Tanta University

Faculty of Engineering

Window Cleaning Robot

"Robots and automation will eliminate many jobs, but we can harness them to improve quality of life." Bill Gates

GRADUATION PROJECT TANTA UNIVERSITY

MECHATRONICS DEPARTMENT January 2025

AUTHORS:

Mohamed Tarek ElDemerdash

Ahmed Wael Helmy

Mahmoud Samir Elsharkawy

Eman Ibrahim Elhendawy

Ahmed Mohamed Gaber

Ahmed Fotouh Zahran

Ahmed Mohamed Refaat

Haidy Mohamed Ramadan

Supervised By:

Dr. Ali Mamdouh

ABSTRACT

Manual window cleaning, especially for large or tall buildings, presents a significant challenge in terms of safety, cost, time, and consistency. Workers are exposed to dangerous heights and harsh weather conditions, increasing the risk of accidents. Additionally, manual cleaning is time-consuming, labor-intensive, and difficult to maintain consistency across various surfaces. There is a clear need for a solution that automates the process while maintaining high standards of cleanliness and ensuring the safety of the cleaning personnel.

The primary problem we aim to address is the lack of an efficient, automated window-cleaning solution that can operate on vertical surfaces, especially on skyscrapers and high-rise buildings. The goal is to eliminate the need for human involvement in dangerous situations while achieving optimal cleaning performance with reduced time and labor costs.

Overview of the Window Cleaning Robot

Our proposed solution is a robotic system designed specifically for window cleaning tasks on vertical surfaces. The robot is equipped with advanced adhesion mechanisms, powered by an impeller-based system that allows it to firmly attach to glass surfaces and move with precision. The design consists of three parts: a front chassis, rear chassis, and a central link, which connect and support the robot's movement across various obstacles.

The key innovation in this robot lies in its ability to overcome the limitations of existing solutions. It is not only designed to adhere to smooth, vertical surfaces but also to navigate around obstacles such as window frames and ledges. This is made possible through a power screw system that lifts each module, enabling the robot to adjust its height and position seamlessly.

Summary

Our window cleaning robot is designed to address the challenges of manually cleaning high-rise and large glass surfaces, with a strong emphasis on safety, efficiency, and consistency. This innovative solution is specifically aimed at improving the cleaning process for skyscrapers and other tall buildings, where manual cleaning poses significant risks and inefficiencies. The robot is engineered to autonomously navigate glass surfaces, utilizing advanced adhesion mechanisms and precise control systems to ensure a safe, effective, and reliable cleaning process. By eliminating the need for human workers at heights, this solution minimizes risk while maintaining high cleaning standards.

Key Features:

- **Adhesion Mechanism**: The robot utilizes an impeller-based suction system that provides strong adhesion to vertical glass surfaces, ensuring reliable operation at various heights.
- **Modular Design**: The robot is composed of a single unit. This modular design enhances flexibility in movement, contributing to the robot's stability during cleaning operations.
- **Obstacle Navigation**: Equipped with a power screw system, the robot can lift and lower its modules to navigate over obstacles such as window frames and ledges, ensuring thorough cleaning of difficult areas.
- **Materials**: The robot's body is made from lightweight acrylic, which offers durability and resistance to environmental factors, while the link component is constructed from aluminum for additional strength and support.

Impact:

The window cleaning robot represents a significant advancement in building maintenance technology. It not only addresses the limitations of traditional cleaning methods but also sets a new standard for safety and efficiency in industry. This innovative approach has the potential to revolutionize how we maintain tall structures, offering a safer, more reliable, and cost-effective solution.

Organization of the thesis

The thesis compromises five chapters as follows:

Chapter 1 shows the different designs of previous window cleaning robots based on different locomotion and adhesion concepts.

Chapter 2 shows the design of the robot using "SOLIDWORKS" and performing the stress analysis of each part

Chapter 3 studies the model of the robot; kinematic, dynamic and adhesion force calculations are done.

