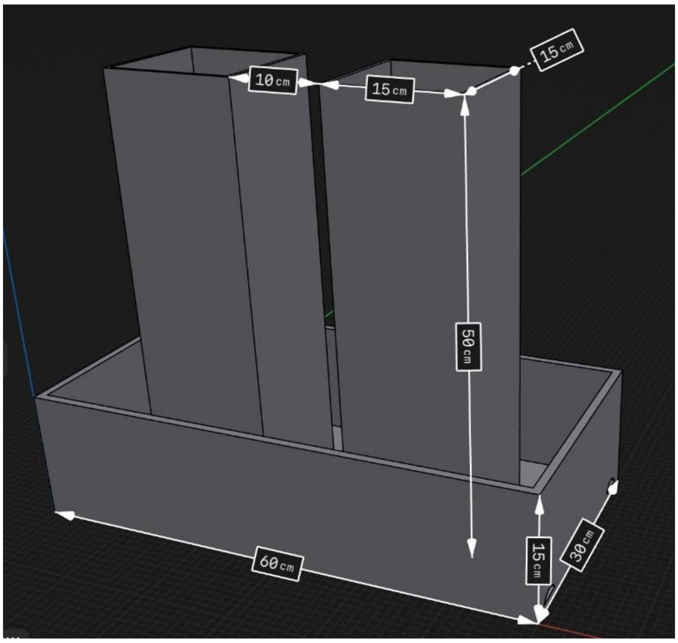


Figure 3.2Feet Dimensions

C. Body: it will attach to feet, it contains batteries, a low-level controller, and speakers.
  - ❖ Dimensions:
    - ✓ Height: 40 cm.
    - ✓ Width: 40 cm.
    - ✓ Length: 15cm

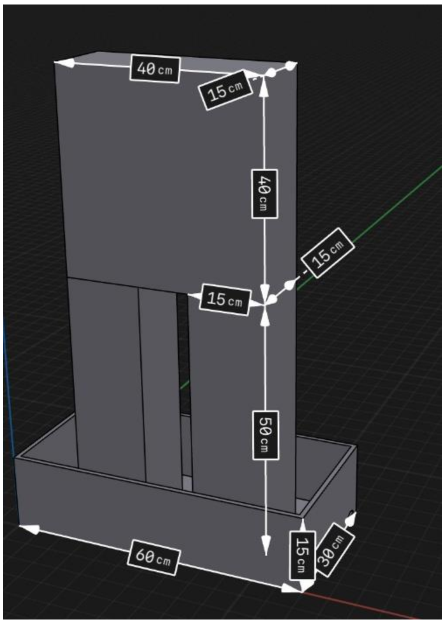


Figure 3.3Body Dimensions

D. Head: it will attach to the body, it contains a screen, camera, and high-level processor.
   - ❖ Dimensions:
     - ✓ Height: neck 10 cm, head 20 cm.
     - ✓ Width: neck 15 cm, head 30 cm.
     - ✓ Length: 15 cm.

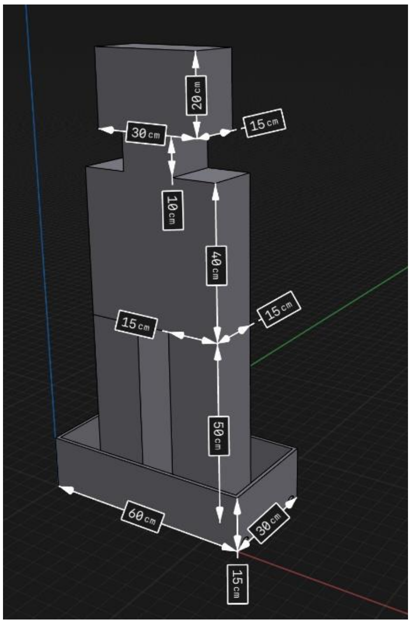


Figure 3.4Head and Robot Dimensions

So the total dimensions:

- Highet: 125 cm.
- Width: 60 cm.
- Length: 30 cm.

The material of the robot will be light wood, for lighter weight and a cohesive body.

3.1.2 Hardware design:

3.1.2.1 Hardware components:

Below are the hardware components used in the project and the responsibility for each one:

1. Arduino Mega 2560:

Figure 3.5Arduino Mega 2560

Its low-level controller is responsible for taking sensor readings sending them to the high-level controller via serial communication, and generating PWM signals for motor drivers depending on high-level controller orders.

2. Raspberry Pi 4:

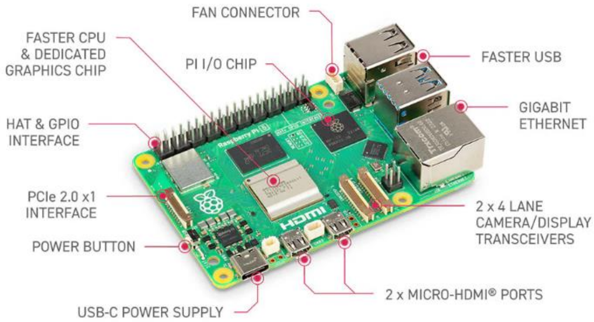


Figure 3.6Raspberry PI4

Its high-level controller is responsible for decision-making, gives orders to the low-level controller via serial communication, and is responsible for GPT, voice recognition scripts, and the ROS2 environment, we can say it’s the brain of our robot.

3. Ultrasonic sensors (HC-SR04):

Figure 3.7Ultrasonic Sensor(HC-SR04)

It uses ultrasonic sound waves to detect the distance to an object, it is responsible for detecting near and short-distance obstacles.

4. RPLIDAR A1 360:

Figure 3.8RPLidar A1

RPLIDAR A1 is based on the laser triangulation ranging principle and uses high-speed vision acquisition. It mainly generates a local map using SLAM (simultaneous localization and mapping) algorithms. This will allow our robot to know the environment around him and detect obstacles. It sends its data readings to Raspberry via serial.

5. Camera:

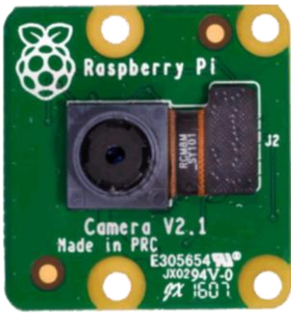


Figure 3.9Raspberry camera

It is mainly used to monitor the robot environment.

6. Motors (jgb37-520):

Figure 3.10DC-Motor

Its 12V DC motor with gearbox is responsible for robot movement.

7. Encoders:

Figure 3.11DC Motor Encoder

It provides feedback about the motor's position, speed, and direction of rotation., and it's used in closed-loop control.

8. Batteries:
   - ✓ 12v 7Ahm: to power the motors.
   - ✓ 2 * 3.7 V 2600 mAhm: to power Arduino mega and sensors.
   - ✓ 10000 mAhm power bank to power Raspberry.

9. Motor Drivers (IBT 2):

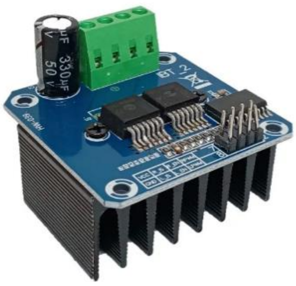


Figure 3.12IBT 2 Motor driver

It used for control DC motors, direction control, speed control, and high current and voltage handling.

10. Buck converter (LM2596 DC-DC):

Figure 3.13LM2596 DC-DC

It is used to reduce the input voltage to a lower, stable output voltage.

11. Display:

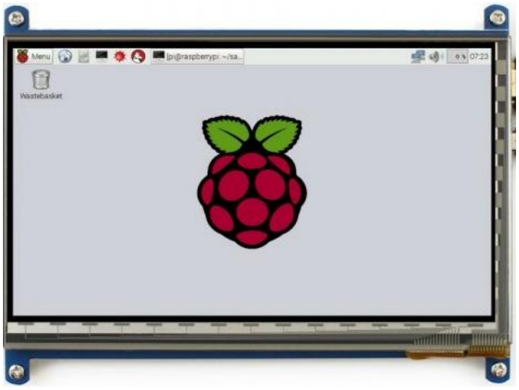


Figure 3.14LCD 7inch display

It will be used for monitoring the robot's face or interactions.

12. Other components:
    - ✓ Wheels.
    - ✓ Wiers.
    - ✓ RGB led strip.
13. Speakers:

Figure 3.15Speakers

It is used for sound output.

14. Microphone:

Figure 3.16microphone

It is used for voice input and voice recognition.

3.1.2.2 Quantities and costs:

Component name	Quantity	Price($)	subtotal
Arduino Mega 2560	1	30	30
Raspberry Pi 4	1	110	110
Ultrasonic sensors (HC-SR04)	4	5	20
RPLIDAR A1	1	120	120
Raspberry camera v2	1	45	45
jgb37-520 motors & EN	4	25	100
12v battery	1	22	22
3.7 lio-into battery	2	5	10
Power bank	1	22	22
Motor Drivers (IBT 2)	4	15	60
LM2596 DC-DC	2	5	10
Display	1	100	100
Wheels	4	15	60
Wires		30	30
wood		60	60
Buttons	4	5	20
speakers	2	10	20
microphone	1	10	10
RGB strip	1	10	10
Total	35		**859**

Table 3.1cost table

3.1.3 Software design:

When we think about our robot it is very complex and has a set of complex operations, so when we think about the software running everything together it should be flexible and powerful enough to manage all the processes, To achieve this goal we will go with ROS2 (Robot Operating System).

ROS2 has large features and the following are the most important:

a) Improved communication middleware.
b) Real-time capabilities.
c) Cross-platform compatibility.
d) Support for Microcontrollers.
e) Time Synchronization.

So ROS is for high-level controlling and decision-making, and Arduino C code is for low-level controlling, and there is a bridge between low-level controlling and high-level controlling to send and receive data between them.

3.1.3.1 ROS2 Overview:

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications. In the following, we will review the most important concepts and terminology so that we can understand the mechanism of the upcoming work:

- **Nodes**: is a basic building block, each node is a standalone process responsible for a specific functionality, for example controlling motors.
- **Topics:** it is the way for communication between nodes via the publish-subscribe model, a node publishes the data on a topic and others subscribe to the topic to receive the data.
- **Messages**: it is the text, sensor data, numbers, or more complex information that the node can exchange with others.
- **Services**: allowing nodes to send a request and receive a response.
- **Actions**: support long-running tasks, providing feedback and result messages during execution.
- **Parameter Server:** it is a storage system containing the configurations that nodes can access and modify at runtime.
- **Packages**: in ROS codes are organized in packages each package contains libraries, configuration files, and more.
- **Launch Files:** XML or YAML-based files used to start multiple nodes and configure parameters simultaneously.

3.1.3.2 Arduino:

We will use the Arduino code and libraries for low-level controlling, C language provides a set of libraries that can handle sensor reading, motor controlling, and more.

3.2 Simulation:

Before we dive into the process I will explain two important tools provided by ROS for simulation purposes:

a) Rviz2 (Robot Visualization 2): It provides a graphical interface for users to view their robot, sensor data, maps, and more, it is a next-generation visualization tool for ROS2, We will learn more about its features during work.
b) GAZEBO: It provides a robust platform for simulating robots in complex, real-world-like environments, allowing developers to test and refine their robotic applications without needing physical hardware.

In the following, we will dive into the simulation process which is divided into a set of steps we will explain everything as we go

3.2.1 URDF (Unified Robot Description Format):

It is a file that tells ROS about the structure of our robot and how it's all set up, It serves as a standard way to represent a robot model, making it easy to share, simulate, and integrate with ROS tools and libraries.

First, we should understand how these files work, XML based files use a tool called XACRO, XACRO combines a set of files in a single complete URDF, passes the URDF file to a node called robot_state_publisher publishes the robot TF transforms, the data will be available on a topic called /robot_description, if there are any joints in the URDF it expect input values published to /joint_states topic, and as a test, we will use joint_state_publisher_gui node for input values, in the following is a diagram explain the process.

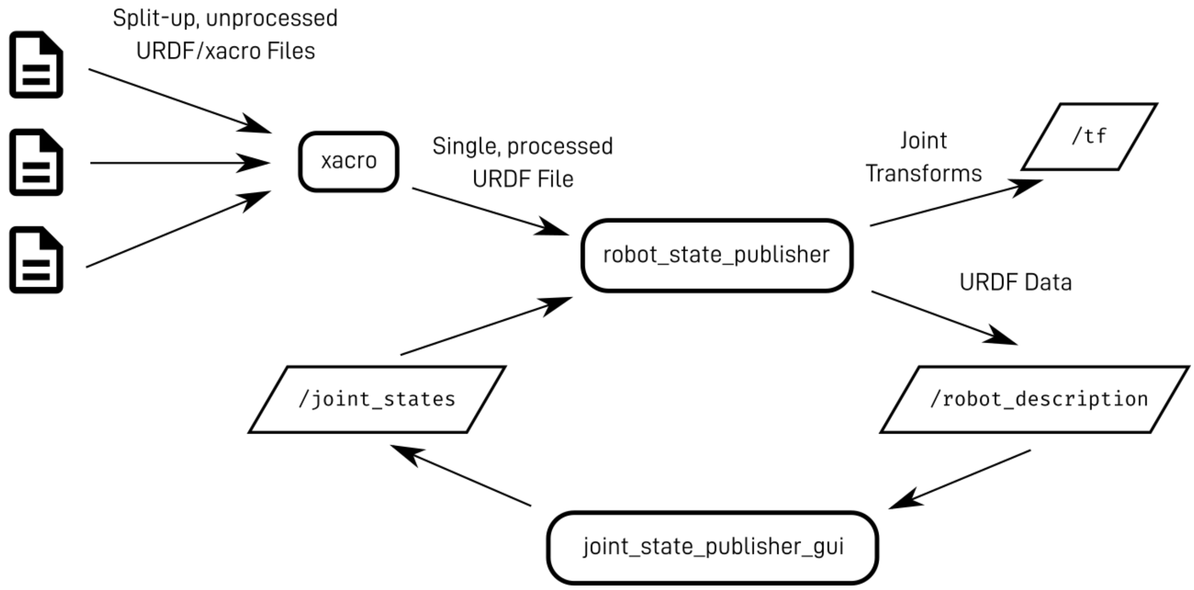


Figure 3.17Robot state publisher and URDF diagram

Our URDFs are split into 4 files:

1. robot.urdf.xacro: this file contains all xacro files and it's the one we will send to robot_state_publisher.
2. robot_core.xacro: this file contains the main components of the robot wheels, base, feet, body, and head.
3. lidar.xacro: this file contains the description of the lidar and later its driver.
4. camera.xacro: this file contains the description of the camera and later its driver.

3.2.1.1 robot_core.xacro file:

In this file, we will describe the main components of our robot to show the full code of this file CTRL+click on the file name.

1. Base:

```
<link name="base_link">
  <visual>
    <origin xyz="0.15 0.0 0.075"/>
    <geometry>
      <box size="0.30 0.60 0.15"/>
    </geometry>
    <material name="Blue"/>
  </visual>
  <collision>
    <origin xyz="0.15 0.0 0.075"/>
    <geometry>
      <box size="0.30 0.60 0.15"/>
    </geometry>
  </collision>
</link>
```

The link tag represents a part of the robot this part is called base_link, visual tag describes this part by specifying its origin, geometry, and material, so our base center is at point (0.15, 0.0,0.075), has a box shape with dimensions 30 cm, 60 cm, 15 cm, and its color is blue and the collision tag is the same as visual tag and does not have a joint because it's the root link.

2. wheels:
   - ❖ Link:

```
<link name="front_left_wheel">
    <visual>
      <geometry>
        <cylinder radius="0.035" length="0.04"/>
      </geometry>
      <material name="Black"/>
    </visual>
    <collision>
      <geometry>
        <sphere radius="0.035"/>
      </geometry>
    </collision>
 </link>
```

Our link is called front_left_wheel it's a cylindrical shape with a 3.5 cm radius and 4 cm length.

   - ❖ Joint:

```
  <joint name="front_left_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="front_left_wheel"/>
    <origin xyz="0.25 0.32 0.0" rpy="-${pi/2} 0.0 0.0"/>
    <axis xyz="0.0 0.0 1.0"/>
  </joint>
```

The joint tag describes how is the parent of the child is linked and in what origins connected to it, type of this joint continuous or fixed, continuous means the joint is rotating around an axis, and fixed means it is in a static position, and the axis tag specifies its rotation axis, so our joint the child link is front_left_wheel link and the parent is the base_link, so the front_left_wheel connected to the base with continuous rotation around the z-axis and it 25 cm forward, 32 cm to the left and it is flipped around that means z-axis pointing at the x-axis.

Repeat the process for front_right_wheel, back_left_wheel, and back_right_wheel

Concerning origins, positions, and rotation axis.

3. Feet:
   - ❖ Link:

```
  <link name="left_feet">
    <visual>
    <origin xyz="0.0 0.0 0.0" />
      <geometry>
        <box size="0.15 0.15 0.38"/>
      </geometry>
      <material name="Black"/>
    </visual>
    <collision>
      <geometry>
```

```
        <box size="0.15 0.15 0.38"/>
      </geometry>
    </collision>
  </link>
```

left_feet box shape with dimensions of 15 cm 15 cm 38 cm black colored.

❖ Joint:

```
  <joint name="left_feet_joint" type="fixed">
    <parent link="base_link"/>
    <child link="left_feet"/>
    <origin xyz="0.15 0.15 0.34"/>
    <axis xyz="0.0 0.0 0.0"/>
  </joint>
```

left_feet connected to the base_link with a fixed joint at point (15,15,34).

Repeat the process for right_feet concerning origin and position.

4. Body:

❖ Link:

```
  <link name="body">
    <visual>
      <geometry>
        <box size="0.15 0.45 0.40"/>
      </geometry>
      <material name="Blue"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.15 0.45 0.40"/>
      </geometry>
    </collision>
  </link>
```

The body is a boxed shape with dimensions 15 cm, 45 cm, 40 cm.

❖ Joint:

```
  <joint name="body_joint" type="fixed">
    <parent link="base_link"/>
    <child link="body"/>
    <origin xyz="0.15 0.0 0.73"/>
    <axis xyz="0.0 0.0 0.0"/>
  </joint>
```

The body is connected to the base_link at point (15,0,73).

5. Head:

❖ Link:

```
  <link name="head">
    <visual>
      <geometry>
        <box size="0.15 0.30 0.30"/>
      </geometry>
      <material name="Black"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.15 0.30 0.30"/>
      </geometry>
    </collision>
```

```
</link>
```

The head is a boxed shape with dimensions 15 cm, 30 cm, 30 cm.

❖ Joint:

```
<joint name="head_joint" type="fixed">
  <parent link="base_link"/>
  <child link="head"/>
  <origin xyz="0.15 0.0 1.08"/>
  <axis xyz="0.0 0.0 0.0"/>
</joint>
```

The head is connected to the base_link at point (15,0,108).

3.2.1.2 lidar.xacro file:

This file will contain the description for Lidar and it will specify the geometry, position, and where it is connected this file will contain the gazebo driver for the lidar later in detail to show the full code of this file CTRL+click on the file name.

```
<joint name="laser_joint" type="fixed">
  <parent link="base_link"/>
  <child link="laser_frame"/>
  <origin xyz="0.25 0.0 0.17" rpy="0 0 0"/>
</joint>
<link name="laser_frame">
  <visual>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
    <material name="Black"/>
  </visual>
  <collision>
    <geometry>
      <cylinder radius="0.05" length="0.04"/>
    </geometry>
  </collision>
  </link>
```

3.2.1.3 camera.xacro file:

This file will contain the description for the camera and it will specify the geometry, position, and where it is connected this file will contain the gazebo driver for the lidar later in detail to show the full code of this file CTRL+click on the file name.

```
<joint name="camera_joint" type="fixed">
  <parent link="head"/>
  <child link="camera_link"/>
  <origin xyz="0.065 0.0 0.16" rpy="0 0.2 0"/>
</joint>
<link name="camera_link">
  <visual>
    <geometry>
      <box size="0.02 0.02 0.02"/>
    </geometry>
    <material name="Blue"/>
  </visual>
</link>
<joint name="camera_optical_joint" type="fixed">
  <parent link="camera_link"/>
  <child link="camera_link_optical"/>
```

```
    <origin xyz="0 0 0 " rpy="${-pi/2} 0 ${-pi/2}"/>
  </joint>
  <link name="camera_link_optical"></link>
```
Note that we have a camera joint and an optical joint the camera joint is for the case of the camera and the optical joint is for the lens of the camera the one we will use in the gazebo driver is the optical joint.

3.2.1.4 robot.urdf.xacro file:

this file contains all the above files and this file will be passed to

xacro → robot_state_publisher

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

<xacro:include filename="robot_core.xacro"/>
<xacro:include filename="lidar.xacro"/>
<xacro:include filename="camera.xacro"/>

</robot>
```

Now we can create our robot state publisher launch file, the launch file is a Python script that can run multiple ROS nodes autonomously, and we will use it to pass our URDF to the robot state publisher.

3.2.1.5 rsp.launch.py file:

```
1.  def generate_launch_description():
2.      # Check if we're told to use sim time
3.      use_sim_time = LaunchConfiguration('use_sim_time')
4.      # Process the URDF file
5.      pkg_path = os.path.join(get_package_share_directory('robomealmate'))
6.      xacro_file = os.path.join(pkg_path,'description','robot.urdf.xacro')
7.      robot_description_config = xacro.process_file(xacro_file)
8.      # Create a robot_state_publisher node
9.      params = {'robot_description': robot_description_config.toxml(), 'use_sim_time': use_sim_time}
10.     node_robot_state_publisher = Node(
11.         package='robot_state_publisher',
12.         executable='robot_state_publisher',
13.         output='screen',
14.         parameters=[params]
15.     )
16.     # Launch!
17.     return LaunchDescription([
18.         DeclareLaunchArgument(
19.             'use_sim_time',
20.             default_value='false',
21.             description='Use sim time if true'),
22.         node_robot_state_publisher
23.     ])
```

In line 3 we created a variable called use_sim_time it's Boolean and tells the robot_state_publisher node if we are using simulation time or not and we can assign this value when we launch the file we can pass it as an argument, pkg_path tells the node the path to our package directory, xacro_file is the path to our robot URDF file, then at line 7 we pass our URDF to xacro to create a single description file as we mentioned earlier, then we create our robot state publisher node by mention the package name, executable file, and parameters, in our case the package name is robot_state_publisher and same for executable file and a variable passes the parameters contains robot description configuration and use_sim_time, and at the end of the file, we specify the default value for arguments.

By accomplishing this we can visualize the robot on rviz2.

3.2.1.6 visualize robomealmate on rviz2:

Before we dive into visualization let us take a look at our ROS package for the robot, in the following is the tree of the directory and files:

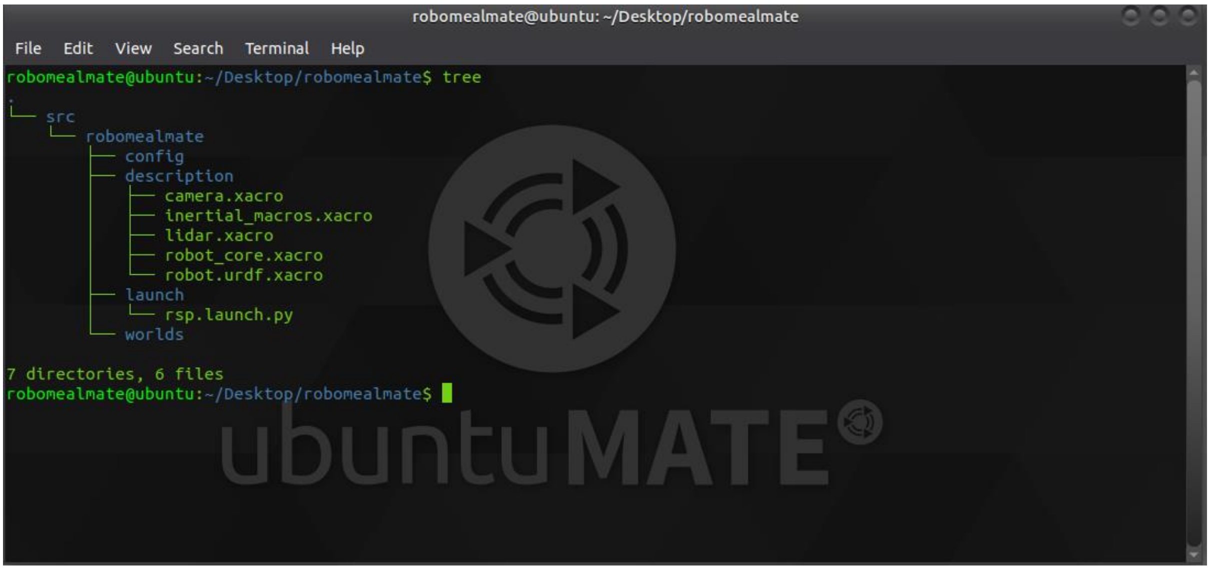


Figure 3.18 robomealmate package tree

We organized the package as in Figure 3.18. The config directory will contain configuration files, the description directory will contain xacro files, the launch directory will contain launch files, and the worlds directory will contain worlds for gazebo.

Let us create our package by ros2 pkg create robomealmate, build it with colcon build –symlink-install, and source our installation by source install/setup.bash

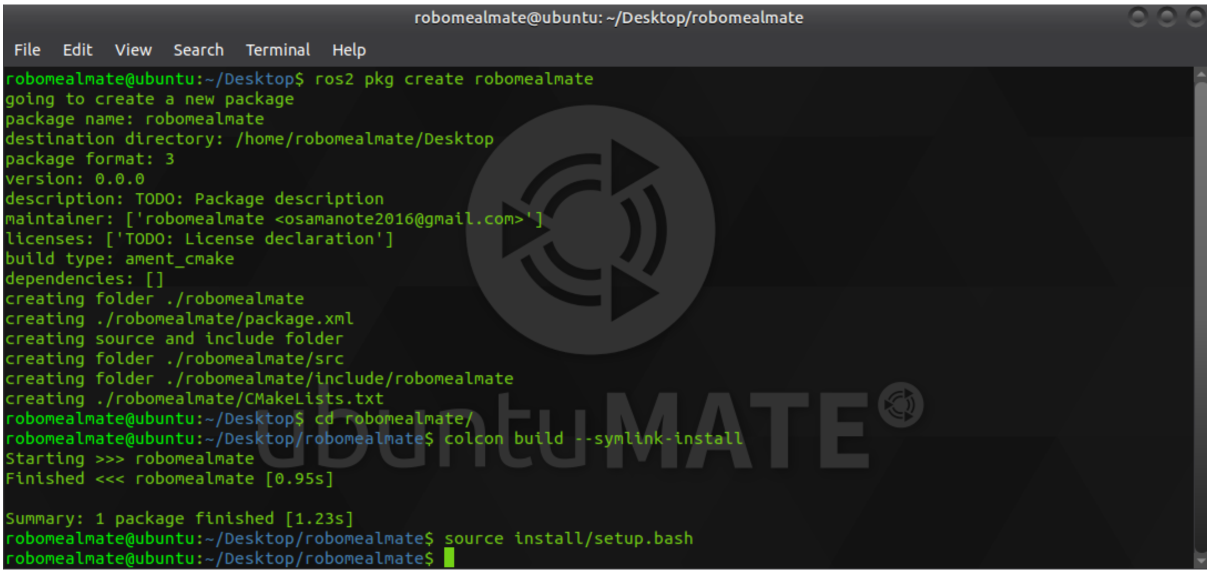
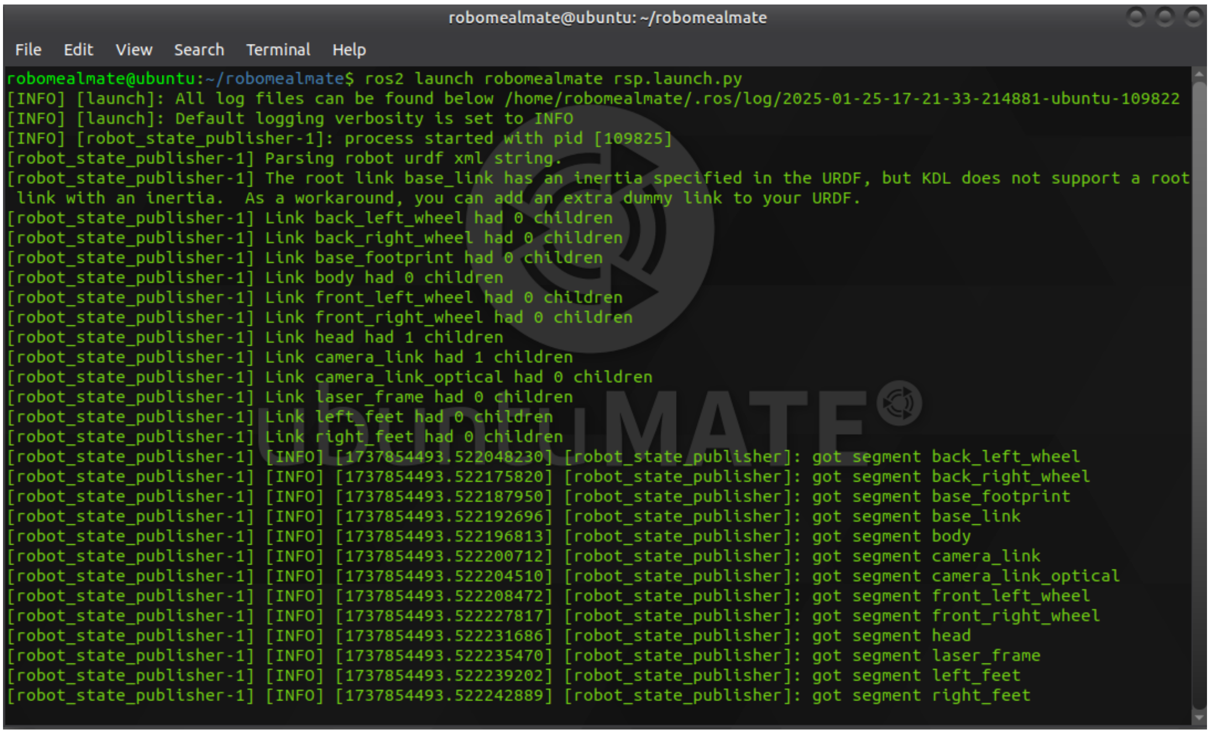


Figure 3.19 robomealmate package creation and build

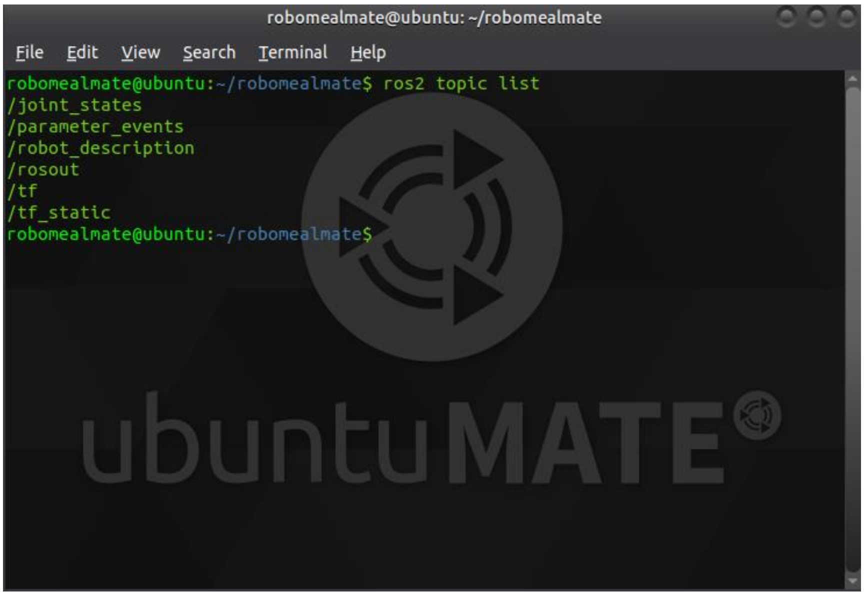
Let us launch our rsp.launch.py by ros2 launch robomealmate rsp.launch.py command.



```
robomealmate@ubuntu: ~/robomealmate
File Edit View Search Terminal Help
robomealmate@ubuntu:~/robomealmate$ ros2 launch robomealmate rsp.launch.py
[INFO] [launch]: All log files can be found below /home/robomealmate/.ros/log/2025-01-25-17-21-33-214881-ubuntu-109822
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [robot_state_publisher-1]: process started with pid [109825]
[robot_state_publisher-1] Parsing robot urdf xml string.
[robot_state_publisher-1] The root link base_link has an inertia specified in the URDF, but KDL does not support a root
 link with an inertia.  As a workaround, you can add an extra dummy link to your URDF.
[robot_state_publisher-1] Link back_left_wheel had 0 children
[robot_state_publisher-1] Link back_right_wheel had 0 children
[robot_state_publisher-1] Link base_footprint had 0 children
[robot_state_publisher-1] Link body had 0 children
[robot_state_publisher-1] Link front_left_wheel had 0 children
[robot_state_publisher-1] Link front_right_wheel had 0 children
[robot_state_publisher-1] Link head had 1 children
[robot_state_publisher-1] Link camera_link had 1 children
[robot_state_publisher-1] Link camera_link_optical had 0 children
[robot_state_publisher-1] Link laser_frame had 0 children
[robot_state_publisher-1] Link left_feet had 0 children
[robot_state_publisher-1] Link right_feet had 0 children
[robot_state_publisher-1] [INFO] [1737854493.522048230] [robot_state_publisher]: got segment back_left_wheel
[robot_state_publisher-1] [INFO] [1737854493.522175820] [robot_state_publisher]: got segment back_right_wheel
[robot_state_publisher-1] [INFO] [1737854493.522187950] [robot_state_publisher]: got segment base_footprint
[robot_state_publisher-1] [INFO] [1737854493.522192696] [robot_state_publisher]: got segment base_link
[robot_state_publisher-1] [INFO] [1737854493.522196813] [robot_state_publisher]: got segment body
[robot_state_publisher-1] [INFO] [1737854493.522200712] [robot_state_publisher]: got segment camera_link
[robot_state_publisher-1] [INFO] [1737854493.522204510] [robot_state_publisher]: got segment camera_link_optical
[robot_state_publisher-1] [INFO] [1737854493.522208472] [robot_state_publisher]: got segment front_left_wheel
[robot_state_publisher-1] [INFO] [1737854493.522227817] [robot_state_publisher]: got segment front_right_wheel
[robot_state_publisher-1] [INFO] [1737854493.522231686] [robot_state_publisher]: got segment head
[robot_state_publisher-1] [INFO] [1737854493.522235470] [robot_state_publisher]: got segment laser_frame
[robot_state_publisher-1] [INFO] [1737854493.522239202] [robot_state_publisher]: got segment left_feet
[robot_state_publisher-1] [INFO] [1737854493.522242889] [robot_state_publisher]: got segment right_feet
```

Figure 3.20 robomealmate state publisher launch

As we see in Figure 3.20 the robot state publisher identified the links and joints of our robot and if we run the ros2 topic list command we should see the topics published.



```
robomealmate@ubuntu: ~/robomealmate
File Edit View Search Terminal Help
robomealmate@ubuntu:~/robomealmate$ ros2 topic list
/joint_states
/parameter_events
/robot_description
/rosout
/tf
/tf_static
robomealmate@ubuntu:~/robomealmate$
```

Figure 3.21 topic list after launch publisher

In Figure 3.21 we see that /tf, and /robot_description topics are published, and /joint_state topic waiting for the values from joint_state_publisher_gui.

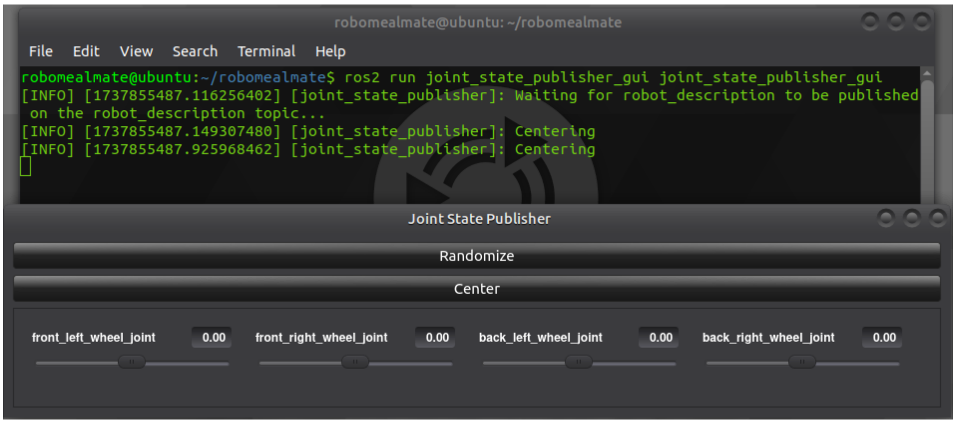


Figure 3.22 robomealmate joint gui launch

From Figure 3.22 we can see that our joint_state_publisher_gui detects the continuous joints and it can assign values to it. Now let us open rviz2 and see our robot.

When opening the rviz2 click on the add button then from the menu select TF to show the robot transforms.

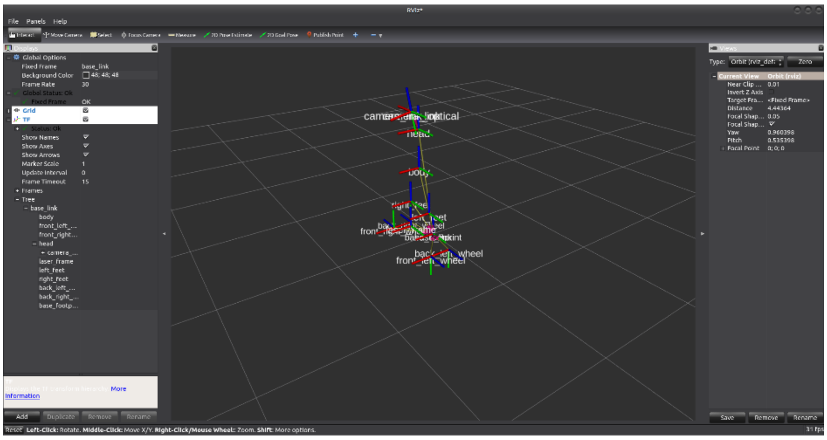


Figure 3.23 robomealmate TF

Now let's show the robot model, by clicking on add then select robot model then ok, on the left side under topic select topic to be /robot_describtion.

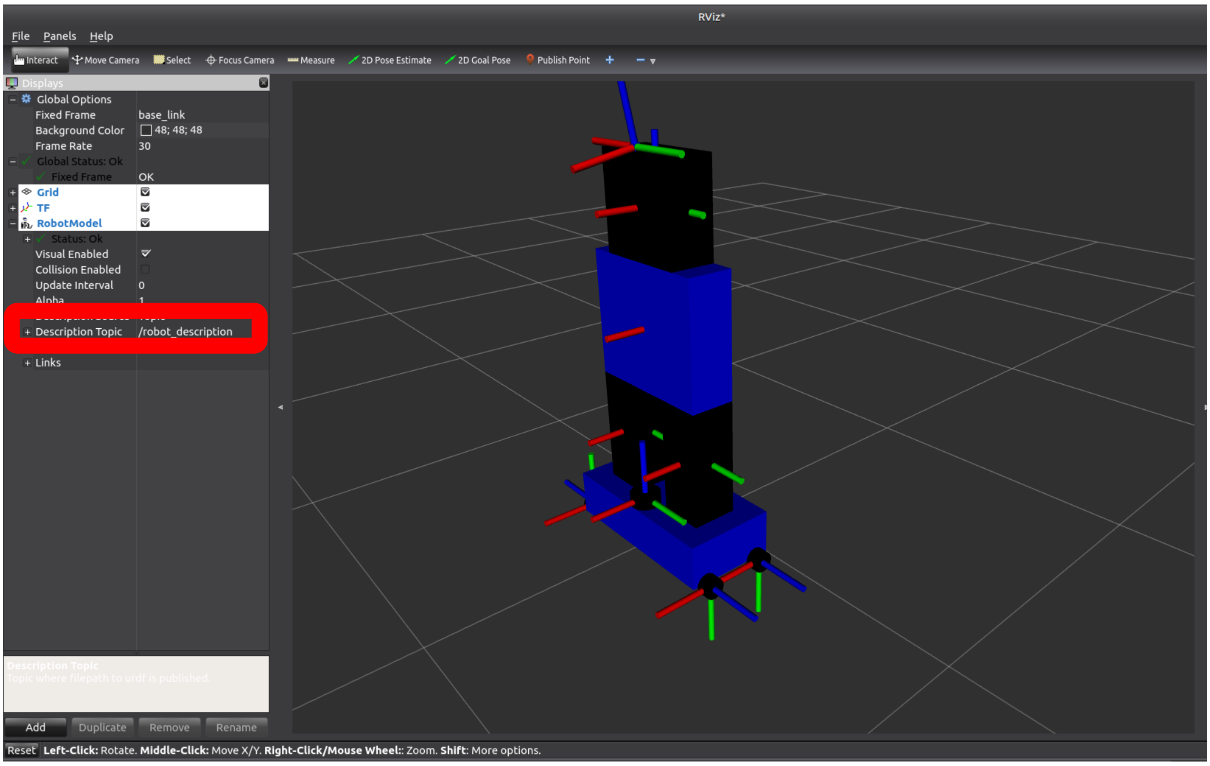


Figure 3.24 robomealmate Model

Now we are ready to move to the next step gazebo.

3.2.2 Gazebo simulation:

Before we can simulate the robot on the gazebo we have to:

1. Modify the URDF to work with the gazebo by adding physics tags (inertia) and materials, because the gazebo supports physics engineers.
2. Add the lidar driver to the lidar xacro file to work on the gazebo.
3. Add the camera driver to the camera xacro file to work on the gazebo.
4. Create a simulation launcher file for the launch of the robot simulation.

3.2.2.1 URDF modifications:

1. Add an inertia tag to the robot parts, An inertia tag is used to define the **inertia properties** of a robot's links, inertia properties are critical for accurate simulation of the robot's dynamics, including how it moves, reacts to forces, mainly in robomealmate robot we have two shapes box and cylinder, so we have to find the equation of moment of inertia for the box and cylinder. [1] , we created a file called inertial_macros.xacro this file contains the equations of the moment of the inertia for box and cylinder and we can include it in robot_core.xacro file and use the macros by name.

   ✓ Box moment of inertia:

```
    <xacro:macro name="inertial_box" params="mass x y z *origin">
        <inertial>
            <xacro:insert_block name="origin"/>
            <mass value="${mass}" />
            <inertia ixx="${(1/12) * mass * (y*y+z*z)}" ixy="0.0" ixz="0.0"
                    iyy="${(1/12) * mass * (x*x+z*z)}" iyz="0.0"
                    izz="${(1/12) * mass * (x*x+y*y)}" />
```

```
        </inertial>
    </xacro:macro>
```

✓ Cylinder moment of inertia:

```
    <xacro:macro name="inertial_cylinder" params="mass length radius *origin">
        <inertial>
            <xacro:insert_block name="origin"/>
            <mass value="${mass}" />
            <inertia ixx="${(1/12) * mass * (3*radius*radius + length*length)}" ixy="0.0" ixz="0.0"
                    iyy="${(1/12) * mass * (3*radius*radius + length*length)}" iyz="0.0"
                    izz="${(1/2) * mass * (radius*radius)}" />
        </inertial>
    </xacro:macro>
```

Now we can include the file on robot_core.xacro file

```
<xacro:include filename="inertial_macros.xacro"/>
```

To use the inertial macros we have to the following tag inside the links under the collision tag:

✓ for box geometry:

```
    <xacro:inertial_box  mass="5.0" x="0.30" y="0.60" z="0.15">
      <origin xyz="0.15 0.0 0.075" rpy= "0.0 0.0 0.0"/>
    </xacro:inertial_box>
```

The origins should be the same as the link origin if there is no origin for a link make it zeros.

✓ For cylinder geometry:

```
    <xacro:inertial_cylinder  mass="0.10" radius="0.035" length="0.04">
      <origin xyz="0.0 0.0 0.0" rpy= "0.0 0.0 0.0"/>
    </xacro:inertial_cylinder>
```

Repeat the process for all links in the robot URDF.

2. Add gazebo material to the URDF to see the colors in the gazebo, we can add by gazebo tag and the reference is for the link we need to add the material as shown below:

```
  <gazebo reference="base_link">
    <material>Gazebo/Blue</material>
  </gazebo>
```

Repeat the process for all links in the URDF.

3.2.2.2 Add lidar gazebo driver

Gazebo packages coms with a bunch of plugins for sensors, the plugins give us the ability to test sensors in virtual environments, and the plugin we will use for the lidar is called "libgazebo_ros_ray_sensor.so", this plugin needs some configurations, below is how to configure the plugin driver in lidar.xacro file:

```
    <gazebo reference="laser_frame">
        <material>Gazebo/Red</material>
        <sensor name="laser" type="ray">
            <pose>0 0 0 0 0 0</pose>
            <visualize>true</visualize>
            <update_rate>10</update_rate>
            <ray>
```

```xml
                <scan>
                    <horizontal>
                        <samples>360</samples>
                        <resolution>1</resolution>
                        <min_angle>-3.14</min_angle>
                        <max_angle>3.14</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <min>0.3</min>
                    <max>12.0</max>
                    <resolution>0.01</resolution>
                </range>
            </ray>
            <plugin name="gazebo_ros_laser" filename="libgazebo_ros_ray_sensor.so">
                <ros>
                    <argument>~/out:=scan</argument>
                </ros>
                <output_type>sensor_msgs/LaserScan</output_type>
                <frameName>laser_frame</frameName>
            </plugin>
        </sensor>
    </gazebo>
```

The reference link for the plugin is laser_frame, the color of the lidar is red, and the position in its original origin, we need to see the laser so the visualize tag is true, the update_rate for readings is 10Hz, the scan should be done horizontal, samples 360 in resolution 1 and the reading angle between -3.14 to 3.14, the range covered is at min 3 cm and the max is 12 meter, and the plugin we will use for this sensor is “libgazebo_ros_ray_sensor.so”, the ros topic that the scan data will be published to it is /scan topic the type of data sensor_msgs/LaserScan, and the frame is laser_frame, by this setting we set up the lidar sensor, and it is similar to our real lidar.

3.2.2.3 Add camera gazebo driver

The plugin we will use for the camera is “libgazebo_ros_camera.so” and below is how to configure the plugin driver in the camera.xacro file:

```xml
    <gazebo reference="camera_link">
        <material>Gazebo/Red</material>
        <sensor name="camera" type="camera">
            <pose>0 0 0 0 0 0</pose>
            <visualize>true</visualize>
            <update_rate>10</update_rate>
            <camera>
                <horizontal_fov>1.089</horizontal_fov>
                <image>
                    <width>640</width>
                    <height>480</height>
                    <format>R8G8B8</format>
                </image>
                <clip>
                    <near>0.05</near>
                    <far>8.0</far>
                </clip>
            </camera>
            <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
                <frameName>camera_link_optical</frameName>
            </plugin>
        </sensor>
    </gazebo>
```

From line 9 to line 13, we are setting up the image properties. Inside the clip tag, we specify the lens capabilities and the frame that will be used is the camera_link_optical.

3.2.2.4 simulation.launch.py file

As we did before with rsp.launch.py, but this file will be responsible for launching 1 node, and 2 launch files the files are rsp.launch.py we created before and the file that launches the gazebo gazebo.launch.py file, and the node is spawning node it responsible for spawn robomealmate in gazebo environment.

1. rsp launch file:

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

Note that the argument 'use_sim_time' is true because we are using a simulated environment.

2. Gazebo launch:

```
    gazebo = IncludeLaunchDescription(
                PythonLaunchDescriptionSource([os.path.join(
                    get_package_share_directory('gazebo_ros'), 'launch',
 'gazebo.launch.py')]),
             )
```

Gazebo launch file, provided by the gazebo_ros package.

3. Run the spawner node from the gazebo_ros package:

```
    spawn_entity = Node(package='gazebo_ros', executable='spawn_entity.py',
                        arguments=['-topic', 'robot_description',
                                   '-entity', 'robomealmate'],
                        output='screen')
```

5. Launching:

```
    return LaunchDescription([
        rsp,
        gazebo,
        spawn_entity,
    ])
```

3.2.2.5 Launch simulation:

Open the terminal navigate to our package directory colcon build to apply new files then run the ros2 launch robomealmate simulation.launch.py command.

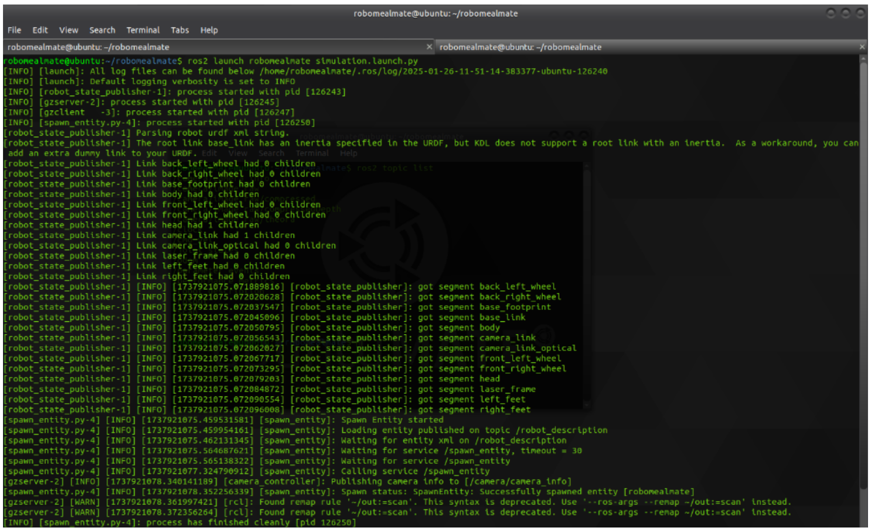


Figure 3.25 robomealmate simulation launch

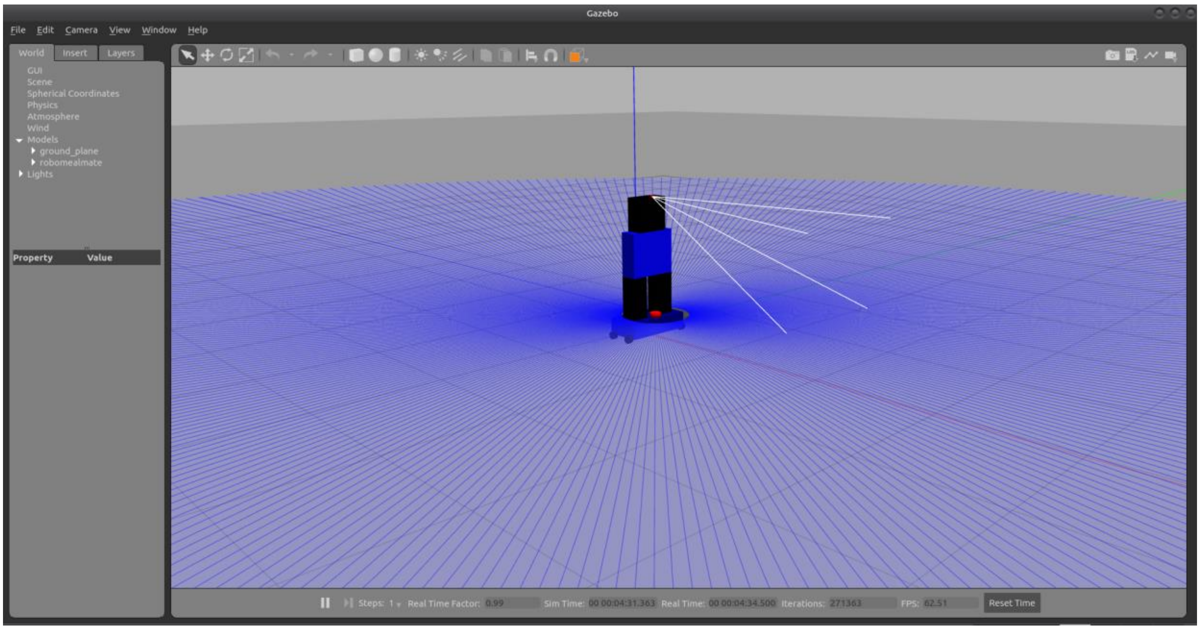


Figure 3.26 robomealmate simulation launch

As we see in Figure 3.25 and Figure 3.26 our simulation launch file was done successfully, the robot_state_publisher was working, the gazebo launch file was working, and the spawner node spawned our robot in the gazebo environment, and it's clear that the lidar and camera is working but lets double check by see the topic list.

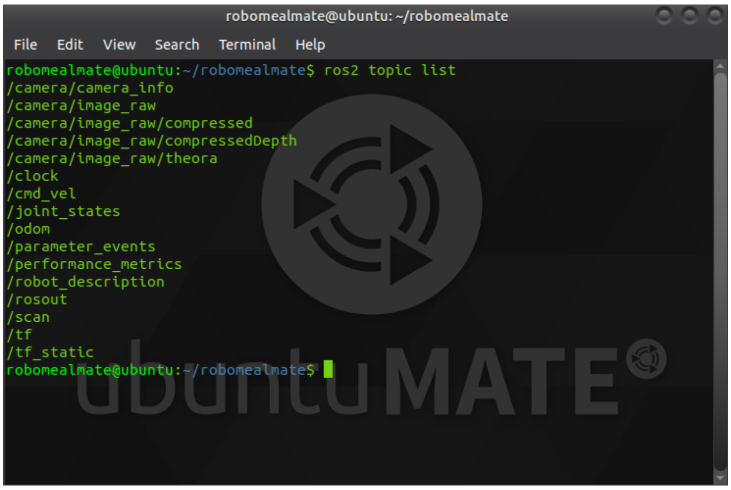


Figure 3.27 topic list after simulation

As shown in Figure 3.27 scan topic and camera topics are publishing data and ready for processing.

Now we are ready to move to the next step driving our simulated robomealmate!!!!.

3.2.3 Gazebo controller (drive simulated robomealmate):

This section is divided into:

1. Understanding control.
2. Gazebo controller setting up.
3. Driving simulated robomealmate.

3.2.3.1 Understanding control:

Each robot needs a control system, the control system is responsible for getting command velocity as an input, then translating that to motor command for motor drivers, reading the actual motor speed from motor drivers, then calculating the true velocity, in ros the command velocity is on a topic called /cmd_vel, and the type of messages is a twist, and the twist message contains 6 numbers linear velocity on x, y, and z axis, and angular velocity on x, y, and z, in our robomealmate we will use a differential drive that’s mean we need just the linear velocity on x for forward and backward movement, and angular velocity on z for rotation left and right, rather than true velocity we are interested on robot position, the control system can estimate it by integrating the velocity over time, adding it up in tiny little time steps. This process is called a dead reckoning, and the resulting position estimate is called our odometry, the following diagram shows the process.

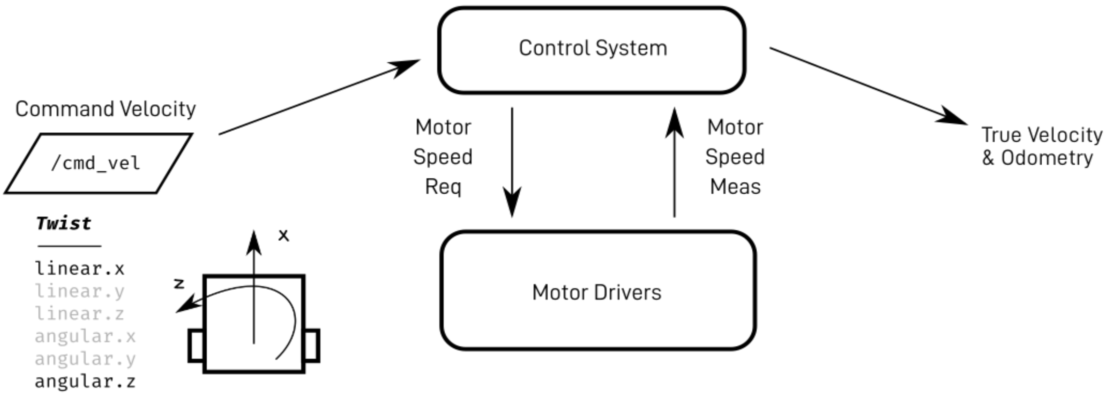


Figure 3.28 understanding control diagram

As we see before whenever we need ros to interact with the gazebo we need a plugin, The gazebo_ros package provides a control plugin for a differential drive.

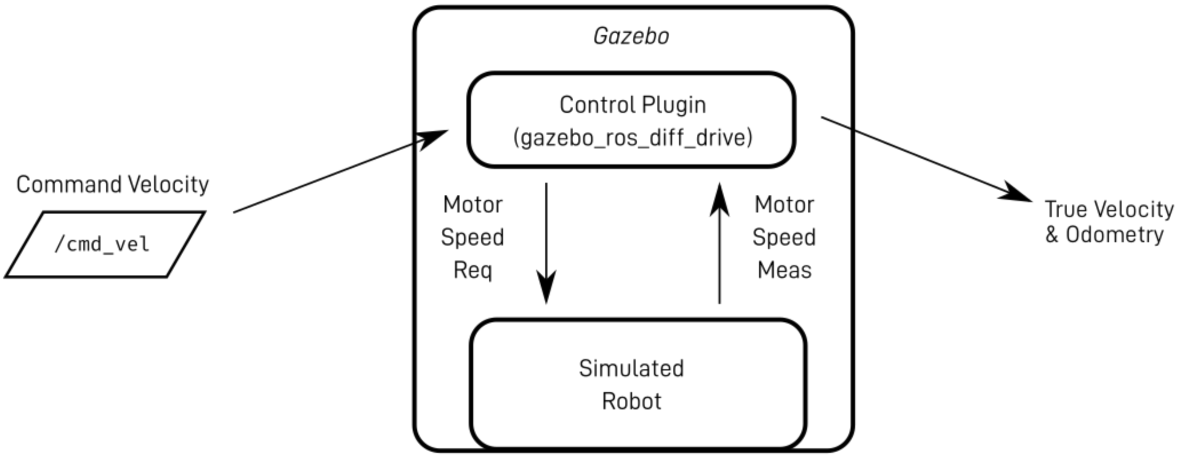


Figure 3.29 control gazebo

This plugin will interact with core gazebo code which simulates motors and others, and we will replace the previous system for /joint_state with this as shown in Figure 3.30.

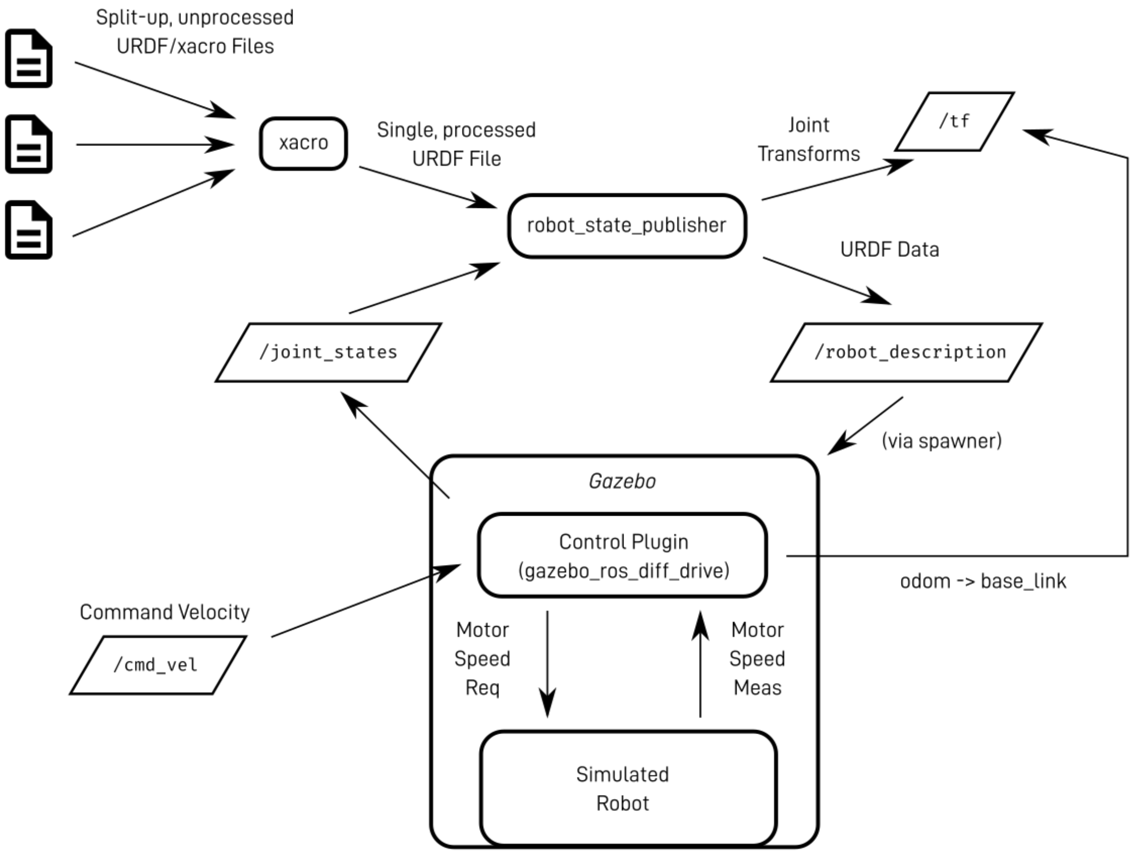


Figure 3.30 controlling simulated robomealmate diagram

Instead of faking /joint_state with joint_state_publisher_gui, the gazebo robot spawned from the robot description, and the joint state is published by the control plugin, the plugin also broadcasts a transform for a new frame called odom (which is like the world origin, the robot's start position), to base_link, which lets any other code know the current position estimate for our robomealmate.

3.2.3.2 Gazebo controller setting up:

To be able to use the plugin we have to add a new file called gazebo_controller.xacro in the description directory and include it on robot.urdf.xacro file, the file contains the configuration and parameters for the plugin.

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <gazebo>
        <plugin name="diff_drive" filename="libgazebo_ros_diff_drive.so">
            <!-- Wheel Information -->
            <num_wheel_pairs>2</num_wheel_pairs>
            <left_joint>front_left_wheel_joint</left_joint>
            <left_joint>back_left_wheel_joint</left_joint>
            <right_joint>front_right_wheel_joint</right_joint>
            <right_joint>back_right_wheel_joint</right_joint>
            <wheel_separation>0.6</wheel_separation>
            <wheel_diameter>0.07</wheel_diameter>
            <!-- Limits -->
            <max_wheel_torque>200</max_wheel_torque>
            <max_wheel_acceleration>10.0</max_wheel_acceleration>
            <!-- Output -->
            <odometry_frame>odom</odometry_frame>
            <robot_base_frame>base_link</robot_base_frame>
            <publish_odom>true</publish_odom>
            <publish_odom_tf>true</publish_odom_tf>
            <publish_wheel_tf>true</publish_wheel_tf>
        </plugin>
    </gazebo>
</robot>
```

We have 2 wheel pairs one on the left with two wheels and one on the right with two wheels, we specify our wheel joints on the left and right, the separation of the wheel means the distance between the wheels pair and in our case is 60 cm, wheel diameter is 7 cm, the max torque for the motors for wheels is 200 rpm, max wheel acceleration 10, the odometry topic will publish is odom, and the base frame for our robot is base_link, we need to publish odom, odom transform, and wheel transforms, by this we are setting up the diff drive plugin and we are ready to control our virtual robomealmate.

3.2.3.3 Driving simulated robomealmate

Before we run simulation.launch.py we need to make sure to include the gazebo_controller.xacro file into robot.urdf.xacro file.

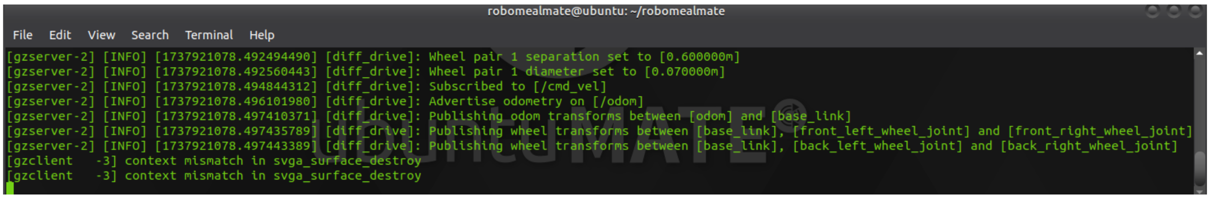


Figure 3.31 simulation with gazebo controller

As we see in Figure 3.31 our configuration for the diff plugins is taken and now we have odom topic and the controller subscribes to the cmd_vel topic.

To drive our simulated robomealmate we will use a ros tool called teleop twist keyboard this tool publishes twist messages to cmd_vel topic using the keyboard.

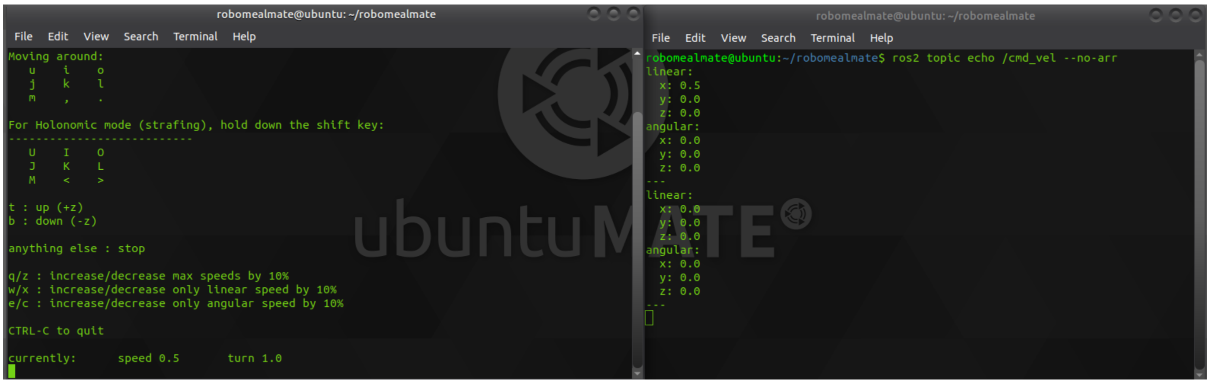


Figure 3.32 teleop twist keyboard with cmd_vel topic

As we see in Figures 3.31 & 3.32 the teleop twist is publishing command velocity to cmd_vel topic and gazebo controller subscribes the cmd_vel and our simulated robomealmate starts moving.

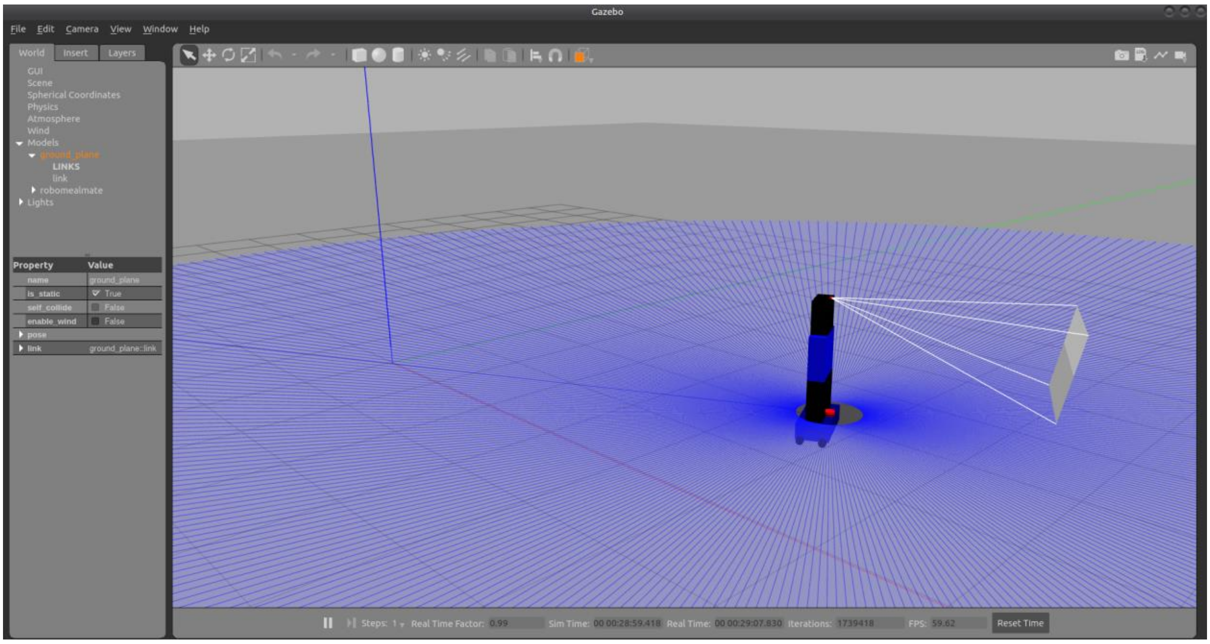


Figure 3.33 simulated robomealmate moving

What have we done so far?

1. Created robomealmate URDF

2. Showed robomealmate on rviz2
3. Created camera and lidar driver for gazebo.
4. Simulated robomealmate in gazebo.
5. Created launch files for simulation
6. Integrated gazebo control with ros to drive robomealmate in virtual environments.

Now we are ready for the next step SLAM.

3.2.4 SLAM (Simultaneous Localization and Mapping):

At the end of this section, simulated robomealmate will be able to generate a map of the environment around him and localize itself within the map.

What are we going to do?

1. Understand SLAM.
2. Integrate the SLAM toolbox coming with ros with simulated robomealmate.
3. Generate a map for the virtual environment.

3.2.4.1 What is SLAM?

SLAM is an acronym that stands for **S**imultaneous **L**ocalization **a**nd **M**apping, so what is localization and mapping?

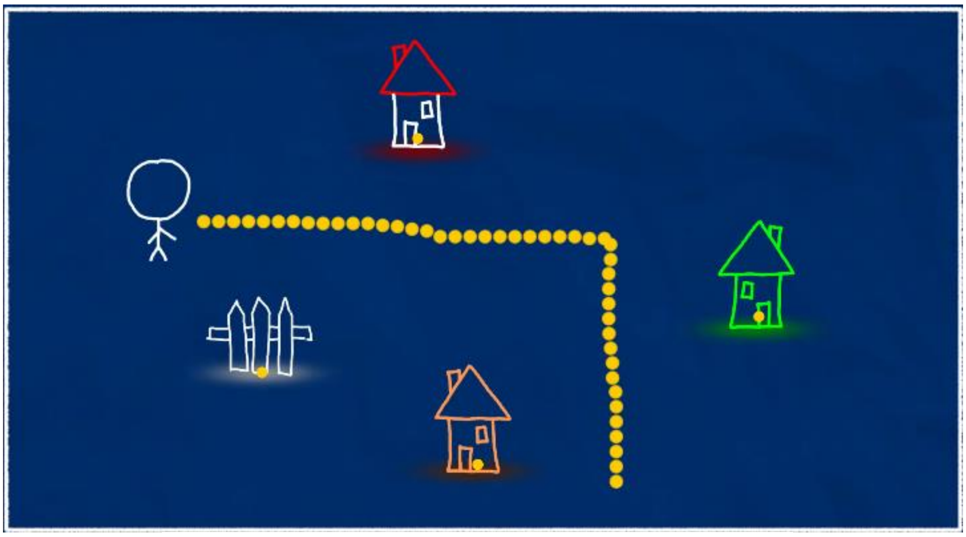


Figure 3.34 mapping

Let's start with mapping, imagen you won't make a map for your street, but you have got a phone on you so that you can track your GPS location and as you walk along the street you take note of what you see, for example, orange house on the left greenhouse on the right, then a corner, turn the corner red roof on the right, white fence on the left and so on, once you finished you have a nice little map of your street.

Now you have got your map you can use it for localization, let's say your phone battery runs out so you have no GPS and you need to figure out where you are.

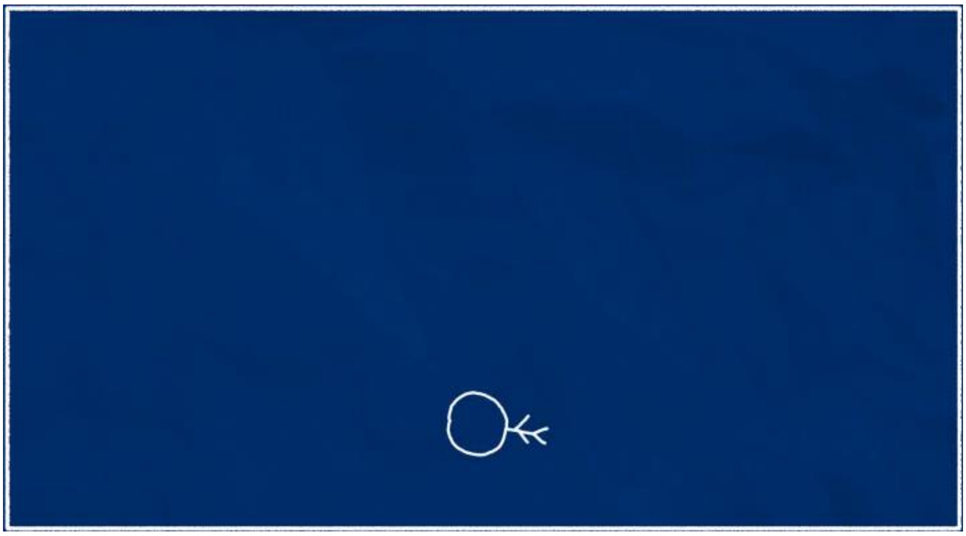


Figure 3.35 localization 1

You can see a white fence on your right and a red roof further up on the left.

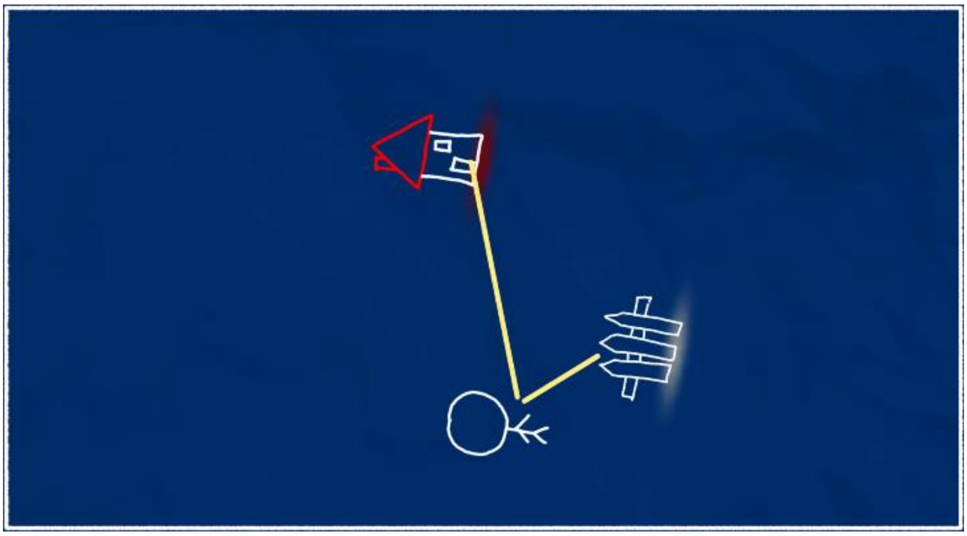


Figure 3.36 localization 2

Using the map you will be able to pretty accurately pinpoint your location wherever you are, you found your location in the global coordinate system, you have localized.

Figure 3.37 localization 3

The problem with the approach we did is that we needed that GPS on our phone in the first place to make an accurate map but sometimes we might only have a GPS coordinate for our starting position or no GPS at all in that case we need to SLAM, we need simultaneously localize and map.

From our starting position, we might be able to see the orange house and the greenhouse.