Chapter 4 ROBOTICS OPERATING SYSTEM 2 (ROS 2)

Chapter 5 presents the experimental work, the manufacture and tests of the different parts

Contents

TABLE OF FIGURE

Chapter 1
INTRODUCTION AND LITERATURE REVIEW

Chapter 1
INTRODUCTION AND LITERATURE REVIEW

1.1 Introduction

Cleaning windows on high-rise buildings and large glass structures presents significant challenges in terms of safety, cost, and efficiency. Traditional methods of manual window cleaning involve labor-intensive processes that expose workers to hazardous conditions, particularly at great heights and in inclement weather. Moreover, manual cleaning is time-consuming, inconsistent, and often expensive due to the specialized equipment and skilled labor required.

In response to these challenges, automation in window cleaning offers a transformative solution. Our project aims to develop an autonomous window cleaning robot capable of operating on vertical glass surfaces with minimal human intervention. The robot is designed to provide a safer, more efficient, and cost-effective alternative to traditional methods while delivering consistent cleaning results across large areas.

The key innovation in this robot lies in its ability to adhere to vertical surfaces using an advanced impeller suction system. This allows the robot to move freely and clean windows in tall buildings without the need for scaffolding or human labor. Additionally, the robot's modular design, consisting of a front chassis, rear chassis, and a central link, enables it to navigate over obstacles such as window frames and ledges, providing comprehensive cleaning coverage.

This project explores various aspects of the robot’s design and functionality, including its locomotion, perception, navigation, and safety mechanisms. By leveraging cutting-edge technology and intelligent engineering, our robot is poised to revolutionize the window cleaning industry by enhancing safety, reducing costs, and improving efficiency in building maintenance.

1.2 Technical Design and Features

The robot's design incorporates several important features that contribute to its functionality:

- **Adhesion Mechanism**: The impeller-based suction system ensures strong adhesion to vertical glass surfaces, allowing the robot to operate reliably at any height.
- **Modular Structure**: The robot is a single unit, which enhances stability and provides the necessary balance during cleaning operations.
- **Obstacle Navigation**: The robot is equipped with a belt-driven front wheel mechanism that allows it to move forward and backward precisely, helping it navigate over obstacles like window frames and ledges, and adjust its position to clean hard-to-reach areas.
- **Materials**: The robot's body is made from lightweight acrylic, while some mounting parts are made of sheet metal to ensure strength and support.

1.3 Innovation and Advantages

This robot introduces several innovations that improve upon current window cleaning technologies:

- **Safety**: By automating the window cleaning process, we eliminate the need for human workers to be exposed to dangerous heights, greatly reducing the risk of accidents.
- **Efficiency**: The robot can clean large surface areas more quickly than manual methods, saving both time and labor costs for building maintenance teams.
- **Consistency and Precision**: Unlike manual cleaning, which can be inconsistent depending on the worker, the robot ensures uniform cleaning across all windows, providing high-quality results every time.

- **Scalability**: The design is scalable, meaning that it can be adapted for use on various building sizes and types. It can also be easily maintained and upgraded to meet specific client needs.

1.4 Applications and Potential Impact

The window cleaning robot has significant potential in commercial applications, particularly in industries where building maintenance is a regular concern. For skyscrapers, high-rise office buildings, and large glass-covered structures, this robot offers a practical and cost-effective solution. It can also be used in residential buildings with large windows or hard-to-reach surfaces.

Beyond window cleaning, the technology behind this robot has the potential to be adapted for other vertical surface tasks, such as inspection, painting, or repair work on high-rise buildings, further broadening its range of applications.

1.5 Locomotion Mechanisms

Locomotion is a critical component of your window cleaning robot, as it determines how the robot navigates vertical surfaces. Given the robot's task of cleaning windows on tall buildings, it needs a robust and reliable system to adhere to vertical glass surfaces and move efficiently.

For our robot, the primary locomotion mechanism is based on the use of **timing belts** for movement and an **impeller suction system** for adhesion. The combination of these two systems allows the robot to:

1. **Adhere securely to vertical surfaces** using the impeller's vacuum suction power.
2. **Move smoothly across the glass** using the timing belt system, ensuring precise control of speed and direction.
3. **Lift and lower each module** using a power screw system, allowing the robot to navigate obstacles like window frames or ledges.

4. **Steer and adjust positioning** to cover the entire surface area efficiently.

By using these mechanisms, the robot achieves efficient and consistent cleaning, regardless of the height of the structure.

1.5.1 CrawleR robots

The crawlers (sliding segments) climbing robots are relatively fast, but it is not suitable for rough environments. They based on using the suction cup or permanent magnet to grab to the surface and so, it is difficult to cross over cracks and obstacles

FIGURE 1 CRAWLER ROBOT

1.5.2 Tracked climbing robots

Tracked climbing robots have better ability to avoid obstacles and adhere to the surface. These robots move on the wall using the track and adhere to the wall with different techniques. Some robots depend on using suction cups in the track for moving on the wall and glass such as Cleanbot II and GEKKO III in figure 2 climbing the glass. Cleanbot II depends on the suction and the vacuum. The robot can turn in a limited range and climb over an obstacle that is less than 6mm high. It is 22kg and its payload is 25kg.The maximum speed is 10 m/min

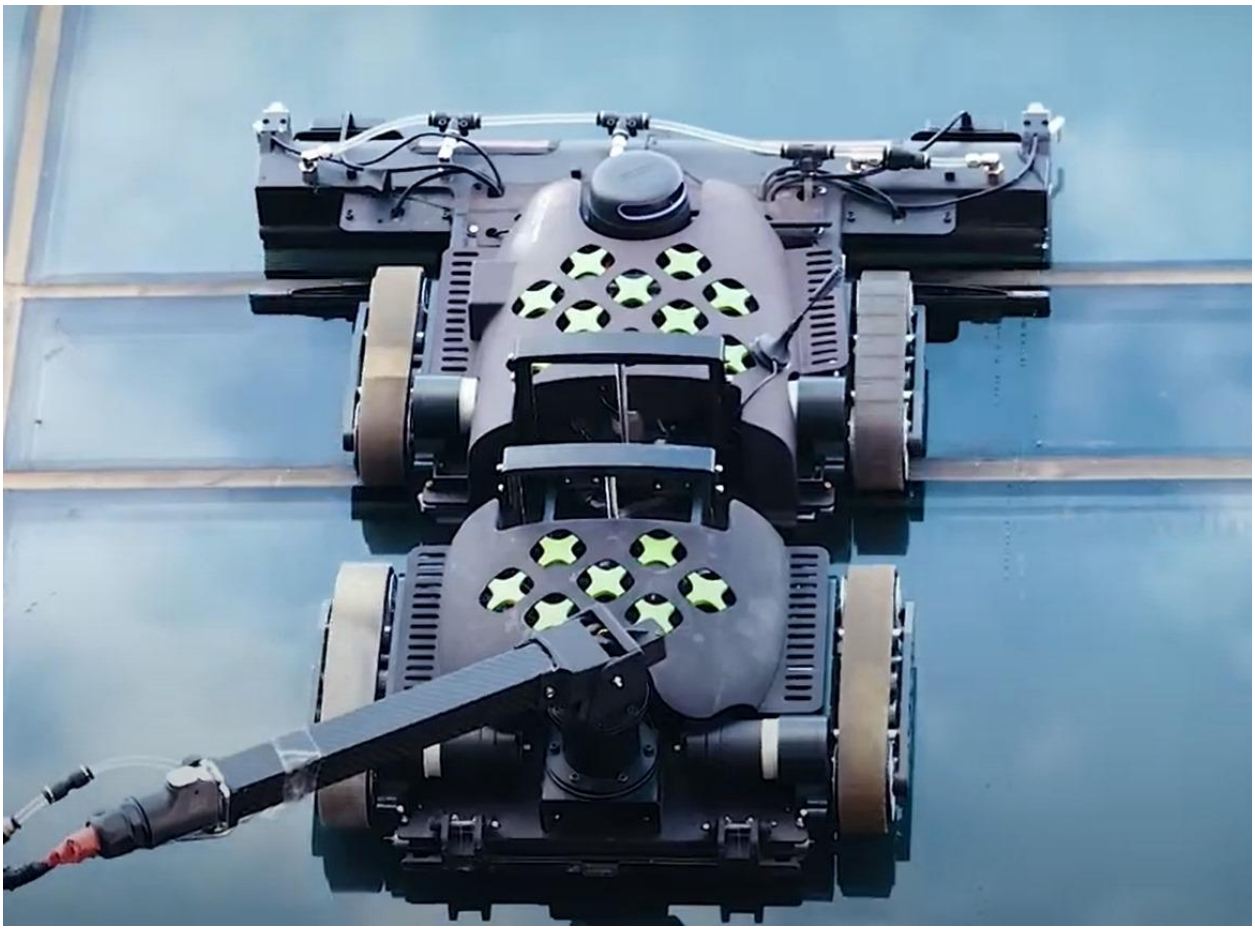
1.5.3 Gekko Facade Robot:

Gekko is an industrial cleaning robot designed to clean large building facades. It uses vacuum suction technology to adhere to vertical surfaces such as glass or metal panels. The robot moves using mechanical tracks and can be programmed for automated cleaning cycles, allowing it to efficiently clean large areas.

FIGURE 2 GEKKO FACADE ROBOT

1.5.4 SHENXI

To address the dangers of working at heights, the Intelligent Curtain Wall Cleaning Robot offers a better solution. The robot is equipped with functions such as automatic operation, gap crossing, edge turning, obstacle overcoming, exterior ceiling cleaning, and adaptation to various materials like curved surfaces, marble, and aluminum-plastic panels. It also features high-temperature steam cleaning, front multi-sensors, and a camera to deliver exceptional performance.



1.5.5 Cable-Driven climbibg robots

These robots use cable for climbing. A tether-supported climbing robot has been designed and realized for the purpose of glass-curtain wall cleaning. The robot does not have its own driving mechanism but can move on smooth glass surfaces by depending on gravity and the lifting force of the trolley crane on the roof, while adhering to the surfaces using dual vacuum suction cups. Obstacles such as horizontal window frames are detected by four groups of photo-electronic sensors and can be crossed while cleaning.

1.5.6 SIRIUS:

SIRIUS is a robot designed to clean the windows of high-rise towers. It uses suction cups to stick to windows and employs AI algorithms for cleaning path planning. The robot is fully autonomous, making it ideal for commercial buildings without the need for direct human operation.

FIGURE 3 SIRIUS

1.5.7 Skyline Robotics' Ozmo:

Ozmo is an advanced industrial robot designed for cleaning high-rise tower windows. It features robotic arms equipped with cleaning tools and strong suction to stick to surfaces. The robot moves across large surfaces efficiently and is ideal for companies managing skyscrapers.

FIGURE 4 SKYLINE ROBOTICS' OZMO

1.5.8 Adhesion force

Wall climbing robot should have enough adhesion force for supporting the robot. The adhesion force may be the suction, magnetic, gripping to the surfaces and the biological inspired findings based on nano- technology.

- **Suction adhesion force**

Wall climbing robots depend on the suction adhesion force take many configurations and designs. The suction force can be active or passive adhesion force. The active adhesion force is based on using a vacuum pump, high-speed rotating impeller. Sky cleaner I, II are wall climbing robot based on a suction cup and vacuum pump. These robots are designed for cleaning glass surfaces.

FIGURE 5 CITY CLIMBER ROBOT ||

1.5.9 Legged robots

Legged robots have many designs where they are very good in cross over obstacles, but their speed is low and their control is complex.