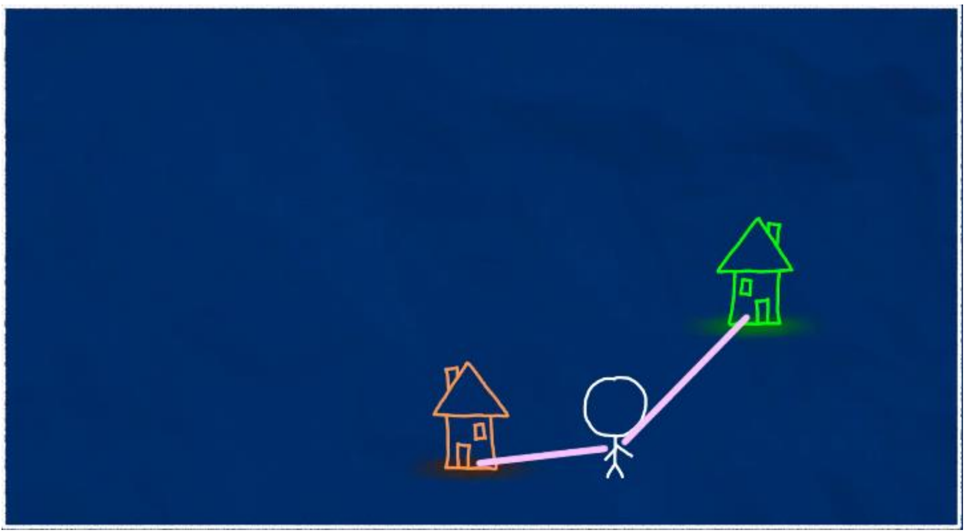


Figure 3.38 SLAM 1

As we walk we will keep an eye on where they are and consequently, where we are compared to them, we could use our stride length to help us with this position estimate.

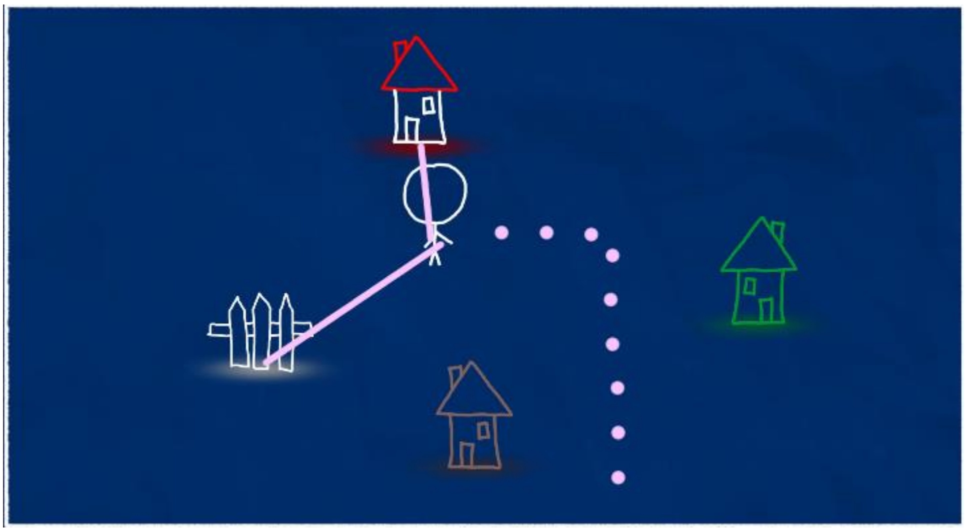


Figure 3.39 SLAM 2

Then whenever we see a new object we know where it is because we know where we are and it all goes on continuously.


Figure 3.40 SLAM 3

Congratulations, we just slammed and the result may not be as accurate as with the GPS but it's much better than having no map at all.

SLAM comes in many different forms with many different algorithms, but often we can categorize most SLAM methods into one of two categories, there is feature or Landmark SLAM, and grid SLAM, what we just saw was feature SLAM, so our features or our Landmarks for things like red roof or white fence, grid slam, on the other hand, is where we divide the world we are seeing up into a series of cells and each cell can either be occupied or unoccupied or somewhere in between, slam toolbox we are going to use on our robomealmate is a grid map-based approach, so we are going to use grid SLAM and 2D lidar.

3.2.4.2 Integrate SLAM toolbox

Before we dive into the process, we want to introduce a new frame, the map frame, we can express the location of base_link compared to the map frame so we can have base_link compared to the odom frame or compared to the map frame.

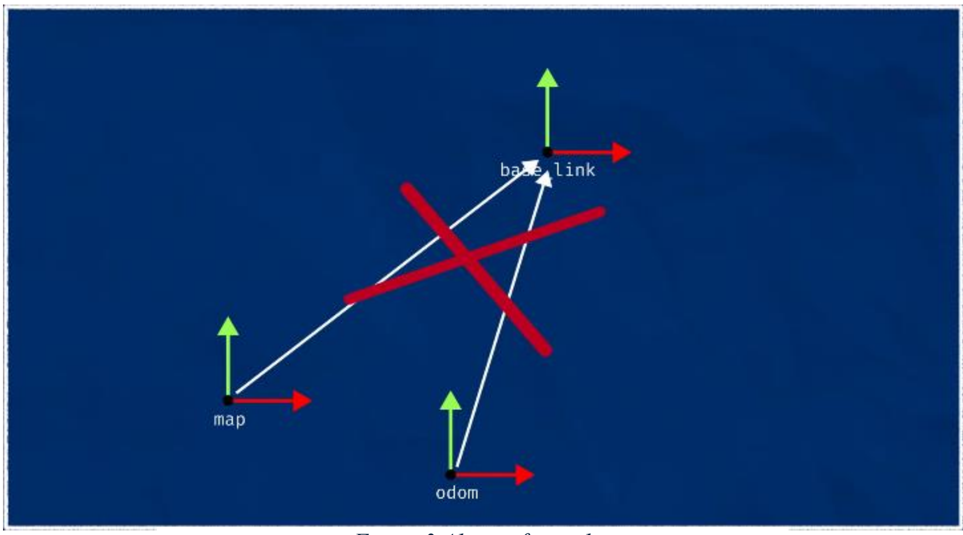


Figure 3.41 map frame 1

This gives us a new problem though which is that a frame in ROS can only have one parent.

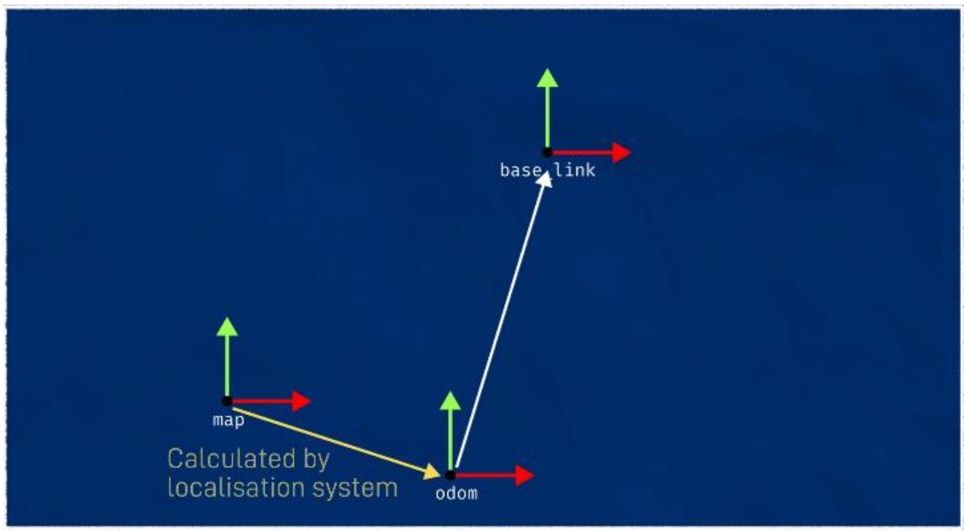


Figure 3.42 map frame 2

The way to get around this is that the code will need to take the position estimate from SLAM along with the current odom to the base_link transform and calculate the appropriate map to the odom transform, that means we can get the base_link position relative to either reference point with no troubles and should move smoothly but drift compared to odom and jump around but generally stay correct over time compared to the map.

As well as the odom and map frames you will also find odom and map topics.

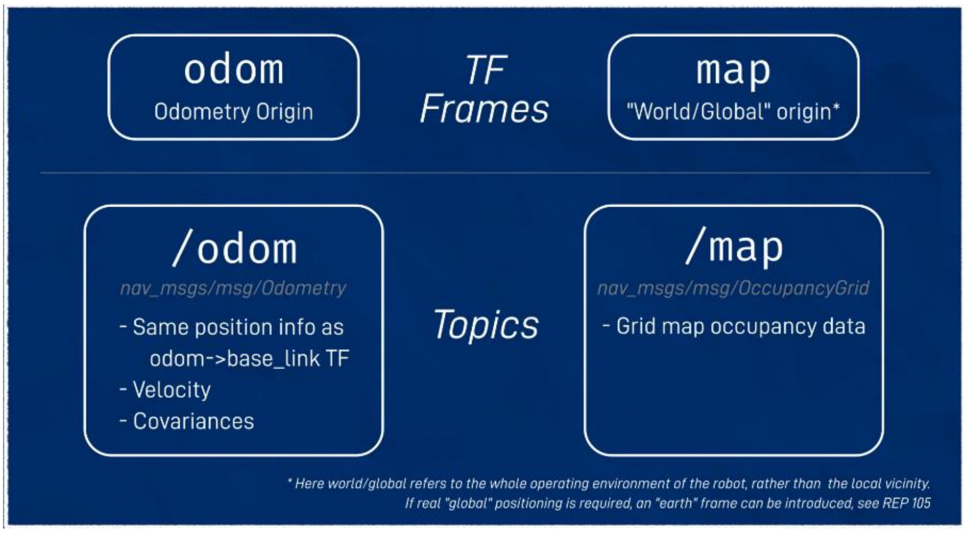


Figure 3.43 map and odom topics

Which contains data about the odometry and the map, the odom topic contains basically the same position information as the odom to base_link transform, but it also contains the current velocity and the associated covariances, the map topic contains the actual occupancy data for the grid map so that it can be shared to other nodes.

One other thing to be aware of is that in addition to the base_link frame some SLAM systems also like you to have a footprint frame, and that's because some robots can move up and down in 3D the base_link will move up around z, but we need to treat the SLAM problem as a 2D SLAM problem, so the base footprint frame is kind like a shadow of the base_link frame on the ground, so it stuck to the x,y plane z equal zero it's stuck on the ground and it will move around underneath the base_link and be used for SLAM.

First, let us add the footprint link to our URDF (robot_core.xacro):

```
<link name="base_footprint">
</link>
<joint name="base_footprint_joint" type="fixed">
  <parent link="base_link"/>
  <child link="base_footprint"/>
  <origin xyz="0 0 0.0" rpy="0 0 0"/>
</joint>
```

SLAM toolbox has a few different modes that it can run in and a lot of different options, the mode we are going to be using is called online asynchronous, so online means we're running it live rather than running it through some previously logged data, and asynchronous means that don't care about processing every scan instead say that our processing rate is a bit slower than the scan rate we just always want to be processing the latest scan even if that means we miss one occasionally.

SLAM toolbox comes with some launch files and params files that help us to set the options so we are going to take one of those params files and copy it into our local directory config, the file name mapper_params_online_async.yaml.

```
robomealmate@ubuntu:~$cp/opt/ros/foxy/share/slam_toolbox/config/mapper_params_online_asynce
.yaml \
> robomealmate/src/robomealmate/config/
```

We are only going to change a couple of the parameters, and the most important ones are:

```
# ROS Parameters
odom_frame: odom
map_frame: map
base_frame: base_footprint
scan_topic: /scan
mode: mapping #localization
```

By this we are telling the SLAM toolbox that the odom frame is odom, we need to publish the map on map topic, the base frame of robomealmate is base_footprint, the scan topic will take the readings from is scan, and the mode is mapping, there is another mode called localization it used for prepared map data we will see it later in detail, but know we are good to go.

3.2.4.3 Launching SLAM

First colcon builds the workspace and launches the simulation as we did before.

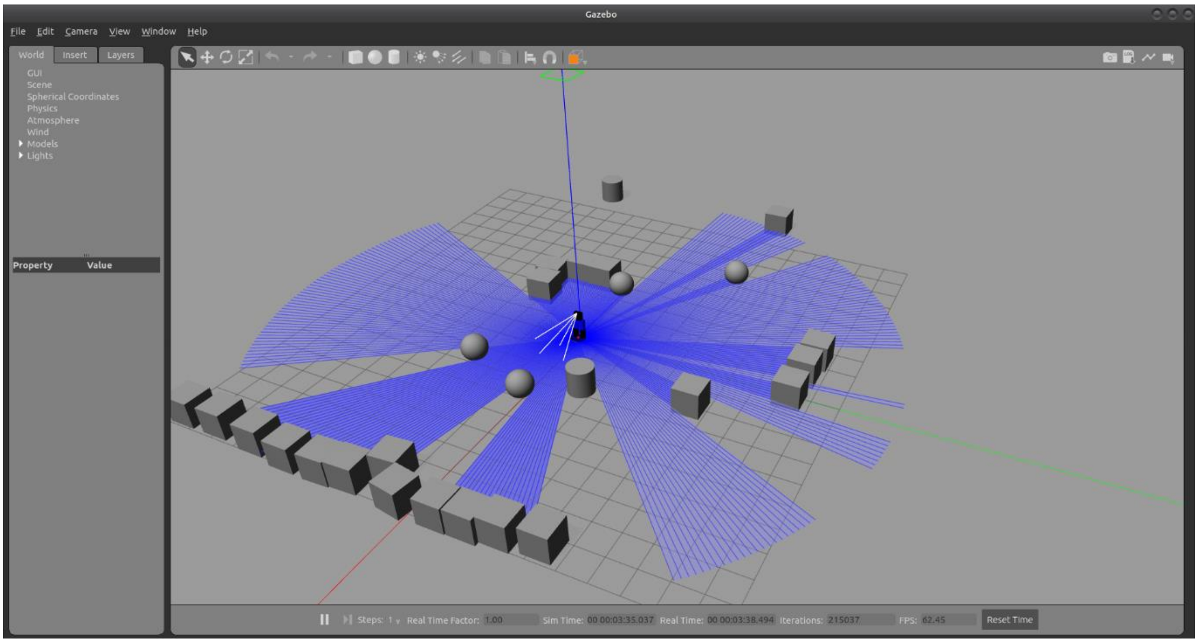


Figure 3.44 gazebo obstacles

To test we added some obstacles in the virtual environment.

Now let's run rviz2, once it launched we need to add a laser scan from the add menu as we did with TF and robot model, the topic for the laser scan is scan, and we need to add the image view to view the virtual environment on rviz2 using the camera.

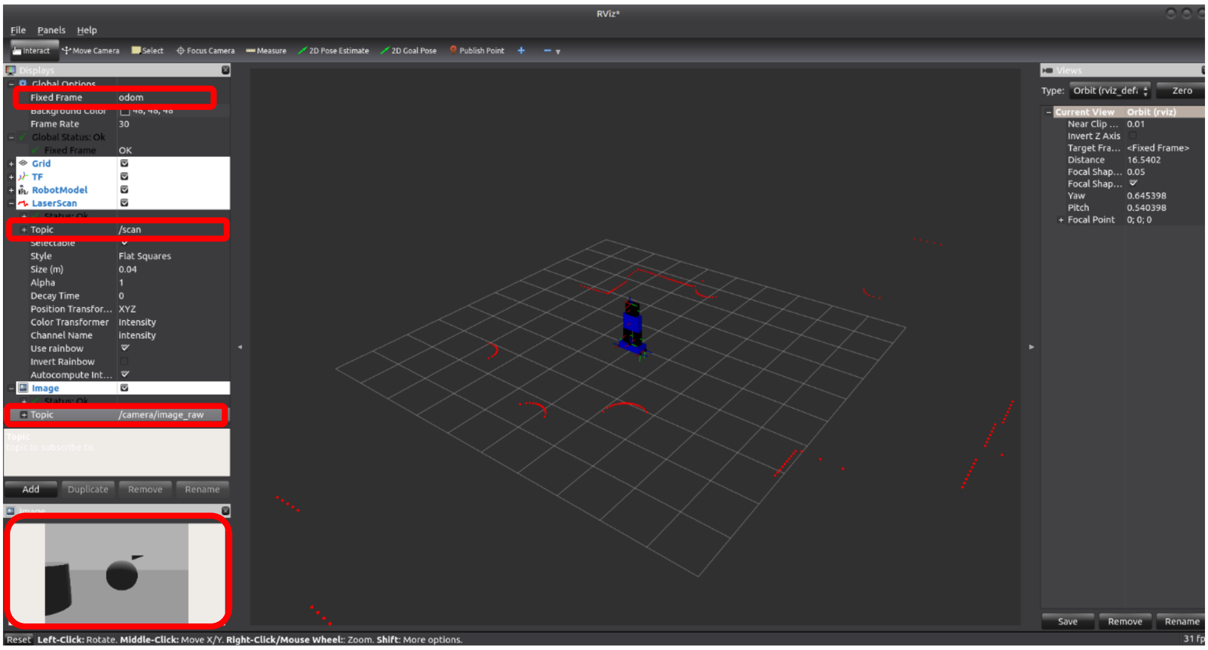


Figure 3.45 rviz2 view

As we see in Figure 3.45 the laser scan is publishing data and we can see the red lines clearly, the camera is working and we can see the virtual environment, note that the fixed frame is odom and this will change when we run the SLAM to be mapped.

Now let us run the SLAM toolbox by executing the following command.

```
1. robomealmate@ubuntu:~/robomealmate$ ros2 launch slam_toolbox online_async_launch.py
   params_file:= ./src/robomealmate/config/mapper_params_online_async.yaml
   use_sim_time:=true
```

in this command, we are running slam_toolbox using online asynchronous mode with the parameter file we created before and setting sim time to true.

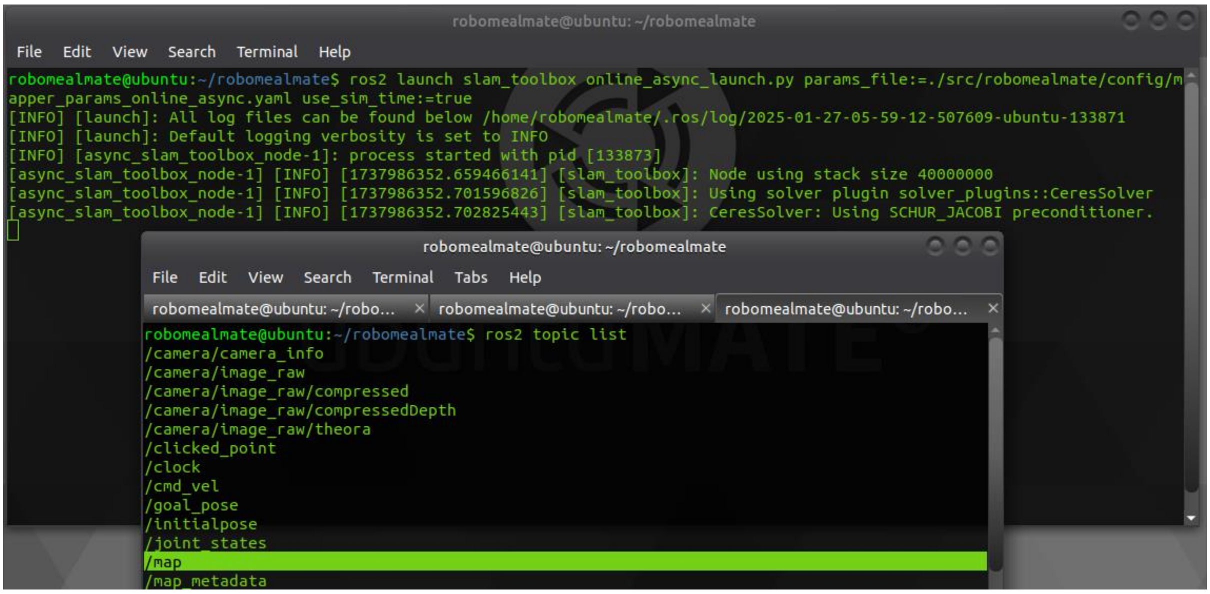


Figure 3.46 launching SLAM toolbox

As we can see from Figure 3.46, the SLAM toolbox has started publishing the map on the map topic, now we can see our map in rviz2 by adding a map from the add menu and then setting the topic to map.

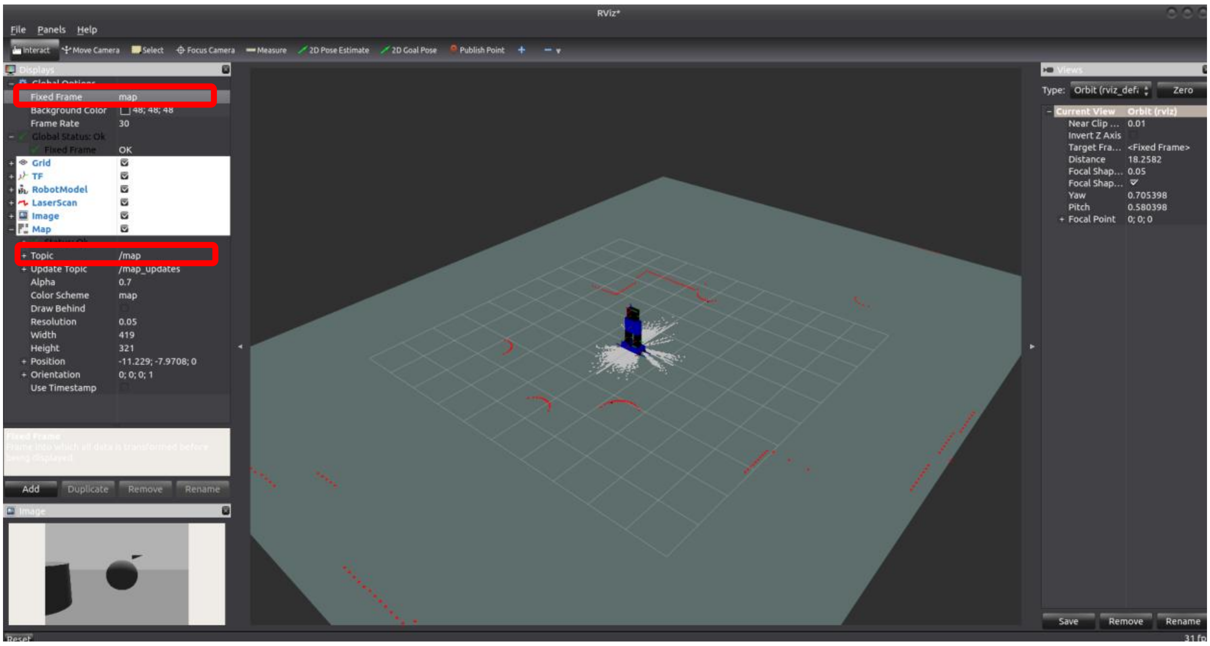


Figure 3.47 rviz2 map view

As is clear from Figure 3.47 our map starts appearing, let's drive around and create our map.

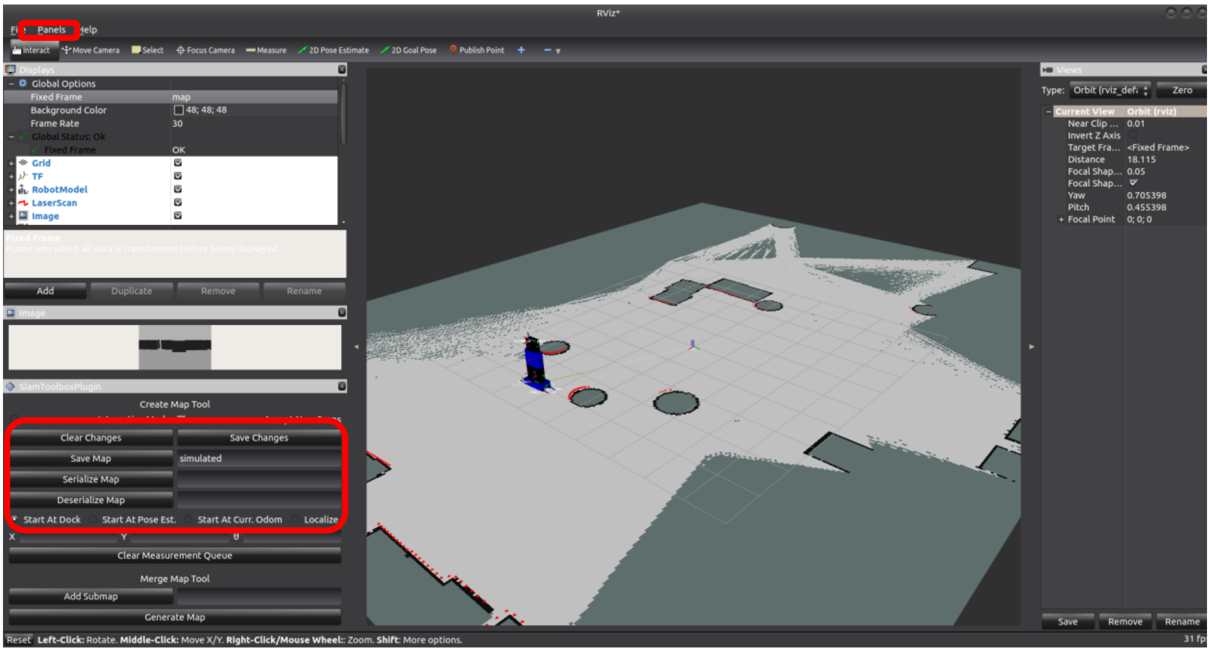


Figure 3.48 rviz2 map save

As shown in Figure 3.48 our map was created and updated while driving, Compared to the virtual environment in Figure 3.44 it is identical.

We can save our generated map by adding SLAM toolbox panel form panels in rviz2, then name our map and save it, we can use our saved map with localization mode but first, we have to edit some parameters in mapper_params_online_async.yaml file.

```
odom_frame: odom
 map_frame: map
 base_frame: base_footprint
 scan_topic: /scan
 mode: localization #mapping
 map_file_name: /home/robomealmate/robomealmate/simulated
 # map_start_pose: [0.0, 0.0, 0.0]
 map_start_at_dock: true
```

First, convert the mode to localization then identify the map file path set to start at dock true, and we can specify a starting position if needed, let's run the SLAM toolbox again and see the result.