Usually using suction cups or magnetic device on the feet, grasping grippers, the last type usually has a lot of degrees of freedom, they can move over rough surfaces and cracks, and are capable of good obstacle avoidance. However, they require complicated control systems because of the use of harmonic gait control and have the disadvantage of low speeds of motion due to discontinuous movement.

Legged robots have many configurations where the legs can be two up to eight legs. Increasing the legs lead to increasing the pay load, size and weight.

1.5.10 Climbot

Climbot is a versatile robot designed to navigate vertical and sloped surfaces. It uses powerful suction for movement and is flexible enough to handle different types of surfaces. It is used in environments requiring automated cleaning of vertical surfaces like windows in high-rise commercial buildings.

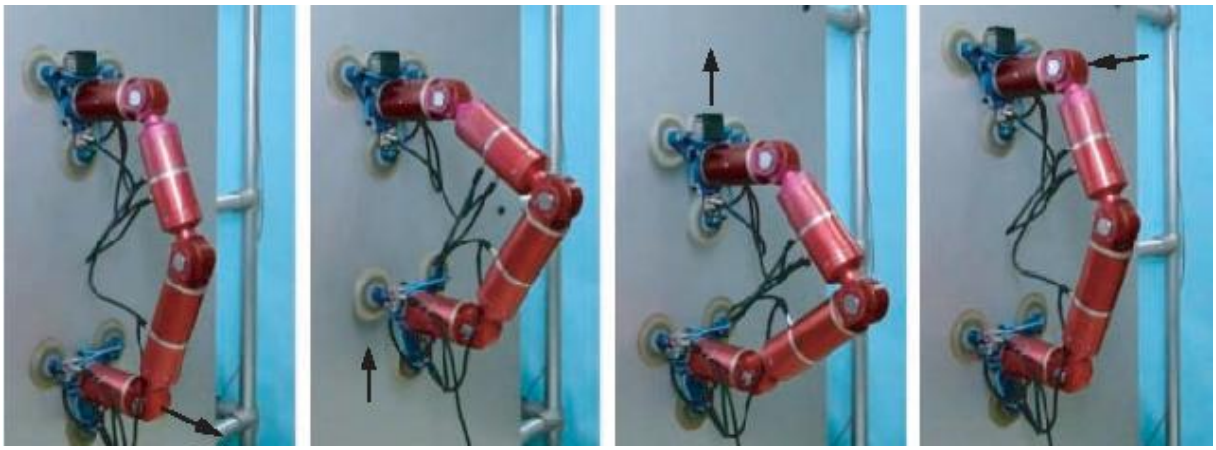


FIGURE 6 CLIMBOT

Chapter 2
Mechanical Design

Chapter 2
Mechanical Design

2.1 Introduction

In this section, we present the mechanical design of our window cleaning robot, which is intended for automated and efficient cleaning of glass surfaces. We describe the main components and features of the robot and explain the design choices and trade-offs made during the development process. We also show the results of the simulation and analysis of the robot's performance, focusing on its capabilities in locomotion and movement, as well as its perception and navigation functionalities. Additionally, we discuss the overall performance and effectiveness of the robot in successfully completing its tasks.

2.2 Appearance and structure:

Our window cleaning robot features a modern and practical design, consisting of a rectangular body measuring 45 cm in length, 45 cm in width, and 17 cm in height. The robot contains two DC brushless motors in the rear wheels, responsible for moving the robot forward and backward. The front wheel is equipped with a shaft, which moves using a belt mechanism. The robot is also equipped with a LIDAR for mapping. The distance between the front and rear wheels is 35 cm.

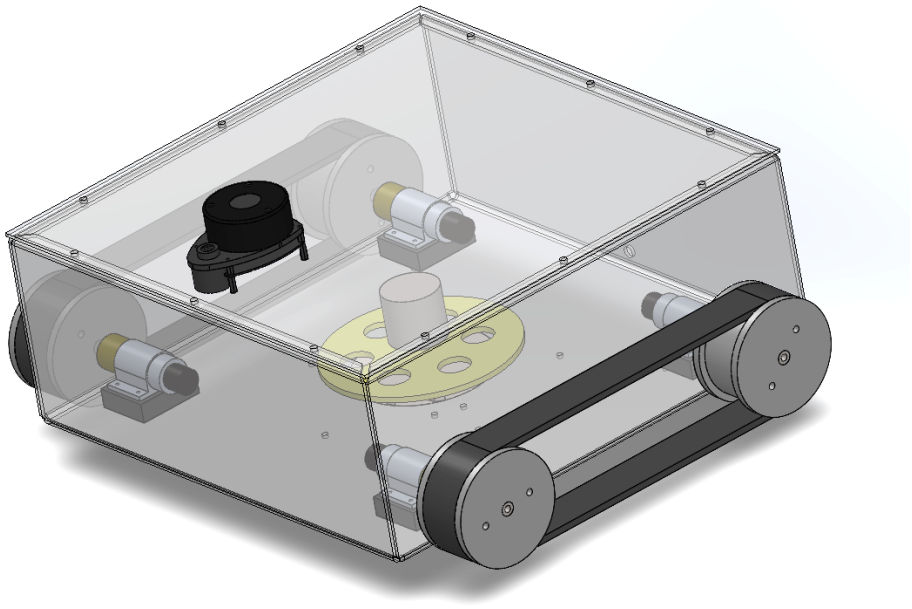


FIGURE 7

2.3 Design and Components:

with its environment. The main sensors include an ultrasonic sensor, IMU, and wheel odometer. The primary actuators are two DC brushless motors for the wheels, a linear actuator, and an impeller suction system. The robot consists of a single unit with no link connecting separate halves, enabling seamless operation across surfaces.

The ultrasonic sensor, mounted at the front, is used for close-range obstacle avoidance. The IMU and wheel odometer provide essential data for movement and orientation. The motors control the movement and steering of the robot, while the impeller suction system ensures strong adhesion to glass surfaces during operation, maintaining stability while cleaning.