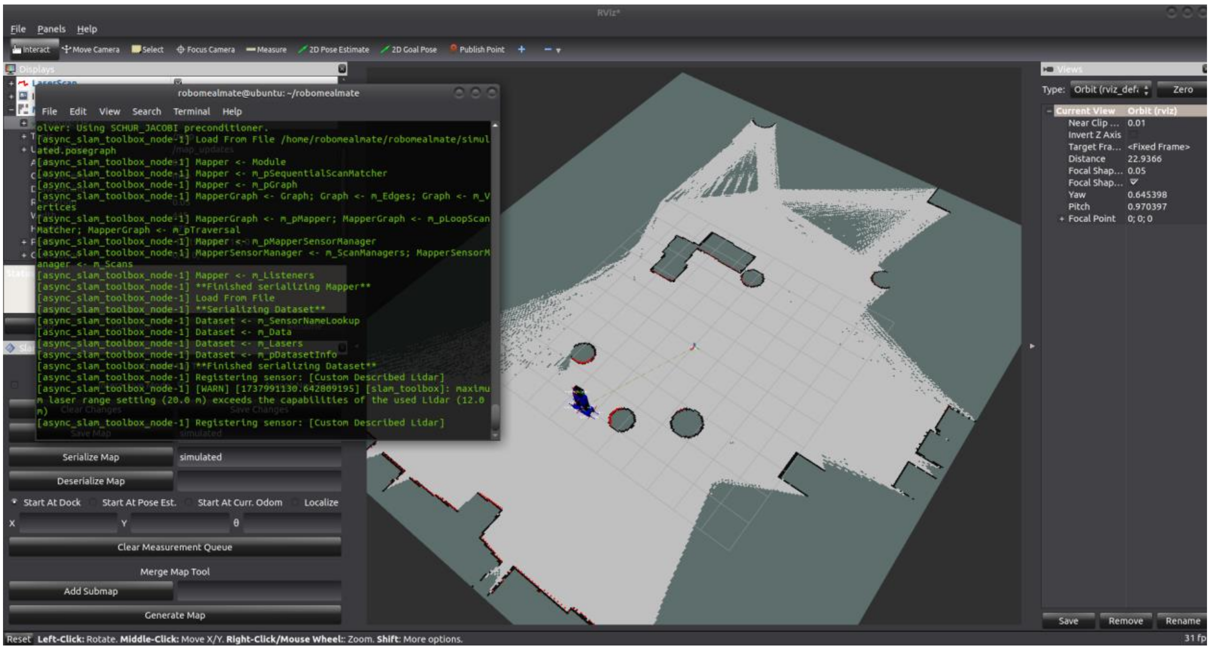


Figure 3.49 localization mode

As we see from Figure 3.49 our map is loaded and the simulated robomealmate is localized in it.

We are ready to move to the next step make our simulated robomealmate navigate autonomously in the virtual environment.

3.2.5 Navigation

Navigation is where we want to plan and execute a safe trajectory from the initial pose to the target pose.

In this section we will go through:

1. ROS nav2 stack concept.
2. Understanding navigation on robomealmate.
3. Set up the configuration file.
4. Navigation method 1 (live navigation).

3.2.5.1 ROS nav2 stack concept

To understand nav2 stack in ROS we must be familiar with some terms and concepts.

- Behavior trees
  It is a hierarchical structure made of nodes that represent tasks or behaviors, organized in a tree-like structure, used for high-level task control and decision-making, it manages the flow of navigation tasks for example, compute path tasks or follow path tasks.
- Planner server
  The task of a planner is to compute a path to complete some objective function, it uses A*, and or Dijkstra to compute the path.
- Controller server
  It executes velocity commands based on the planner to drive the robot.
- Cost maps
  There are two types of cost maps, global cost map covers the entire map for global path planning, and local cost map covers the area around the robot for obstacle avoidance, it has 3 layers, static layer represents unmovable objects like walls, inflation layer adds a buffer zone around the obstacle to ensure safe navigation.

Let us dive through a scenario, suppose we need robomealmate to go to (x,y) from (x0,y0), the user sends a navigation goal the goal is processed by the behavior tree, and then the global planner computes an optimal path from the current position to the goal position with avoiding known obstacles, the local planner generates velocity orders to the controller server to follow the global path while avoiding dynamic obstacles in real time, if the robot stuck or detects an issue, recovery actions are triggered, for example replanning or rotating, when robomealmate reaches the goal a success result send back.

3.2.5.2 Understanding navigation on robomealmate

The first information we are going to need to be able to navigate is an accurate position estimate, because if we don’t actually know were we are at any point in time how do we figure out where we trying to go, SLAM is going to provide this position estimate for us, the second thing we need awareness of the obstacles that are around our robot and this information can usually come from two sources or sometimes both combined, the first of those is a map so we have got a SLAM system that has already produced a map for us, we can use that map as a basis of navigation so the same obstacles we are using for localization before now become something to avoid it, this is most straightforward with the grid SLAM type of map, where we take our occupied cells add a buffer around them and that is our obstacles, the second one is generate a map on the fly using sensor data in our case lidar, this will prevent our robot from bumping into things which is good, but it means cant plan whole trajectory from the start it kind of needs figure things out on the fly, so the ideal world is where we combine both of these, we have an initial map help us plan our trajectory and then

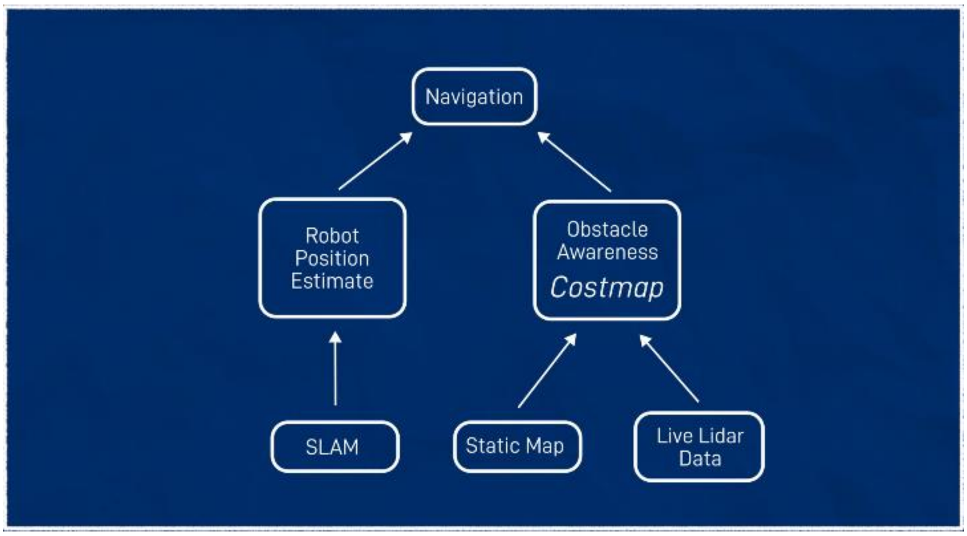


Figure 3.50 robomealmate navigation diagram

use live lidar data to update this with any new obstacles or things change along the way, all the information will be stored in the cost map.

3.2.5.3 Set up the configuration file

This file is a bit complicated so we will divide it into multiple configurations but in the same file, the file name is nav2_param.yaml and it will contain the configuration for the nav2 stack, it will be divided into the following:

1. Locale costmap configuration.
2. Global costmap configuration.
3. Planner server configuration.
4. Controller server configuration.

3.2.5.3.1 Locale cost map configuration

The local cost map is a grid map representing the robot's immediate surroundings, dynamically updated using sensor data.

```
local_costmap:
  local_costmap:
    ros__parameters:
      update_frequency: 5.0
      publish_frequency: 2.0
      global_frame: odom
      robot_base_frame: base_link
      use_sim_time: True
      rolling_window: true
      width: 3
      height: 3
      resolution: 0.05
      robot_radius: 0.3
      plugins: ["voxel_layer", "inflation_layer"]
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      voxel_layer:
        plugin: "nav2_costmap_2d::VoxelLayer"
        enabled: True
```

```yaml
        publish_voxel_map: True
        origin_z: 0.0
        z_resolution: 0.05
        z_voxels: 16
        max_obstacle_height: 2.0
        mark_threshold: 0
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"
      static_layer:
        map_subscribe_transient_local: True
      always_send_full_costmap: True
  local_costmap_client:
    ros__parameters:
      use_sim_time: True
  local_costmap_rclcpp_node:
    ros__parameters:
      use_sim_time: True
```

- update_frequency: the rate (in Hz) at which the local costmap is updated with sensor data in our configuration 5 times per second.
- publish_frequency: the rate (in Hz) at which the costmap is published to topics for visualization in our case 2 times per second.
- global_frame: the reference frame for the costmap. odom is used for a dynamic, robot-relative frame to account for movement.
- robot_base_frame: the robot's physical reference frame. In our case base_link.
- rolling_window: enables a moving window centered on the robot, rather than a static map. This is ideal for local navigation.
- width & height: dimensions (in meters) of the local costmap we set it to be 3m * 3m.
- resolution: grid resolution in meters per cell resolution of 0.05 means each cell represents 5 cm.
- robot_radius: the robot's radius in meters helps define safe clearance around the robot and our robot radius is 30 cm.
- plugins: specifies the costmap layers to load a voxel layer and an inflation layer are used.
- cost_scaling_factor: determines how rapidly the cost decreases as the distance from an obstacle increases. Higher values create sharper gradients.
- inflation_radius: the maximum distance (in meters) from an obstacle where inflation occurs.
- plugin: defines the layer type as a voxel layer for 3D obstacle representation.
- enabled: enables the voxel layer.
- publish_voxel_map: publishes the voxel grid for debugging and visualization.
- origin_z: the Z-coordinate of the costmap's base.
- z_resolution: vertical resolution 5 cm per voxel.
- z_voxels: The number of voxels along the Z-axis, allowing for a 3D map height of z_resolution * z_voxels = 0.05 * 16 = 0.8m.
- max_obstacle_height: obstacles above 2m are ignored.

- mark_threshold: minimum number of hits required for a voxel to be marked as an obstacle.
- observation_sources: Specifies the sensor used for obstacle detection it's a LiDAR sensor (scan).
- topic: the ROS topic to subscribe to for laser scan data.
- max_obstacle_height: obstacles above this height (2m) are ignored.
- clearing: enables the removal of obstacles from the costmap when they are no longer detected.
- marking: enables marking detected obstacles on the costmap.
- data_type: specifies the type of sensor data.
- static_layer: used for incorporating a static map into the costmap.
- map_subscribe_transient_local: ensures the subscription to static map data persists across node lifecycle changes.

For more information about costmap configuration see the resources. [2].

3.2.5.3.2 Global Costmap Configuration

```
global_costmap:
  global_costmap:
    ros__parameters:
      update_frequency: 1.0
      publish_frequency: 1.0
      global_frame: map
      robot_base_frame: base_link
      use_sim_time: True
      robot_radius: 0.3
      resolution: 0.05
      track_unknown_space: true
      plugins: ["static_layer", "obstacle_layer", "inflation_layer"]
      obstacle_layer:
        plugin: "nav2_costmap_2d::ObstacleLayer"
        enabled: True
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"
      static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
        map_subscribe_transient_local: True
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      always_send_full_costmap: True
```

- track_unknown_space: it allows the costmap to represent and track unknown space.
- static_layer: it loads the entire static map of the environment.
- obstacle_layer plugin is also included here but is not present in the local costmap, which uses the voxel_layer for 3D obstacle representation.
- plugin: "nav2_costmap_2d::StaticLayer": loads the static map for global navigation.

- map_subscribe_transient_local: ensures reliable subscription to the static map topic, even for transient or late-joining nodes.

Below are the differences between local and global costmap:

Feature	Local Costmap	Global Costmap
Purpose	Local path planning	Global path planning
Frame	odom	map
Update Frequency	Higher	Lower
Coverage	Immediate surroundings	Entire map
Window Size	Small	Full map size
Plugins	voxel_layer, inflation_layer	static_layer, obstacle_layer, inflation_layer
Dynamic/Static	Dynamic (rolling window)	Static (entire map)
Sensors	Real-time	Static map + sensors

Table 3.2 differences between local and global costmap

For more information about costmap configuration see the resources. [2] .

3.2.5.3.3 Planner Server Configuration

```
planner_server:
  ros__parameters:
    expected_planner_frequency: 20.0
    use_sim_time: True
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: false
      allow_unknown: true
```

- expected_planner_frequency: the frequency in Hz at which the planner is expected to be called 20.0 Hz means the planner is expected to generate plans up to 20 times per second.
- "nav2_navfn_planner/NavfnPlanner": NavfnPlanner is a grid-based planner that uses Dijkstra's algorithm by default (or A* if enabled), it generates a path from the robot's position to the goal using the global costmap.
- tolerance: specifies the tolerance (in meters) for achieving the goal position, if the robot is within 5 cm of the goal, the planner considers the goal reached, and it can be adjusted as needed.

For more information about planner server configuration see the resources. [3].

3.2.5.3.4 Controller server configuration

```
controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    controller_plugins: ["FollowPath"]

    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
```

```
debug_trajectory_details: False

# Velocities
min_vel_x: 0.0
max_vel_x: 2.5           # Increase to 0.7 or 1.0 if hardware permits
min_vel_y: 0.0
max_vel_y: 0.0            # 0 for diff-drive
max_vel_theta: 7.0
min_speed_xy: 0.0
max_speed_xy: 2.5         # match max_vel_x for diff-drive
min_speed_theta: 0.0

# Accelerations
acc_lim_x: 3.5
decel_lim_x: -3.5
acc_lim_theta: 6.0
decel_lim_theta: -6.0

# Trajectory sampling
vx_samples: 20
vtheta_samples: 20
sim_time: 1.5
linear_granularity: 0.05
angular_granularity: 0.025
transform_tolerance: 0.2

# Tolerances
xy_goal_tolerance: 0.25
trans_stopped_velocity: 0.2

# Critic configuration
critics: ["RotateToGoal", "Oscillation", "BaseObstacle",
          "GoalAlign", "PathAlign", "PathDist", "GoalDist"]
BaseObstacle.scale: 0.05
PathAlign.scale: 32.0
PathAlign.forward_point_distance: 0.1
GoalAlign.scale: 24.0
GoalAlign.forward_point_distance: 0.1
PathDist.scale: 32.0
GoalDist.scale: 24.0
RotateToGoal.scale: 32.0
RotateToGoal.slowing_factor: 1.5
RotateToGoal.lookahead_time: -1.0
short_circuit_trajectory_evaluation: True
stateful: True
```

It generates velocity commands to drive the robot along the planned global path.

- FollowPath: it represents a local planner for trajectory following.
- DWBLocalPlanner: is the Dynamic-Window Approach-based local planner.
- min_vel_x: the minimum forward velocity for the robot in meters per second.
- max_vel_x: the maximum forward velocity in meters per second.
- max_vel_theta: the maximum angular velocity in radians per second for rotation.
- acc_lim_x: maximum forward acceleration in meters per second squared.
- decel_lim_x: maximum deceleration in meters per second squared.
- acc_lim_theta: maximum angular acceleration in radians per second squared.
- decel_lim_theta: maximum angular deceleration in radians per second squared.
- xy_goal_tolerance: the distance tolerance (in meters) to the goal.
- critics: A list of cost function critics that evaluate trajectories based on different criteria, critics guide the robot to generate safe, efficient, and goal-aligned trajectories.

- ✓ RotateToGoal: ensures the robot faces the goal before driving.
- ✓ Oscillation: penalizes oscillatory behaviors.
- ✓ BaseObstacle: avoids obstacles in the local costmap.
- ✓ GoalAlign: ensures alignment with the goal.
- ✓ PathAlign: ensures alignment with the path.
- ✓ PathDist: minimizes the distance from the path.
- ✓ GoalDist: minimizes the distance to the goal.

For more information about the controller server see the resources. [4], for the DWB controller to see the resources [5].

Now we are ready to test the navigation.

3.2.5.4 Navigation (live navigation)

Live navigation means using sensor data for map creation and same time for costmap creation, adding the costmap for unknown areas while adding them to the man map.

Launching steps:

1. Launch simulation file for robomealmate.
2. Launch slam toolbox.
3. Launch navigation by the following command:

```
1.   ros2 launch nav2_bringup navigation_launch.py
    params_file:=src/robomealmate/config/nav2_params.yaml use_sim_time:=true
```

note that we passed our param file as a parameter.

Below is a video showing the navigation process, note that in order to play the video click on it.

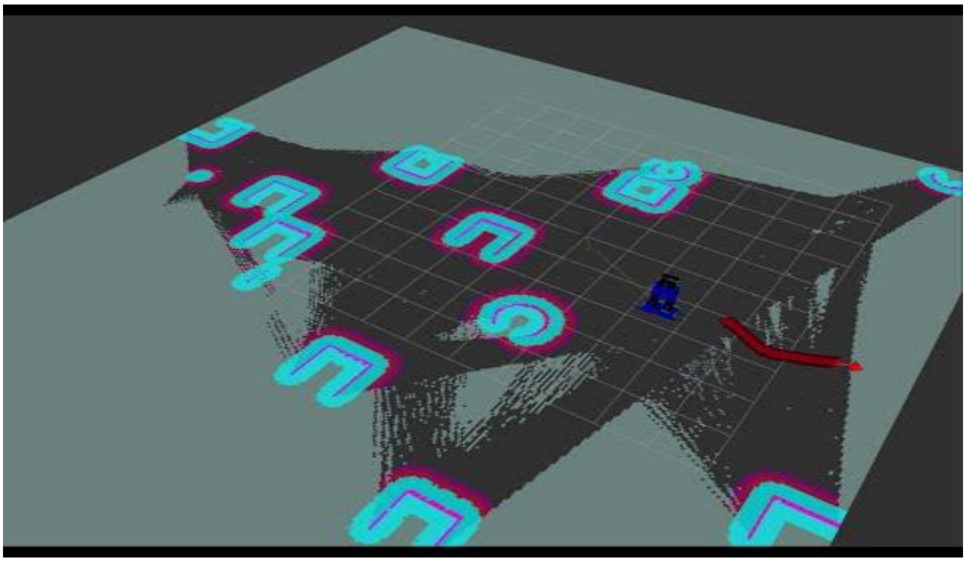


Figure 3.51 robomealmate simulated navigation video

Now after we simulated robomealmate successfully we can assemble our hardware and build our physical model.

3.3 Hardware assembly

Since we have two kinds of controllers, high-level controller (Raspberry Pi 4), and low-level controller (Arduino Mega) this section will be divided as follows:

1. Low-level controller assembly: in this subsection, we will show the connection and the components of the low-level controller.
2. High-level controller assembly: in this subsection, we will show the connection and the components of the high-level controller.
3. Power supply and buttons: in this subsection, we will show the battery connections and button connections.

3.3.1 Low-level controller assembly

Our work here will be divided into:

1. Motors, encoders, and motor driver connections.
2. Ultrasonics connections.
3. Bluetooth HC-05 and RGB LED strip connection.

3.3.1.1 Motors, encoders, and motor driver connections

The simple form to make a motor spin is to apply an appropriate supply voltage but we have no way to change the speed or direction and no way to control it remotely, so we have to add a motor driver, part of the reason is we can't just plug a motor straight into the controller if that motor require a relatively high voltage and higher current compared to what the pins on the controller chip can handle.

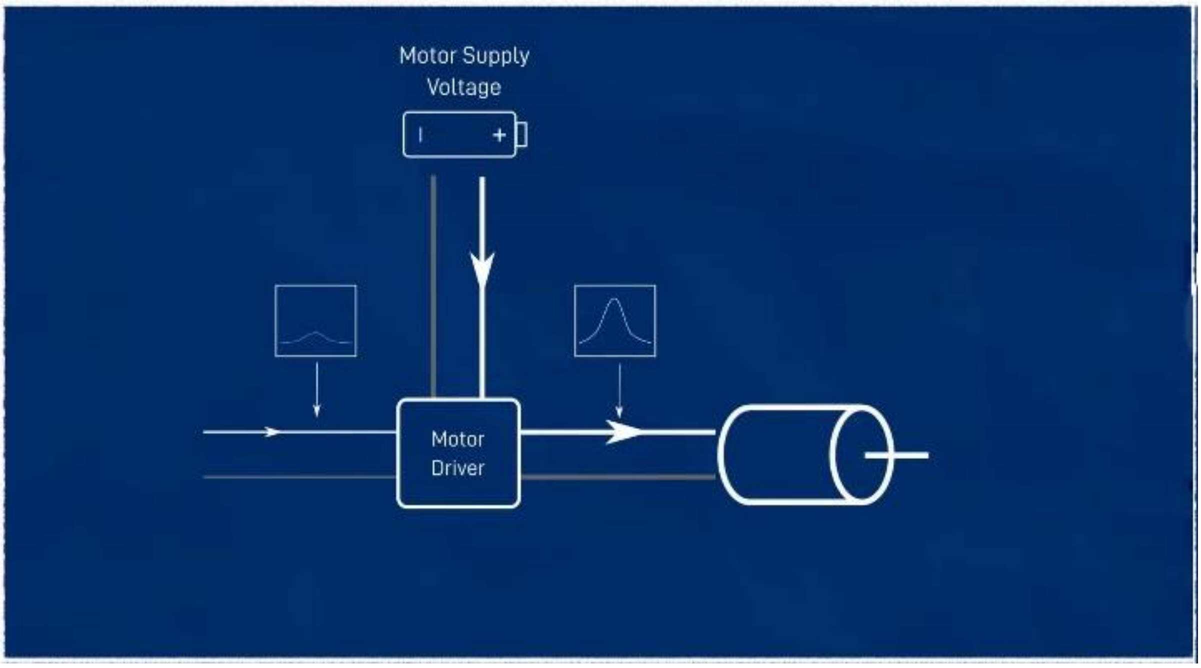


Figure 3.52 motor driver concept

Instead, we use a motor driver this little board takes a low voltage low current signal from the controller and uses the power supply to amplify it creating a higher voltage higher current to drive the motors.

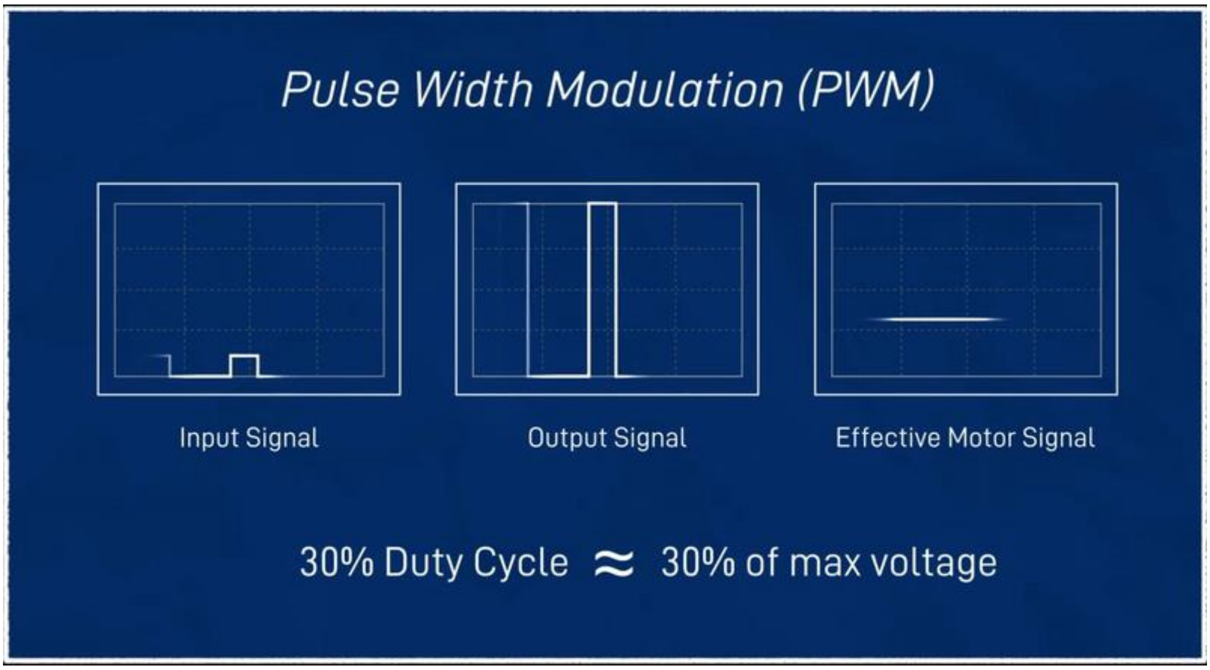


Figure 3.53 PWM concept

This signal is usually PWM (pulse with modulation), a series of fast pulses you average out over time, the ratio of on-time to off-time called the duty cycle determines the percentage of the total voltage that a motor effectively sees, basically the shorter the pulses is the slower the motor will go, the longer the pulse is the faster the motor will go, and if the pulses are on 100 of the time then the motor is going to be going full speed.