2.3.1 suction package

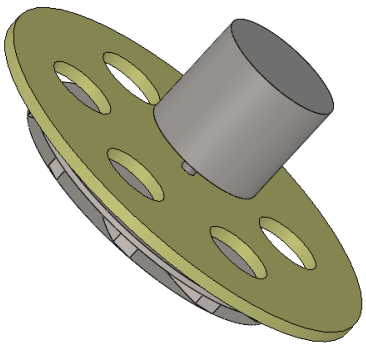


FIGURE 8 AC MOTOR

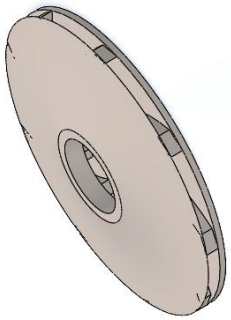


FIGURE 9 CLOSED IMPELLER

2.3.2 timing belt

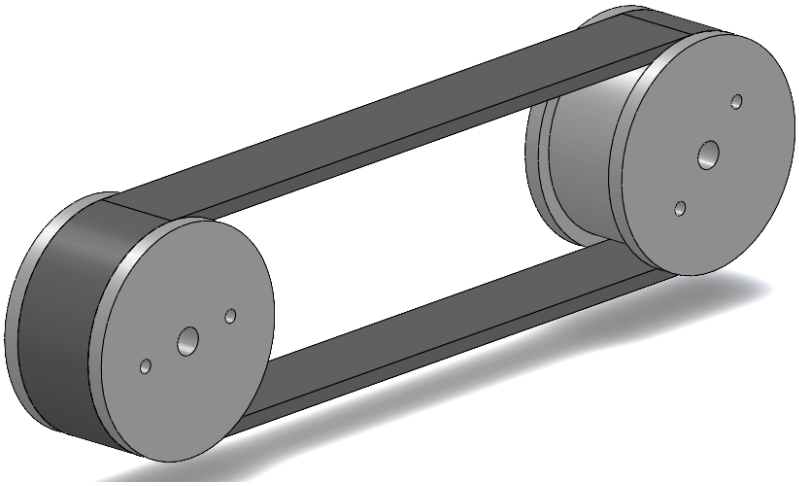


FIGURE 10 TIMING BELT

2.3.4 **cleaning part**

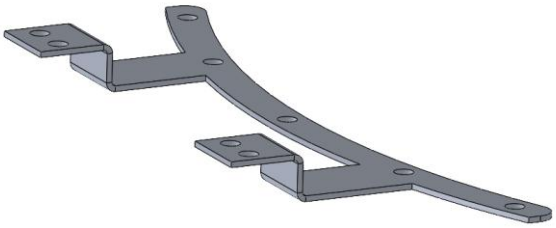


FIGURE 12 HOLDER

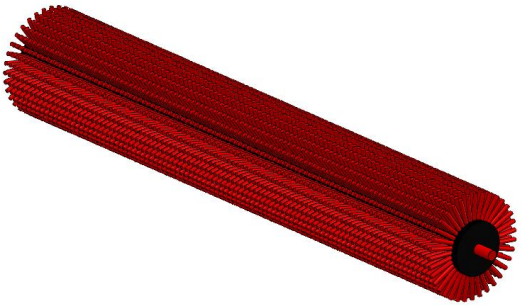


FIGURE 11 BRUSH

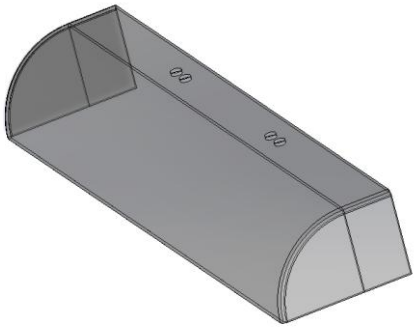


FIGURE 14 COVER

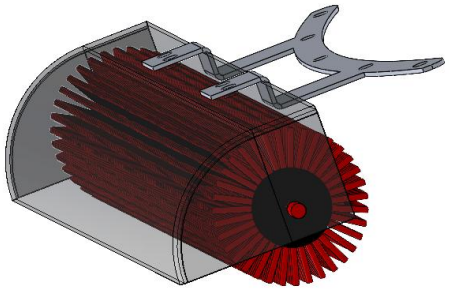


FIGURE 13

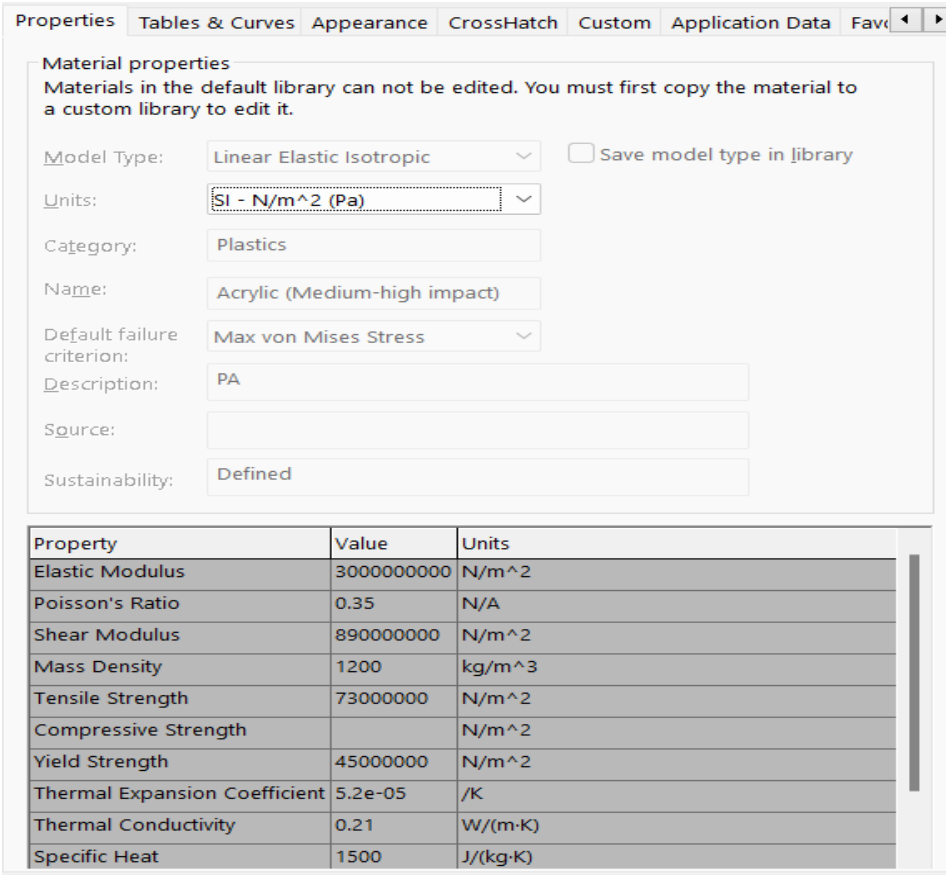
2.4 Stress Analysis using SolidWorks

In this part, stress analysis is a method used in engineering to determine the stresses and strains in materials and structures subjected to forces. In the context of our window cleaning robot, stress analysis is crucial for several reasons:

1. **Safety**: It ensures that the robot can withstand the forces it will encounter during operation, especially when adhering to and moving across vertical surfaces.
2. **Durability**: It helps in predicting the robot's lifespan by analyzing how the materials will behave under different stress conditions.
3. **Optimization**: It aids in material selection and design optimization, ensuring the robot is neither over-engineered (leading to unnecessary costs) nor under-engineered (which could cause premature failure).

2.4.1 Setting up the Simulation (Chassis):

Acrylic is a suitable choice for the robot's body due to its lightweight and rigidity. Although it is lighter than many other materials, acrylic offers good resistance to the stresses the robot may encounter during operation. Its transparency is an added advantage in certain applications, allowing visibility of internal components or progress tracking. Acrylic is also resistant to corrosion and environmental factors, enhancing the robot's durability and maintaining its appearance over time. Additionally, it is easy to mold and process, making it ideal for custom and flexible designs.