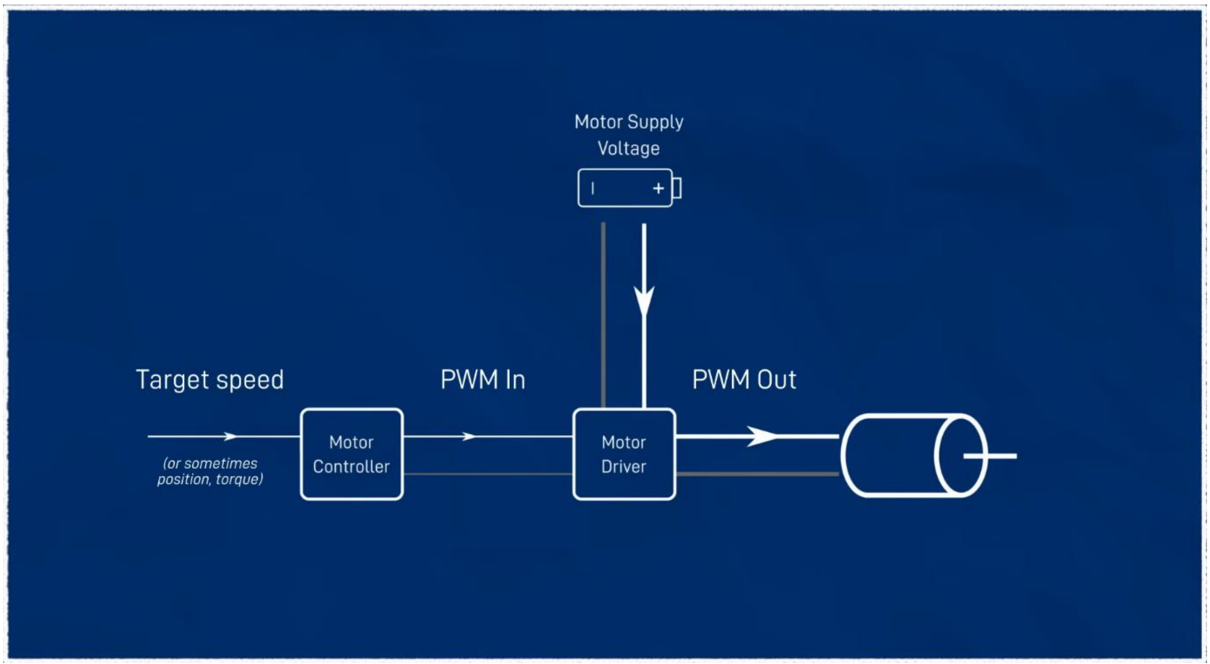


Figure 3.54 motor controller concept

Now we need a way to generate that PWM input signal for that we use the controller, the motor controller is a bit smarter its input is in a more practical format maybe a target speed or a target position then it calculates the appropriate signal to send to the driver to get the motor to go the right speed.

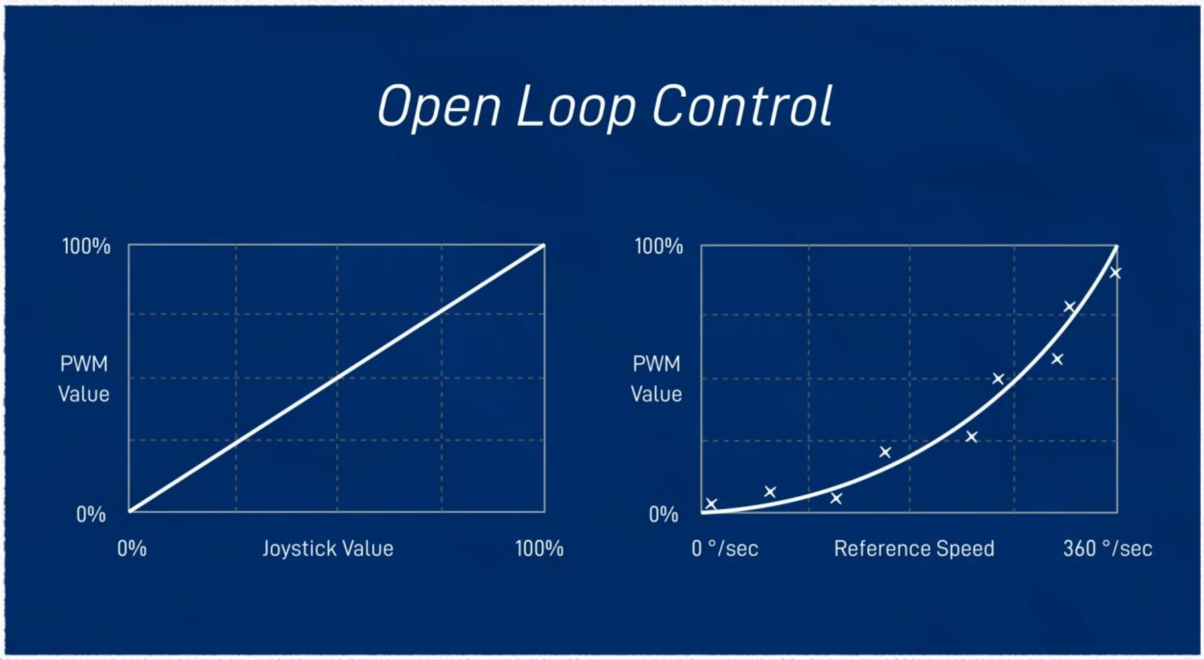


Figure 3.55 open loop control concept

The simplest kind of motor controller is an open loop controller this is where we have some kind of predefined map between the input signal and the output signal, if the robot is ultimately being driven by a human this can be as simple as taking the operator's joystick signals and mapping 0 percent to 0 percent and 100 percent to 100 percent, but if we want to start getting a bit more automated we can try come up with a function that maps the requested input speed in whatever unit maybe degrees per second to the appropriate PWM value, and we can figure out what this function is by testing a bunch of input values measuring the speed we get out and fitting an equation to the result, also this approach is really simple and might be good enough for some applications it tends to fall down pretty quickly, the appropriate speed to drive a signal function can change depending on a whole lot of factors the load that the robot under the battery charge or even if the motors going forwards versus backward, a human operator can naturally compensate these with slight changes.

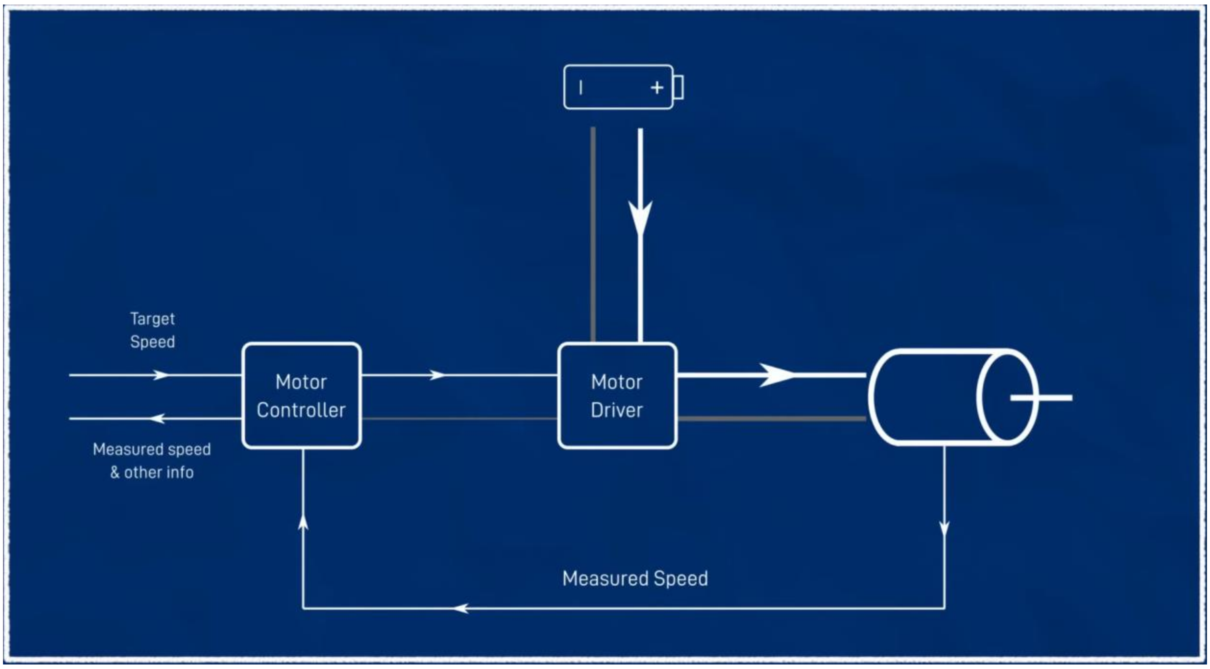


Figure 3.56 closed-loop control concept

but if we want to do robust automation we need to be able to command the motors to go an exact speed, to achieve that we need a way to measure the actual of the motor live and feed that back into the motor controller, so that we can adjust the input until we are getting the speed exactly right, and this is called closed-loop control or feedback control, so we look at how far we are away from where we want to be and adjust that signal accordingly if we are far away we make big changes if we are not we make small changes.

We got motor encoders in order to achieve closed-loop control. The encoder is a little wheel on the back of the motor that spins around. It will send signals back to the controller that we can use to calculate the speed.

As we mentioned before robomealmate is a bit heavy it's between 10kg to 15 kg to hold that weight we are using 4 DC motors with gearboxes for higher RPM, the RPM of these motors is 330, and the motors will attach to the base of the robot, motor A and C on the left side, motor B and D on the right side.

The motors need 12V and 100 milliamps in no load and on load need 1.5 – 2 amp, so we can’t use regular H-bridge L298N because it limits the out but current to 2 amp and it not enough to operate 2 motors, so the solution is to use IBT_2 driver, it can drive a current of 43A, but the problem is we cant use one to control 2 motors with a different signal so we will use 4 of it, one for each motor.

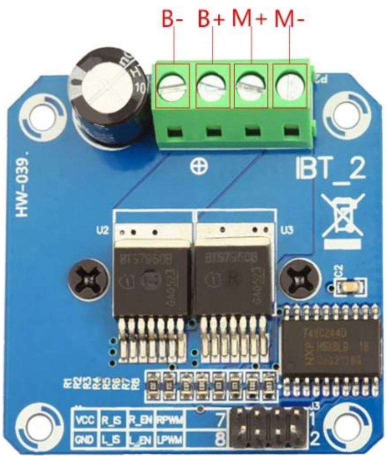


Figure 3.57 IBT_2 motor driver

Pin number	Pin name	Description
1	VCC	+5v input
2	GND	Ground input
3	R_IS	Forward current alarm output
4	L_IS	Backward current alarm output
5	R_EN	Forward movement enable
6	L_EN	Backward movement enable
7	RPWM	Forward pulse with modulation
8	LPWM	Backward pulse with modulation
-	M-	Motor negative polarity
-	M+	Motor positive polarity
-	B+	Power supply positive
-	B-	Power supply negative

Table 3.3 IBT_2 pin diagram

The motor driver will be connected to Arduino Mega as follows:

Motor attached to	Driver pin	Arduino Mega pin
Motor A	RPWM	Pin 4 (PWM)
	LPWM	Pin 5 (PWM)
	R_EN	Pin 26
	L_EN	Pin 27
Motor B	RPWM	Pin 6 (PWM)
	LPWM	Pin 7 (PWM)
	R_EN	Pin 28
	L_EN	Pin 29
Motor C	RPWM	Pin 8 (PWM)
	LPWM	Pin 9 (PWM)
	R_EN	Pin 30
	L_EN	Pin 31
Motor D	RPWM	Pin 10 (PWM)
	LPWM	Pin 11 (PWM)
	R_EN	Pin 32
	L_EN	Pin 33

Table 3.4 motor drivers --> Arduino Connections

Now we have to connect the motor wire to the drivers, note that the IBT_2 drivers are polarity sensitive and we can know the wire's polarity of the motors by applying a voltage using a battery if the motor spins forward the wire connected to the positive panel of the battery is positive and the negative is negative, if spins backward reverse is correct, after identifying the polarities connect each motor to its drive, by this we have the open-loop control and we are ready to set up the close-loop control.

Closed-loop control means we need a feedback speed from the motors to the controller we can solve this problem using encoders, encoder is mounted in the back of the motor, and as the motor spins the magnetic field is going to past the little hall effect sensors and they all send a signal that goes up and down in pulses back to the controller, for each revolution of the small disk we will going to get the signal go up and down a certain number of times, note that our motors have a gearbox which means each revolution of the motor it might be hundreds revolution of the small disk, the controller converts the encoder signal into a counts and by checking this count regularly for example lets say 10 times per second, we will be able to calculate the motor speed in count per second, then if we know how many counts there are in a full revolution we can convert that counts per second speed into rpm or degrees per second or radians per second whatever we want, below is the connection of the encoders with the Arduino.

Pin name	Description
VCC	+5v input
GND	ground
A	Channel A
B	Channel B

Table 3.5 Encoder pins

Encoder	Encoder pin	Arduino pin
Motor A Encoder	Channel A	Pin 2 (Interrupt Pin)
	Channel B	Pin 22 (Digital Pin)
Motor B Encoder	Channel A	Pin 3 (Interrupt Pin)
	Channel B	Pin 23 (Digital Pin)
Motor C Encoder	Channel A	Pin 18 (Interrupt Pin)
	Channel B	Pin 24 (Digital Pin)
Motor D Encoder	Channel A	Pin 19 (Interrupt Pin)
	Channel B	Pin 25 (Digital Pin)

Table 3.6 Encoder --> Arduino connections

To get good efficiency we need to know the total counts per revolution, we could figure this out by making the motor spins a known number of times and then dividing the final count by the number of times we rotated it to get the counts per revolution.

We rotate the motor 20 times and we get 26390 counts.

$$\frac{count}{times} = count\ per\ revolution\ = \frac{26390}{20} = 1319.5 \approx 1320\ count\ per\ revelution$$

We can integrate that number into equations to calculate the speed we will discuss it later in detail.

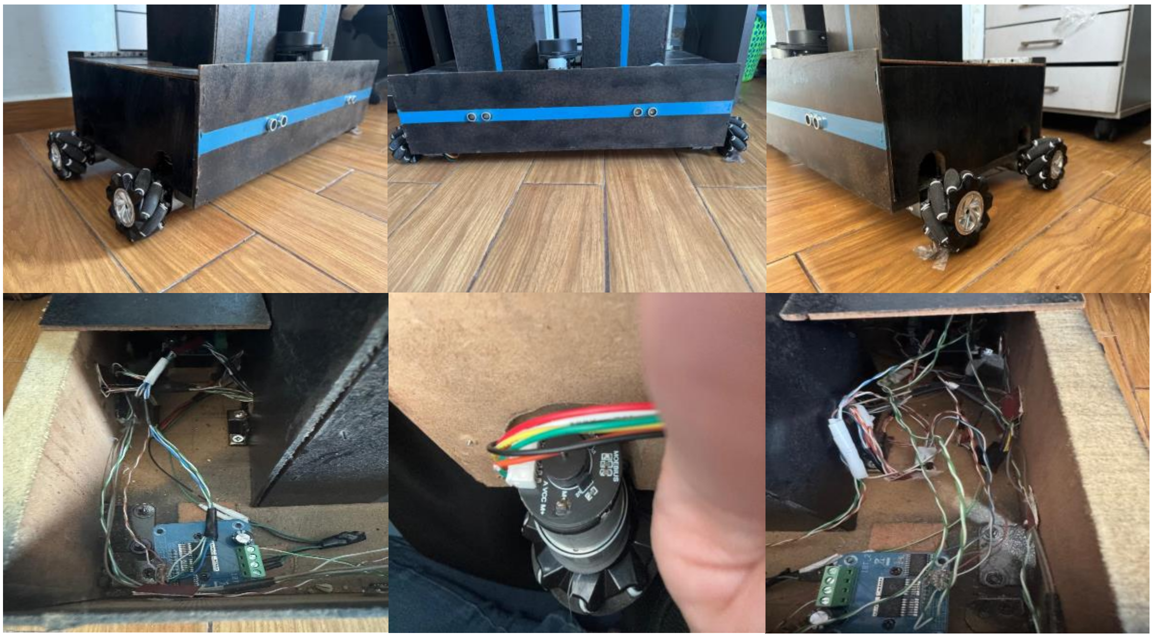


Figure 3.58 motor, motor drivers, and encoders

As we see from Figure 3.58 motors and their attachments are in the base of the robot.

3.3.1.2 Ultrasonic connections

In robomealmate, the ultrasonic sensors work like emergency brakes, and the lidar handles everything about distances and safe movement, but the lidar is mounted at the top of the base, and it is above the ground about 20cm, so there a 20 cm not detected and it may be an obstacle there so the ultrasonic will detect this obstacle and stop the motors and go backward to force movement algorithms to find a new path, cover front and back we are going to use 4 ultrasonics sensors (HC-S04).

It works by using sound waves to measure distance. It emits high-frequency sound pulses (ultrasonic waves) and calculates the time it takes for the waves to bounce back after hitting an object. Based on this time, the sensor determines the distance to the object.

$$Distance = \frac{speed\ of\ sound\ \times time}{2}$$

The division by 2 is needed because the sound travels to the object and back.

pin name	Description
VCC	+5 v input
GND	Ground
Trig	Trigger signal input to send an ultrasonic pulse usually at 40 kHz
Echo	Output signal returns HIGH when an echo is received.

Table 3.7 ultrasonic pins

Where is mounted	Ultrasonic pin	Arduino pin
Front left	Trig	34
	Echo	35
Front right	Trig	36
	Echo	37
Rear left	Trig	38
	Echo	39
Rear right	Trig	40

	Echo	41

Table 3.8 ultrasonic connections

3.3.1.3 Bluetooth HC-05 and RGB LED strip connection.

In order to debug the process happening in the Arduino controller we added a Bluetooth model as a serial monitor and commander and also added an RGB LED strip as a visual indicator.

Component name	Pin name	Arduino pin
HC-05	TX	RX3
	RX	TX3
RGB	D in	Pin 13

Table 3.9 Bluetooth and RGB led strip connections

3.3.2 High-level controller connections

1. Camera
   We will use Raspberry Camera v2, raspberry pi 4 comes with CSI (Camera Serial Interface) on the board, in order to connect the camera there is plastic clip on the CSI port, open it insert the blue side of the ribbon facing the HDMI ports, press the clip

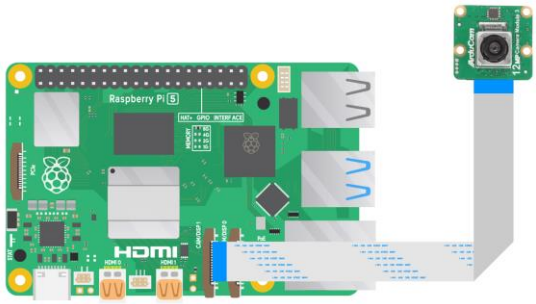


Figure 3.59 raspberry camera connection

   back down to secure it.
2. LCD touch screen: we are using 7 inch touch screen we will connect it using HDMI caple, on end is micro HDMI connected to one of two micro HDMI ports on raspberry and the other one is normal HDMI connected to the screen.
3. Lidar A1 is connected using micro USB for the lidar and USB for raspberry.

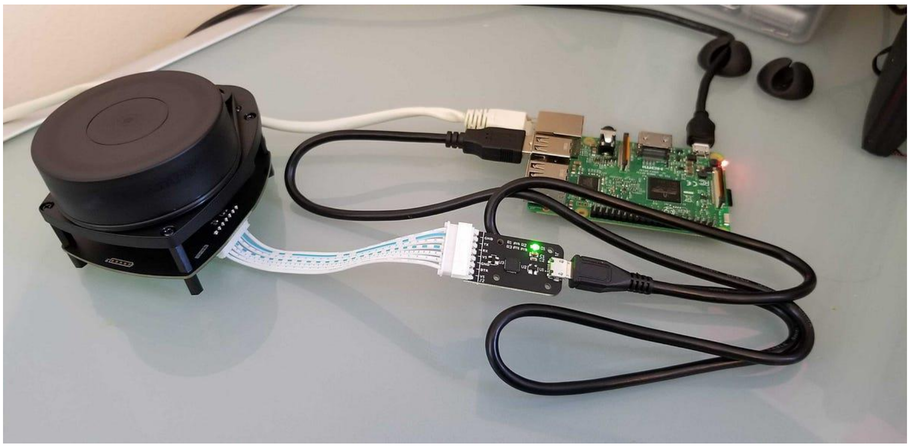


Figure 3.60 lidar connected to raspberry

4. Raspberry ←→ Arduino: in order to get the sensors and motors data from the Arduino we connected the Raspberry with Arduino over USB.

3.3.3 Power supply and buttons

To power every thing up we have to know 2 important things:

1. Voltage for each component
2. Current consumption for each component then the total current consumption.

We have two main kinds of components, components need 5v, and components need 12v, so we will divide our components voltage supply into components with 5v supply and components with 12v supply, then get the current consumption for each one to find the total current needed.

Component name	Voltage supply (v)	Current consumption(A)
Raspberry pi	5V	2
Lidar A1		0.5
Lcd screen		1
Lidar		0.45
Arduino Mega		0.05
Bluetooth HC-05		0.03
Ultrasonic sensors		0.015
Encoders		0.05
IBT_2 motor driver		0.01
DC motors	12V	2
RGB LED strip		0.6
Cooling fan		0.1

Table 3.10 components voltage and current

$$total\ current = component_1\ current \times number\ of\ components + component_2\ current \times number\ of\ components + \cdots + component_n\ current \times number\ of\ components$$

$$total\ current = 2 + 0.5 + 1 + 0.45 + 0.05 + 0.03 + (0.015 \times 4) + (0.05 \times 4) + (0.01 \times 4) + (2 \times 4) + 0.6 + 0.1 = 13.03A$$

We will separate the power of the low-level controller and its components from the power of the high-level controller and its components, we will use a 12V 9A battery and buck converter to step down the voltage to 5V, and a power bank with 5V 3.5A for powering the raspberry and its components , the buck converter out will power the Arduino and 5V components, and the 12v out will power up the 12V components.

We added an emergency button for the power going to the motors. In case something goes wrong, this button will cut off the power and stop the motors.

There are 3 buttons for power on/off one for powering the Raspberry and its components, one for the Arduino and sensors, one for the cooling fan

We added a 12V charging port for charging 12V battery.

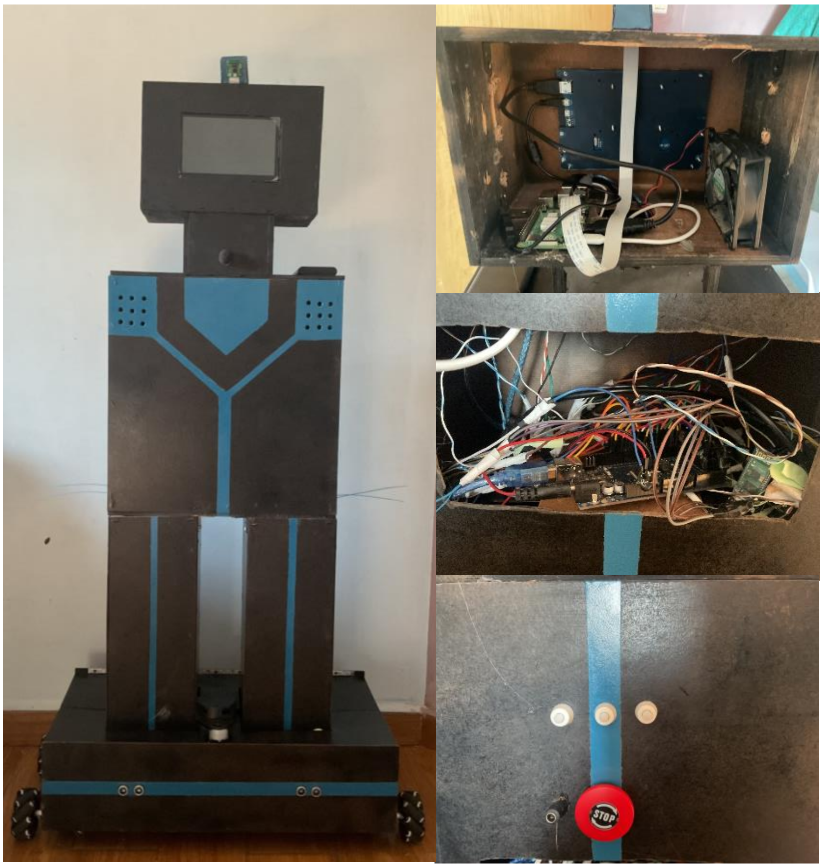


Figure 3.61 robomealmate

3.4 Real Robomealmate Implementation & ROS Integration

In this section we will apply the programming and integration done with simulation with the real robomealmate, first, everything we did before was on a computer and now we will use a raspberry pi with Ubuntu server 20.04 and Ubuntu Mate desktop, one of the greatest ROS features it shares the topics over the network that means whatever I run on raspberry I will see it on the computer, we created a GitHub repo and pushed all the code we wrote it early on the GitHub and cloned it on the raspberry on order to easy transfer the code between the devices.

To make things clear everything we have done before on simulated robomealmate applicable on real robot by just set the use_sim_time to false, but there is other important thing we have to do, in simulated robomealmate we used gazebo plugin to drive it but now we have to drive the real robomealmate and the plugin will not help us, to solve this problem we will use ROS2 control package,ROS2 control is powerful and complex so to understand it we must take it step by step, so we will divide this topic into:

1. ROS2 control understanding
2. ROS2 control integration with simulated robomealmate
3. ROS2 controlling real robomealmate
4. Running robomealmate.

3.4.1 ROS2 control understanding

There are many different kinds of actuators and interfaces out there and so many different control methodologies that we use to solve problems with them, but within this huge universe of options there is actually quite a lot of commonality. Unless we have some kind of standardized system, if we are just writing custom controllers and drivers every time we start a new project then we end up rewriting things and wasting lots of time solving problems that people solved many times before.

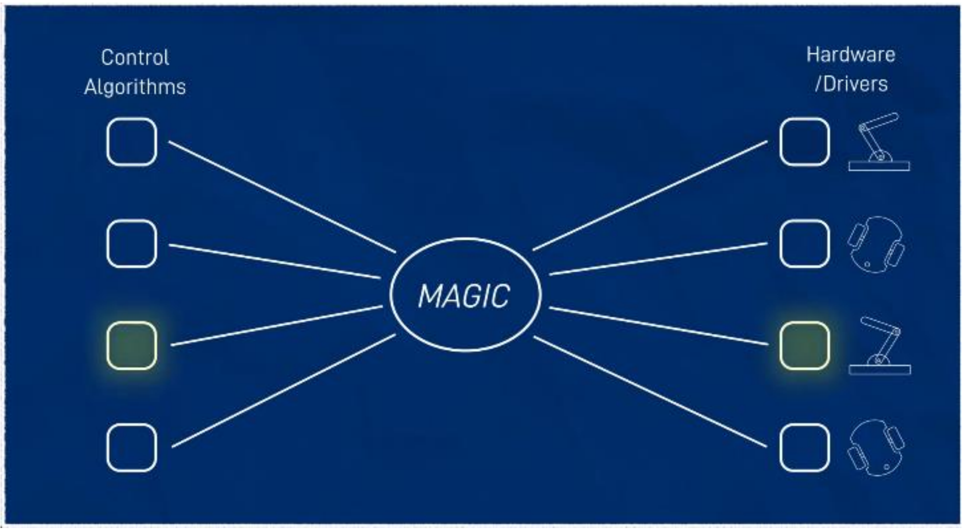


Figure 3.62 framework concept

instead of what we want is a framework where we can write the drivers for all the different hardware platforms out there and the algorithms for the different control methodologies and how they speak a common language so we can choose whatever we need to solve a particular problem, we need a method to be fast should be no delay between the controller and the hardware interface and that where ros2 control com in.

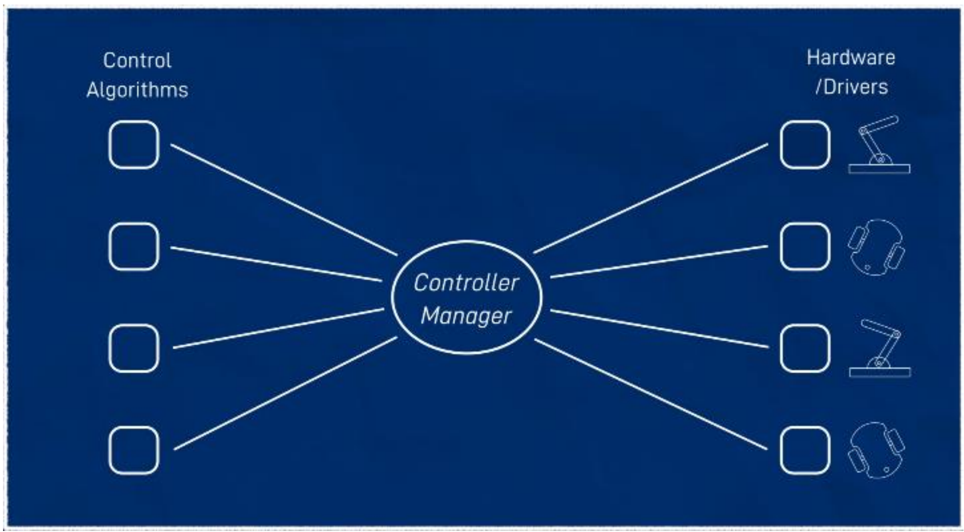


Figure 3.63 controller manager

at the center we have something called controller manager its going to find all these bits of code for our drivers and our controllers and link them together, to achieve this it uses a plugin system so each of these little things isn't running its own executable it's just a library that is loaded up at run time by the controller manager and it's got a set of functions that link into the system.

To use ros2 control with the hardware the first thing we going to need is a hardware interface, that's a little bit of code that speaks to the hardware and exposes it in the ros2 control standard and we will write this code later in this section.

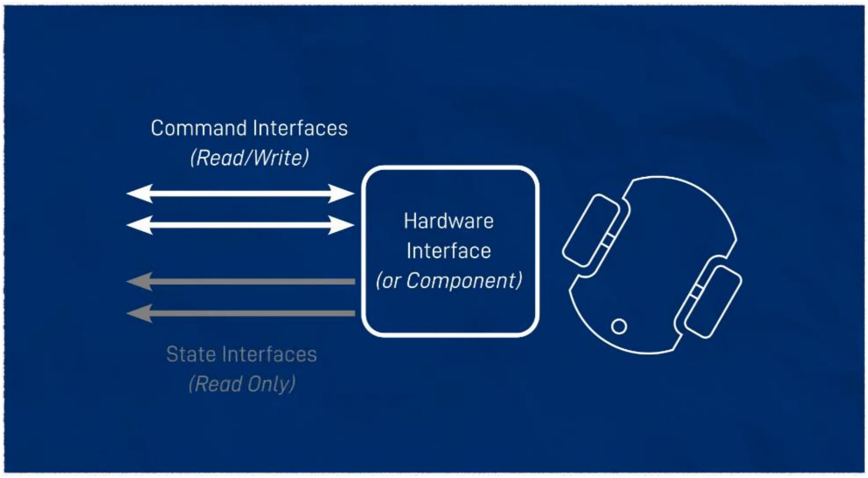


Figure 3.64hardware interfaces

All we need to understand is the way that it represents the hardware and that's through command interfaces and state interfaces, command interfaces are things that we can control in the robot where state interfaces are things that we can only monitor, for example, robomealmate motors we can only control the velocity so that would be a command interface, but we are also able to measure the position with the Encoders as well as the velocity so the position and the velocity are going to be state interfaces, so for robomealmate, we have 4 command interfaces to control the speed for each wheel, and 8 state interfaces for the position and velocity of each wheel.

At this level of abstraction, there is no understanding of what the robot is, it could be a differential drive robot or could be a robot arm that has only to joints that are for some reason only velocity controlled, ros2 control doesn't know doesn't care all it knows about are these twelve interfaces.

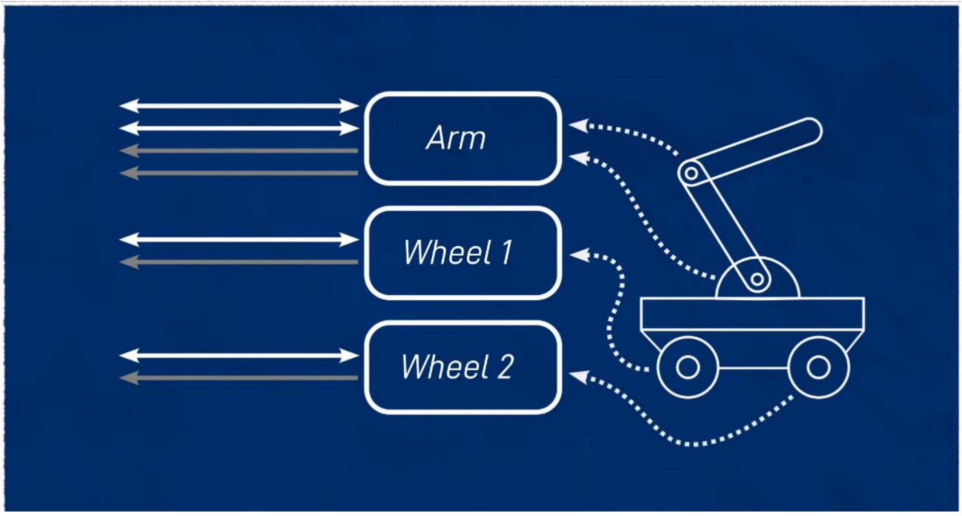


Figure 3.65 multiple hardware interface

one robot might have multiple hardware interfaces each of which can have multiple command and state interfaces this could be because we have got a robot arm mounted on the mobile base each with separate control systems or could be each of the motors is fully independently controlled and we want them to have their drivers.

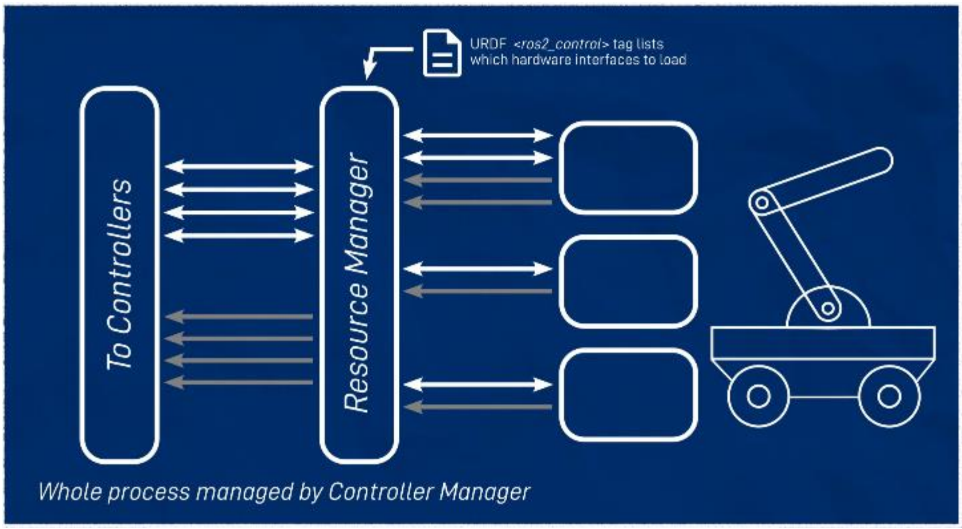


Figure 3.66 hardware side

To deal with this the controller manager uses something called the resource manager which gathers up all the different hardware interfaces and exposes them together so that the controllers just see a big list of command and state interfaces, how does the controller manager or the resource manager know about these hardware interfaces, because of it's tied so closely to the hardware of the robot we put it into the URDF using a ros2 control tag we will see it in detail later, but to be clear inside the tag we tell ros2 control the name of the hardware interface plugin to load and also which joints of the URDF it's going to be talking to but what about the controllers?

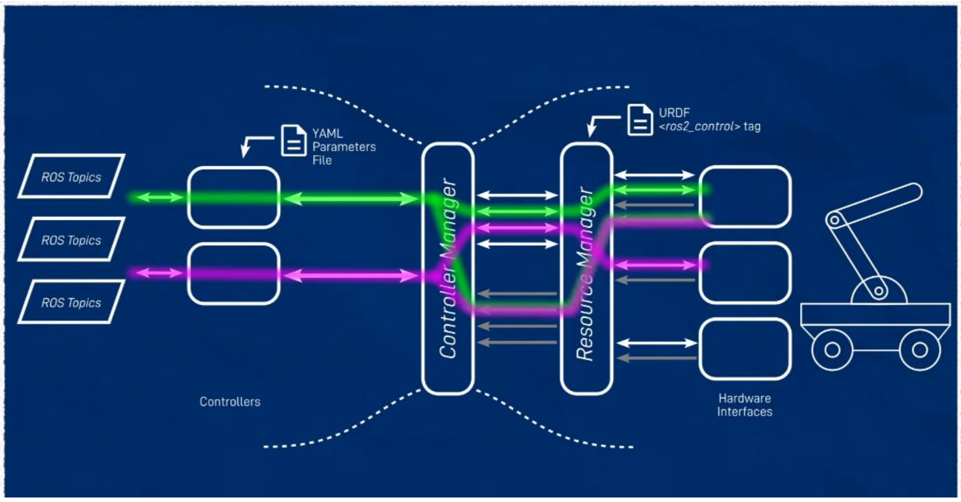


Figure 3.67controller side

Controllers are the way we interact with ros2 control, so in on the one end they are going to be listening for an input on a ROS topic, that can be something like joint position in a robot arm or a mobile robot body velocities they are going to take this input they are going to calculate what the appropriate motor positions and speeds and that sort of things are and they are going to send these to the hardware interfaces that have been exposed by the resource manager.

The hardware interfaces need to be designed specifically for individual hardware the controllers are going to be designed specifically for different robot applications and since there are a few pretty common applications the ros2 controllers package provides controllers that should cover most people's needs, since robomealmate is a differential drive robot we will be using diff drive controller, but if we have something different we might use other controllers.

The controller manager's job is to take the controllers that it's been asked to load and match them up with the right command and state interfaces that the resource manager is exposing, to set up the controllers we write a yaml file with the various parameters we need and pass that into the controller manager from there we can tell it to start and stop the controllers as needed and just like we going to have multiple1 hardware interfaces in one robot also we can have multiple controllers in one robot as long as they both don't want to clam the same resources they are not trying to command the same interface they can share state interfaces since their read-only.

We have got hardware interfaces on the one side and the controllers on the other side how do we run this thing, there are two different kinds of ways that we can run a controller manager, firstly we can write our own node and have a controller manager running inside that, secondly we can just use a node that's provided for us and this what we going to do, but either way we are going to provide it the information about the hardware interfaces usually via URDF and the information about the controllers usually via yaml file, either way once the controller manager is up and running we want a way to interact with it so that we can check the hardware interfaces and start and stop the controllers ..etc, there is a few different ways to do this the controller manager has some services that we can call to access this functionality there is also the ros2 control command line tool that's makes the service calls a little bit easier and also there is some nodes that handles some of those key function that we can run.

Note that one of the features of ros2 control is that it can control both the simulated robot and real robot, for more information about ros2 control see the resources [6].

That's the theory, let's put it into practice, first we will test it on simulation then we will dive into real robot.

3.4.2 ROS2 control integration with simulated robomealmate

We earlier created a file called gazebo controller we going to replace this file with ros2 control so we will create a file called ros2_controller.xacro and then comment the including of the gazebo controller we will back to it later and include ros2_control.xacro in the robot.urdf.xacro file.

```
        <ros2_control name="GazeboSystem" type="system">
            <hardware>
                <plugin>gazebo_ros2_control/GazeboSystem</plugin>
            </hardware>
            <joint name="front_left_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10.0</param>
                    <param name="max">10.0</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
            <joint name="front_right_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10.0</param>
                    <param name="max">10.0</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
            <joint name="back_left_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10.0</param>
                    <param name="max">10.0</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
            <joint name="back_right_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10.0</param>
                    <param name="max">10.0</param>
                </command_interface>
                <state_interface name="velocity"/>
```

```
            <state_interface name="position"/>
        </joint>
    </ros2_control>
```

The plugin tag have the name of the plugin that we are going to use this is a pice of code that's been installed separately and registered with ros that tells it how to talk to the gazebo simulation just like talking to real hardware so the name of the plugin we going to use is gzebo_ros2_control/Gazebo- System and also we need to tell it about the different joints that its going to be controlling, we used joint tag and identified the joint by its name, then we need to specify the interfaces we are going to have, so we used command interface tag for velocity and state interface for velocity and position, then we set the limits for velocity, then repeated the process for the four wheels , so we have got 4 joints we can control we can control the velocity for each of them through command interface and we can get the velocity and the position of them through a state interface, ros2 control done now we need to add the gazebo plugin it different than plugin we used for ros2 control, it tells the gazebo to use ros2 control.

```
    <gazebo>
        <plugin name="gazebo_ros2_control" filename="libgazebo_ros2_control.so">
            <parameters>$(find robomealmate)/config/controller.yaml</parameters>
        </plugin>
    </gazebo>
```

The plugin is libgazebo_ros2_control.so, this plugin do a lot of things, it sets a bunch of things on the gazebo end and it actually contains its own controller manager so we don't need to start our own one, that controller manager is able to get the URDF through the robot state publisher but it will need to know where our controller yaml is, so we used parameters tag to identify it, lets create the controller.yaml file.

```
controller manager:
  ros__parameters:
    use_sim_time: true
    update_rate: 30
    diff_cont:
      type: diff_drive_controller/DiffDriveController
    joint_broad:
      type: joint_state_broadcaster/JointStateBroadcaster
diff_cont:
  ros__parameters:
    publish_rate: 30.0
    base_frame_id: base_link
    Wheels_per_side: 2
    left_wheel_names: ['front_left_wheel_joint', 'back_left_wheel_joint']
    right_wheel_names: ['front_right_wheel_joint', 'back_right_wheel_joint']
    wheel_separation: 0.64
    wheel_radius: 0.07
    use_stamped_vel: false
```

we are using the controller in simulation so use sim time parameter will be true later we will change that, then we need to set up the controllers, we name the controller then we give it a type, so the diff_cont will be differential drive controller, and the joint broad will be joint state broad caster, actually joint broad controller does not control any thing but robot state publisher needs the joint state topic to be published on order for it to get the wheel transforms correct, so we need to set up the parameters for the controllers, diff_cont need to know the wheels joint names, number of wheels per side, wheels separation, and wheel radius, know we can launch our simulation.

The following video shows the process of launching the simulated robomealmate and control it using ros2 control.

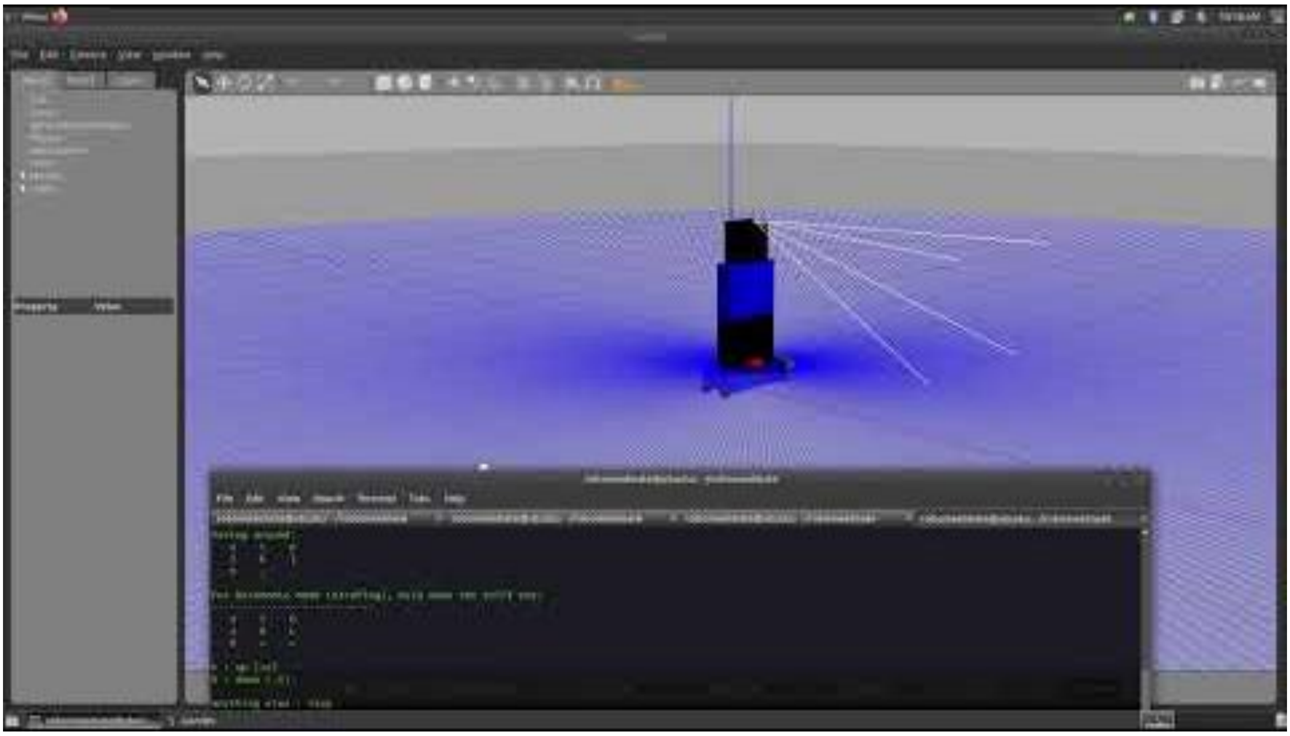


Figure 3.68 video for ros2 control integration for simulated robomealmate

To spawn the controllers automatically we will add two nodes into simulation file

```
    diff_drive_spawner = Node(
        package="controller_manager",
        executable="spawner.py",
        arguments=["diff_cont"],
    )
    joint_broad_spawner = Node(
        package="controller_manager",
        executable="spawner.py",
        arguments=["joint_broad"],
    )
    #Launch!
    return LaunchDescription([
        rsp,
        gazebo,
        spawn_entity,
        diff_drive_spawner,
        joint_broad_spawner
    ])
```

By this the diff_cont and the joint broad will load automatically.

3.4.3 ROS2 control integration with real robomealmate

First lets recap the concept:

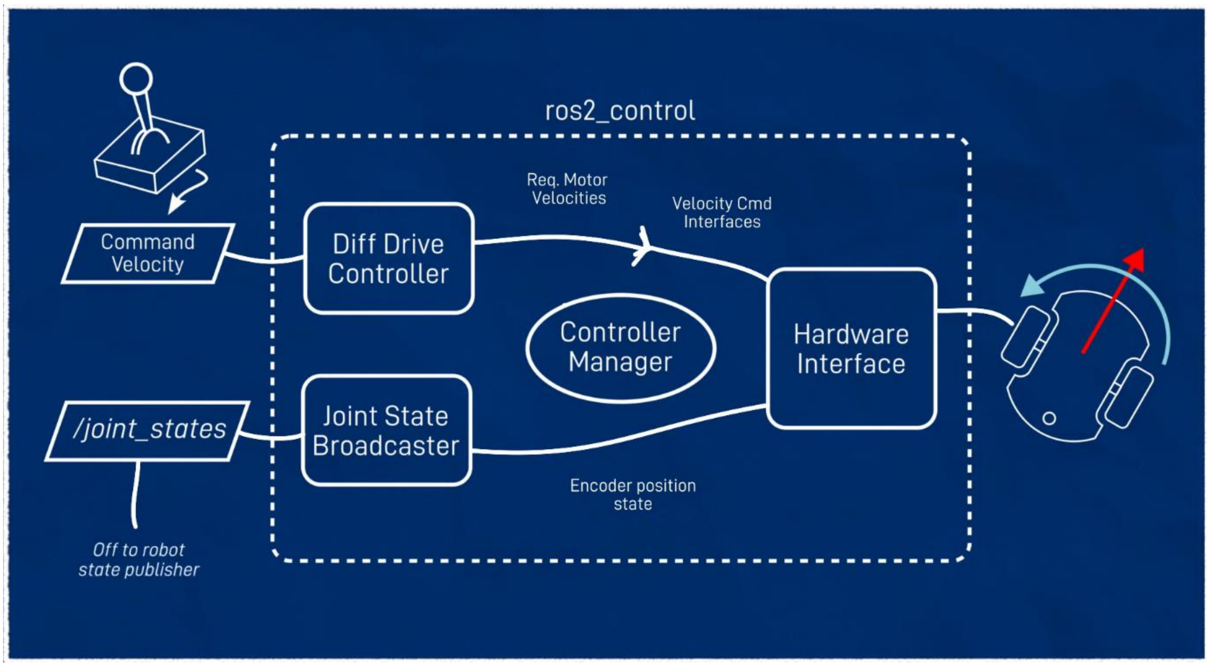


Figure 3.69 ros2 control concept recap

At the one end, we have got the physical robomealmate its got four drive wheels each of which is velocity controlled, at the other end we have got a body velocity request that's the velocity that we want the robomealmate to be moving through space this could come from an operator or something like nav2 stack, this particular robomealmate's motion is quite constrained in fact the only thing we can do is to move forward and backward in x and rotate around z so through the input is going to be ROS topic of type twist that is six-dimensional velocity and we are going to be using two dimensions, linear x, and angular z. we call this the command velocity and we are going to use ros2 control to link up the command velocity to the actual motors, and this little echo system consist of three main parts, in the middle we have got the controller manager then in the left hand side we have got the differential drive controller and on the right hand side we have got the robomealmate's hardware interface, the controller manager is a node provided for us by ros2 control libraries the diff drive controller is also provided for us as a plugin in ros2 controllers, and hardware interface is a plugin that's we need to write to meet our hardware, the diff drive controller will convert the command velocity into abstract wheel velocities and then the hardware interface is going to turn abstract wheel velocities into motor hardware commands the controller manager just sets in the middle and links two up together, in addition to this we have got another controller called joint state broadcaster it reads the motor encoder position provided by the hardware interface and publishes them to joint states topic for robot state publisher to use to generate its transforms, in the previous step we set up the controllers using a gazebo controlled controller manager and a gazebo hardware interface where now we want to be running our own controller manager using our own hardware interface and run it on robomealmate.

The first thing we need to set up is the hardware interface, so we will write the hardware interface and we will call it diffdrive_arduino, it expose the four command interfaces the velocity of each wheel and controls them using serial communication with the Arduino, it also expose the eight state interfaces that's the position and the velocity of each wheel and then the control will happened in the same diff drive controller we used on simulated robomealmate.

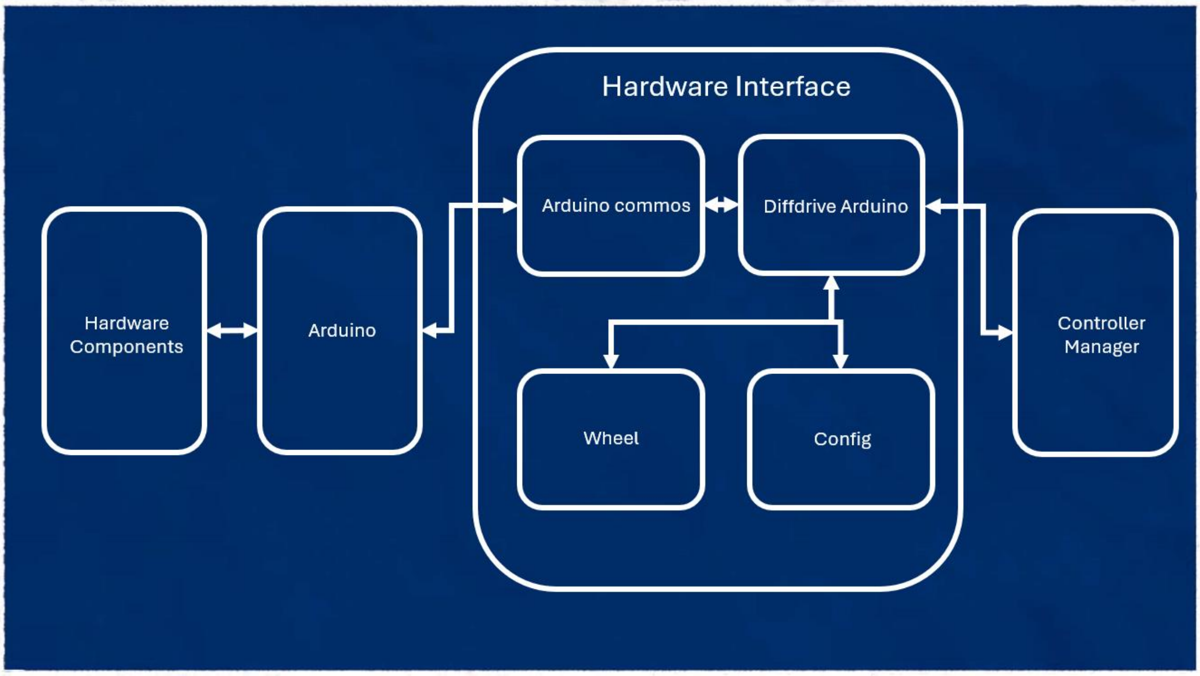


Figure 3.70 hardware interface

How our hardware interface will work, it will be divided into 2 main classes, Diffdrive Arduino and Arduino comms, Diffdrive Arduino will handle the communication with controller manager it will receive velocity commands and publish states, Arduino comms class will handle the communication with Arduino sends velocity commands and receive encoder data, the data will be send over serial communication, the Arduino controller will execute the commands received and apply the commands in actual hardware components, for organizing purpose we will add another class called wheel it will contains the necessary information about the wheels.

In order to write our hardware interface plugin we should know what are things we need to know about our hardware, and we will use this information to set up the parameters of the plugin like we did in ros2 control file with gazebo plugin, so our robomealmate hardware interface will contain a config file hold the hardware information and these information are depends on the interfaces we need, we need command interface that controls the four wheels and state interface that broadcast the transforms depending on encoders, these information should contains the information of connection with Arduino in order to make the interface talk to the hardware, we will write our plugin as a package these package contains two directory, include directory for header files and src directory for cpp files, our config file will be a header file in order to create an instance of it and organize our code.

The information will be divided into 3 main categories:

1. Wheels information for command interface.
2. Encoder information for state interface.
3. Arduino communication information.

The wheels information is the names of wheels joints and number of wheels in each side, encoder information is encoder count per revolution, Arduino communication information will be the port name and we will call device the baud rate and time out for the connection, so our config.h file will be as follows:

```
struct Config
{
  std::string front_left_wheel_name = "front_left_wheel";
  std::string front_right_wheel_name = "front_right_wheel";
  std::string back_left_wheel_name =  "back_left_wheel";
  std::string back_right_wheel_name = "back_right_wheel";
  float loop_rate = 30;
  std::string device = "/dev/ttyACM0";
  int baud_rate = 9600;
  int timeout = 1000;
  int enc_counts_per_rev = 1320;
};
```

These values are the default values.

In Arduino comms class there are 4 main functions:

```
  void setup(const std::string &serial_device, int32_t baud_rate, int32_t timeout_ms);
  void sendEmptyMsg();
  void readEncoderValues(int &motorA_enc, int &motorB_enc, int &motorC_enc, int &motorD_enc);
  void setMotorValues(int motorA, int motorB, int motorC, int motorD);
```

- Setup function: it will open a serial communication depending on the values of serial device, baud rate, and connection time out.
- Send empty massage function: it will send an empty massage to the Arduino to check the communication.
- Read encoder values: it will send a command to read the encoder values and send it back.
- Set motor values: it will send the motor PWM values to control the motors.

The information we should know about each wheel, wheel name, encoder value, velocity command, position, actual velocity, and radians per count:

```
  std::string name = "";
    int enc = 0;
    double cmd = 0;
    double pos = 0;
    double vel = 0;
    double rads_per_count = 0;
```

also we have to calculate radius per count in order to get the wheel angle and that implies to the wheel position, to convert encoder count to radius we have to apply the following equation:

$$rads\_per\_coutn = \frac{2\pi}{enc\_counts\_per\_revolution}$$

Explanation:

- 2π represents one full revolution in radians (since $360° = 2\pi\ radians$).
- Enc_counts_per_rev: is the number of encoder pulses (or counts) generated per full revolution of the motor shaft.

- Dividing 2π by enc_counts_per_rev gives the angle (in radians) that corresponds to a single encoder count.

To calculate wheel angle:

$$wheel\_angel \; = enc\_counts \times rads\_per\_count$$

So we have 2 functions in our wheel class:

```
    void setup(const std::string &wheel_name, int counts_per_rev);
    double calcEncAngle();
```

setup function to set the wheel information, calc angle to calculate the wheel angle.

The Diffdrive Arduino class will contain 7 functions:

```
  return_type configure(const hardware_interface::HardwareInfo & info) override;
  std::vector<hardware_interface::StateInterface> export_state_interfaces() override;
  std::vector<hardware_interface::CommandInterface> export_command_interfaces() override;
  return_type start() override;
  return_type stop() override;
  return_type read() override;
  return_type write() override;
private:
  Config cfg_;
  ArduinoComms arduino_;

  Wheel f_l_wheel_;
  Wheel f_r_wheel_;
  Wheel b_l_wheel_;
  Wheel b_r_wheel_;
```

- Configure function: it will load the information of our hardware from the xacro file and set up the config instance, then set up wheels and arduino instances depending on config instance data
- Start function: it will start the controller by send an empty massege to the arduino in order to ensure the connection.
- Read function: it will read the encoders values then calculate and store the position and the velocity of each wheel by:
  1. calculate the time difference to determine the time interval between updates
     $$\Delta t = new\_time - time$$
  2. read the encoders values to get the last encoder readings from the arduino.
  3. Store the previous position then calculate the position by calculating the wheel angle using the function provided by wheel class.
  4. calculate the velocity by applying the following equation:
     $$wheel\_velocity = \frac{wheel\_pos \; - previous\_pos}{\Delta t}$$
- export state interfaces function: it will export the position and velocity for each wheel.
- Export command interfaces function: it will set the cmd (command velocity) for each wheel depending on the data coming from diff drive controller.
- Write function: it will write the cmd values coming from diff drive controller to the actual motors using set motor values function.
- Stop function: it will stop the controller.

Now we have to set up our plugin in ros2 control file as we did with gazebo plugin but this time with our hardware interface plugin:

```
        <ros2_control name="RealRobot" type="system">
            <hardware>
                <plugin>diffdrive_arduino/DiffDriveArduino</plugin>
                <param name="num_wheel_pairs">2</param>
                <param name="front_left_wheel_name">front_left_wheel_joint</param>
                <param name="back_left_wheel_name">back_left_wheel_joint</param>
                <param name="front_right_wheel_name">front_right_wheel_joint</param>
                <param name="back_right_wheel_name">back_right_wheel_joint</param>
                <param name="loop_rate">30</param>
                <param name="device_name">/dev/ttyACM0</param>
                <param name="baud_rate">9600</param>
                <param name="timeout">1000</param>
                <param name="enc_counts_per_rev">1320</param>
            </hardware>
            <joint name="front_left_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10.0</param>
                    <param name="max">10.0</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
            <joint name="front_right_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10.0</param>
                    <param name="max">10.0</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
            <joint name="back_left_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10.0</param>
                    <param name="max">10.0</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
            <joint name="back_right_wheel_joint">
                <command_interface name="velocity">
                    <param name="min">-10.0</param>
                    <param name="max">10.0</param>
                </command_interface>
                <state_interface name="velocity"/>
                <state_interface name="position"/>
            </joint>
        </ros2_control>
```

Note that the device name is the serial port name on raspberry pi and the baud rat

now we have to write the code for the arduino to handle ros requests, the code mainly will contain 2 functions:

```
hard_brake();
handleIncomingCommands(Serial,Serial3);
```

- Hard brake function: this function is responsible for detecting the obstacle under the lidar and adjust the robot position to be far from that object to give the chance for planning algorithm to find a new path.

- Handle incoming command function: this function is responsible for take the orders from ros2 control over serial and apply them to the actual hardware and send the feed back over serial to the Bluetooth serial monitor.

To see the full code go to robomealmate_controller.ino file.

3.5 Open AI and google cloud integration for Realtime conversation

In order to make robomealmate able to talk to people and take their orders we have integrate a python script uses chat GPT API for cleaver responses and google cloud for voice recognition and text to speech and speech services.

Let us understand the process

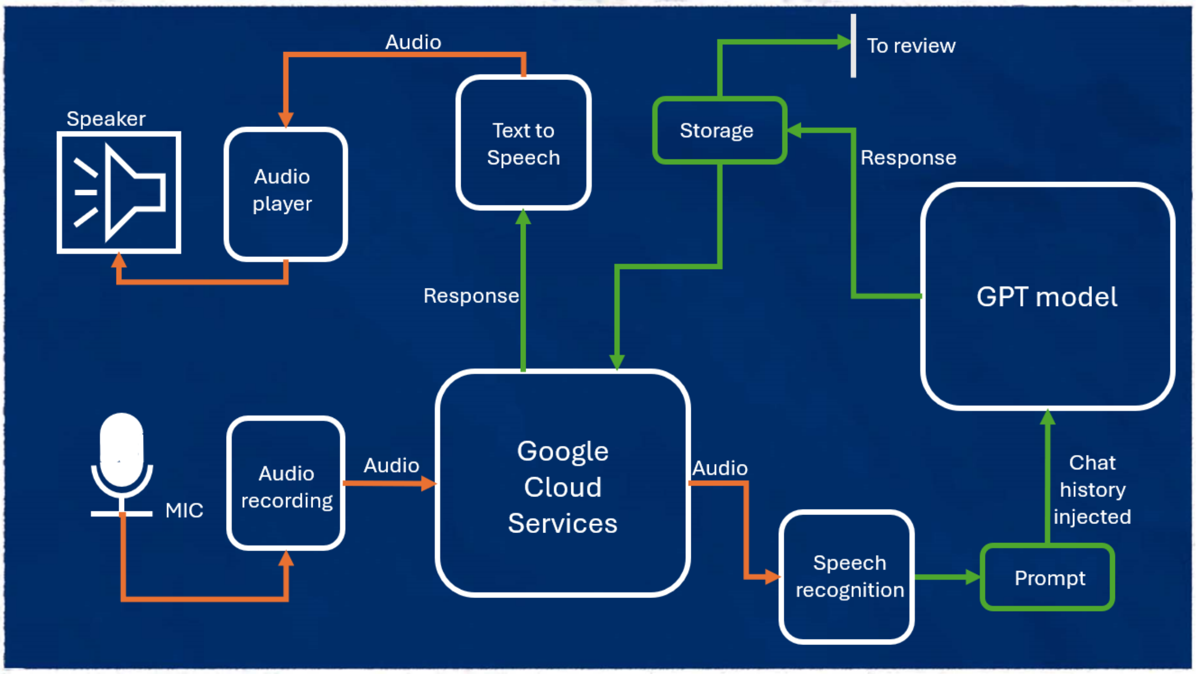


Figure 3.71 GPT and Google cloud services

The script will record the voice using the microphone then send it to google cloud services requesting speech recognition service the voice recognition service will convert the audio to text, the script will send that text as a prompt and inject chat history as a reference for responding the chat history will contains the information about robomealmate and the restaurant menu, the GPT model will generate a response depending on user requests and all conversation will be stored in a file in order to review orders, the response text will be sent to google cloud and request text to speech service, the text to speech service will return back the audio, and the script will play that audio and out it on the robomealmate speaker.

Note that the language and the voice can be changed to cover the users needs, and the chat history can be an instructions for the GPT model how it will behave, you can use any GPT model by specifying it but note that some GPT models are paid, by this script the robomealmate will be able to communicate with its environment using AI services.

We have applied all the functionalities of robomealmate, for resource code visit this link.

4 Standards, Specifications, and Constraints

4.1 Standards and Specifications

- Safety requirements for personal care robots (iso 13482:2014)
  For a robot interacting with humans in different environments, RoboMealMate fits with ISO 13482 guidelines for collision avoidance and emergency stop and user safety, ultrasonic sensors and LiDAR are integrated to provide a safe distance from customers, and emergency stop are installed on the robot body.
- Wifi communication standards
  We used the wifi integredted with raspberry and its standards is
    - Wi-Fi Standard: IEEE 802.11ac (Dual-band 2.4GHz and 5GHz).
    - Backward Compatibility: Supports 802.11a/b/g/n.
    - Frequency Bands: 2.4 GHz (2400 – 2483.5 MHz) and 5 GHz (5150 – 5850 MHz).
    - Maximum Data Rate: Up to 433 Mbps on 5GHz and 150 Mbps on 2.4GHz.
    - Security Protocols: Supports WPA3, WPA2-PSK, WPA-Enterprise, WEP.
    - Antenna Type: Integrated PCB antenna.
    - Modulation Techniques: OFDM, DSSS, QAM, PSK.
- Security and communication protocols
  İt uses ssh connection for comunicat and data transfer over local network it uses passwrod athentication encrypted with SHA-2 algorithim the trasformation port is 22 by default and uses TCP/IP protocol.
- CE Marking (EU Compliance for Electronic Devices)
  All electronic components (motors, Raspberry Pi, LiDAR,..etc.) were sourced with CE certification, ensuring electromagnetic compatibility and low-voltage safety.

Design Alternatives Considered:

- ROS2 and other frameworks: ROS2 was chosen over alternatives like MATLAB Robotics System Toolbox due to its open-source nature, modularity, and alignment with industry standards.

- LiDAR and Camera: LiDAR (RPLIDAR A1) was selected for its compliance with ISO 13849 safety standards, offering reliable obstacle detection in low-light conditions compared to vision-based systems.

4.2 Design and constrains

- Economy
  The total hardware cost around 850$ and requires cost effective components (e.g., Raspberry Pi over NVIDIA Jetson, The modular design allows for easy replacement of parts (such as motors and sensors) to reduce costs in the long run.
- Environment
  12V 7Ah battery and 10000mAh power bank were chosen to balance running time of about (4 hours) and energy efficiency, and motors were chosen with noise levels <50dB to avoid disturbing the restaurant atmosphere, all electronic components were chosen with CE certification, ensuring electromagnetic compatibility and low-voltage safety.

- Society
  The robot designed to be inclusive of most groups in society, the screen supports large fonts, and multiple interactions, and the sound unit supports multiple languages to help elderly or visually impaired users, processing voice data on-device has reduced reliance on cloud services, reducing the risk of data breaches, and the text conversation stored as unknown name just we store the order and table number.
- Politics
  The robot's neutral design and voice avoid cultural or gender biases, creating fair service.

5 Results and Analysis

5.1 Localization and mapping

The results shows that robomealmate is capable of generating a map identical to its environment, the lidar data is processed efficiently with the SLAM toolbox, and results show that robomealmate is capable of locating itself in the generated map the following video shows robomealmate reading the map and localize itself within that map.

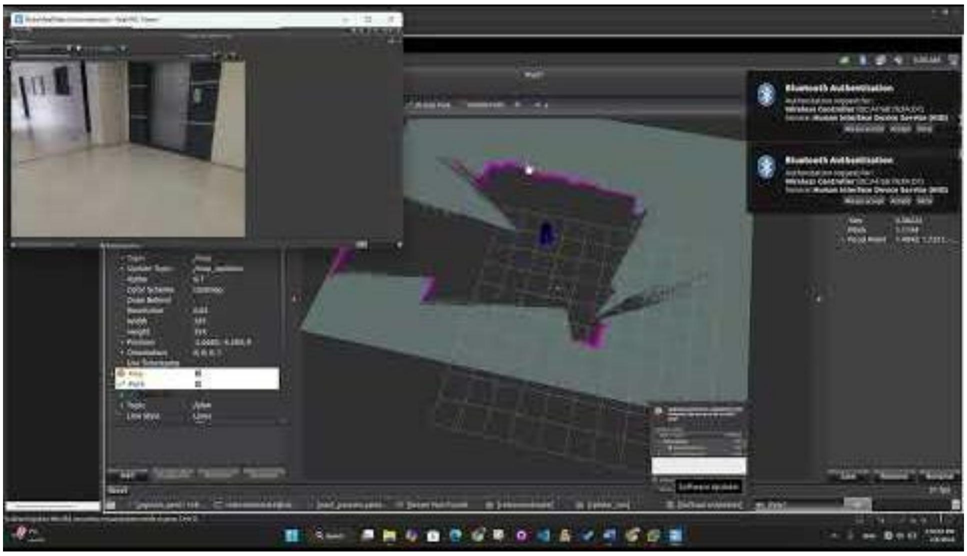


Figure 5.1 mapping and localization results

As we can see the details of the environment are taken perfectly and the generated map shows these details with variation in some situations but it accepted and serves our needs, and we can see form the robomealmate locating itself perfectly with the same orientation that’s mean the odometry and position calculations depending on the encoders is pretty good and the lidar plays a major role in this accuracy by using transforms and scans data.

5.2 Navigation

The results shows that robomealmate is capable of navigating to a goal position and it finds the path pretty well the following video shows the navigation of robomealmate from start point to goal point in different situations.

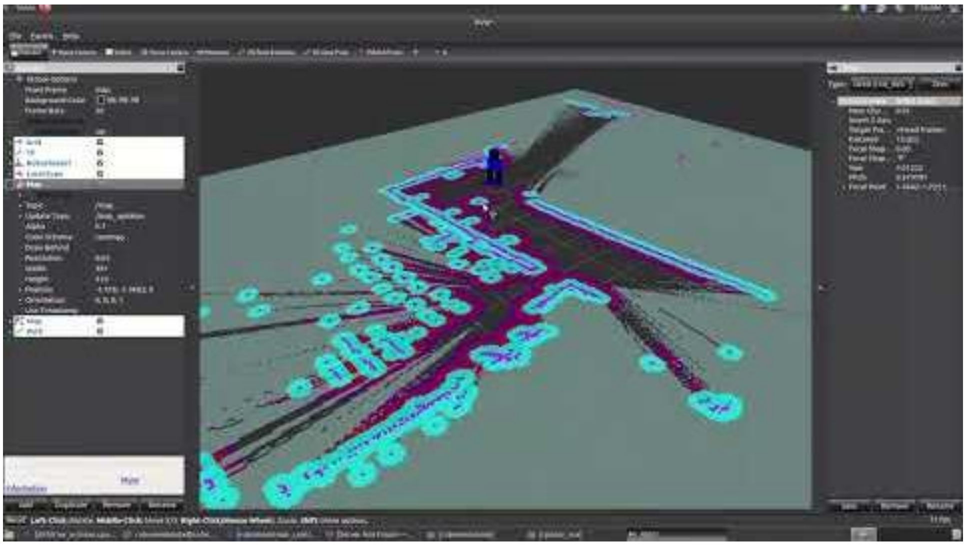


Figure 5.2 robomealmate navigation

As we can see in the figures robomealmate generated costmap while moving and detects the obstacles and it find a path to the specified goal and navigated to it while it avoiding the obstacles, but in some cases it the wheels being stuck and that because the weight we can adjust this by lighting the weight, and some cases it lagging and that because the low capability of raspberry pi for handling graphics, for now we can rerun the navigation node.

The simulated robot also shows the same result as we showed in section 3.2.4.3.

5.3 Realtime conversation and order taking

The results shows that robomealmate is capable of interacting with people and take the orders and store them in a file the following text is a conversation with robomealmate(beta).

```
[2025-02-07 19:46:06] User:  مرحبا بيتا
[2025-02-07 19:46:09] Beta:  مرحبًا! كيف يمكنني مساعدتك اليوم؟
[2025-02-07 19:46:35] User:  اريد ان اطلب مشروبا ساخنا
[2025-02-07 19:46:40] Beta: بالطبع! لدينا الخيارات التالية من المشروبات الساخنة:

 -  القهوة - $2.99
 -  الشاي - $2.49
 -  القهوة العربية - $3.99
 -  شاي بالنعناع - $3.49

ما الذي ترغب في طلبه؟ ولا تنسى إخباري برقم الطاولة.
[2025-02-07 19:47:16] User:  اريد ان اطلب الشاي بالنعناع
[2025-02-07 19:47:18] Beta:  تم تسجيل طلبك للشاي بالنعناع بقيمة $3.49. من فضلك، ما هو رقم الطاولة الخاصة بك لأتمكن
  من إكمال الطلب؟
[2025-02-07 19:47:37] User:  رقم طاولتي 5
[2025-02-07 19:47:39] Beta:  شكرًا لك! تم تسجيل طلبك للشاي بالنعناع على الطاولة رقم 5. سيكون رقم طلبك هو 1. إذا كنت
  بحاجة إلى أي شيء آخر، فلا تتردد في إبلاغي. استمتع بوقتك!
```

Note that the robomealmate is capable of understanding people thoughts, taking there orders, and organizing them with numbers, it knows the menu of the restaurant, but sometimes there is a delay in capturing audio the reason behind the big load on Raspberry Pi, and sometimes to have to repeat your thought because of the noise around.

6 Discussion

The RoboMealMate project aimed to address critical challenges in restaurant service automation by integrating AI, robotics, and autonomous navigation. In this chapter, we will highlight on the project outcomes against its objectives, spotlight on his contributions, and discusses limitations and future directions.

6.1 Resolution of the Problem

The project successfully resolved several key issues identified in the restaurant industry by integrating OpenAI and Google Cloud APIs for speech recognition and natural language processing, RoboMealMate minimized manual entry errors and miscommunication, the AI-driven system accurately transferred and processed customer orders, storing them digitally for kitchen staff, the LiDAR-based SLAM implementation enabled accurate mapping and localization in dynamic environments, the robot demonstrated reliable obstacle avoidance and path planning, even in crowded environment, solving slow service caused by human waiters' mobility constraints, by automating tasks that require human resources, robomealmate reduces operational costs while maintaining the quality and availability of the service.

6.2 Contributions to the Fields

RoboMealMate improved the field of service robotics through several new contributions, unlike existing robots like Pepper or Servi, RoboMealMate integrates real-time conversational AI (GPT models) to deliver personalized customer experiences, it have multi language capability opens the way to accessibility in diverse restaurant settings, the modular hardware (Raspberry Pi, Arduino) and ROS2-based software architecture allow easy adaptation to different restaurant sizes and layouts. The total cost ($859) is significantly lower than commercial alternatives, making it viable for small-to-medium enterprises, and integrating LiDAR SLAM with ultrasonic sensors ensured robust obstacle detection at varying heights, solving blind spots in purely LiDAR-based systems.

6.3 Logical Implications of Results

The results shows that the feasibility of deploying AI-powered service robots in real-world hospitality environments, in tests conducted, the robot achieved 85% navigation success in crowded environments. However, performance deteriorated in highly dynamic scenarios (e.g., sudden crowd movements), highlighting the need for faster sensor fusion algorithms, and the Raspberry Pi's computing power sometimes caused delays in processing high-resolution sensor data, impacting real-time decision making.

6.4 Limitations

Although the project is promising, it has notable limitations, Although the wooden frame is lightweight, it lacks the durability needed for long-term use. Also, the motor power (12V) limits speed and load capacity, affecting performance in larger spaces, SLAM has difficulty in areas without clear landmarks, which reduces the accuracy of mapping, and relying on cloud-based APIs introduces latency and requires a stable internet connection, which may not be possible in all regions.

7 Conclusions and Recommendation

7.1 Summary of Key Results

The RoboMealMate project achieved its primary goals, demonstrating the feasibility of AI-powered robots in restaurant automation:

1. OpenAI and Google Cloud APIs have been successfully integrated to recognize speech in real-time, process multilanguage requests, and interact with customers in a human-like manner, reducing manual errors.
2. LiDAR-based SLAM technology enabled reliable navigation in dynamic environments, good path planning accuracy in mapped areas.
3. The modular design, built on Raspberry Pi and Arduino, brought the total hardware cost to $859, making it affordable for small and medium-sized restaurants.

7.2 Recommendations for Improvement

1. Use lighter weight materials like carbon fiber.
2. Upgrade the motors to handle more weight

7.3 actual conclusion

The flexibility of OS2R allowed for seamless integration between simulated and real-world components, confirming its suitability for robotics development, and the limited computing power of the Raspberry Pi highlights the need to allocate resources evenly between sensor processing and AI tasks.

7.4 Open problems

1. Noise effect on real time conversation.
2. Raspberry pi hardware limitations.
3. In some situations the navigation stops and that because lagging of raspberry pi

7.5 Future Work

1. Upgrade the high-level controller to NVidia jetson nano for better graphical handling and high load tasks.
2. Integrate depth camera for 3D mapping.
3. Enhance the Realtime conversation by integrate it with ros.

7.6 Final remarks

RoboMealMate embodies the transformative potential of artificial intelligence and robotics in hospitality. While the project is validating basic technical and operational concepts, addressing its limitations through targeted upgrades and research would open the door to broader commercial application. By prioritizing affordability, adaptability and user-centered design, RoboMealMate sets a precedent for next-generation service robots poised to redefine industry standards.

References

[1] Wikipedia, "List of moments of inertia," Wikipedia, 8 march 2021. [Online]. Available: https://en.wikipedia.org/wiki/List_of_moments_of_inertia. [Accessed 2025 january 24].

[2] n. stack, "nav2 stack documantation-costmap," 2023. [Online]. Available: https://docs.nav2.org/configuration/packages/configuring-costmaps.html.

[3] n. stack, "nav2 stack documantation-planner-server," 2023. [Online]. Available: https://docs.nav2.org/configuration/packages/configuring-planner-server.html.

[4] n. stack, "nav2 stack documantation-controller-server," 2023. [Online]. Available: https://docs.nav2.org/configuration/packages/configuring-controller-server.html.

[5] n. stack, "nav2 documantation-BWDController," 2023. [Online]. Available: https://docs.nav2.org/configuration/packages/configuring-dwb-controller.html.

[6] r. control, "ros2 control documentation," [Online]. Available: https://control.ros.org/rolling/index.html.