Property	Value	Units
Elastic Modulus	3000000000	N/m^2
Poisson's Ratio	0.35	N/A
Shear Modulus	890000000	N/m^2
Mass Density	1200	kg/m^3
Tensile Strength	73000000	N/m^2
Compressive Strength		N/m^2
Yield Strength	45000000	N/m^2
Thermal Expansion Coefficient	5.2e-05	/K
Thermal Conductivity	0.21	W/(m·K)
Specific Heat	1500	J/(kg·K)

FIGURE 15 TABLE OF ACRYLIC MATERIAL

Base Plate Stress Analysis

In the design of our window cleaning robot, the base plate serves as the fundamental structural element to which all components are attached. This includes the passive and drive wheels, the lifting arm, and the second module when crossing obstacles. Additionally, the adhesion force generated by the impeller system is transferred through the base plate.

The base plate is made of acrylic, chosen for its lightweight properties and sufficient strength for the application. To evaluate its structural integrity, a force of 90 N is applied to the upper surface, simulating the load during operation. Furthermore, approximately 50 N of force is applied to the cross-sectional area surrounding each bolt hole, representing the stresses experienced during the attachment of components.

The results of the stress analysis show that the acrylic base plate can handle the applied forces effectively. The maximum displacement observed is less than 2.69 mm, indicating minimal deformation under load. Moreover, the factor of safety is greater than 3.26, ensuring that the base plate can withstand the operational stresses without risk of failure. These results confirm that the base plate is structurally sound and capable of supporting the robot's components and operational forces.

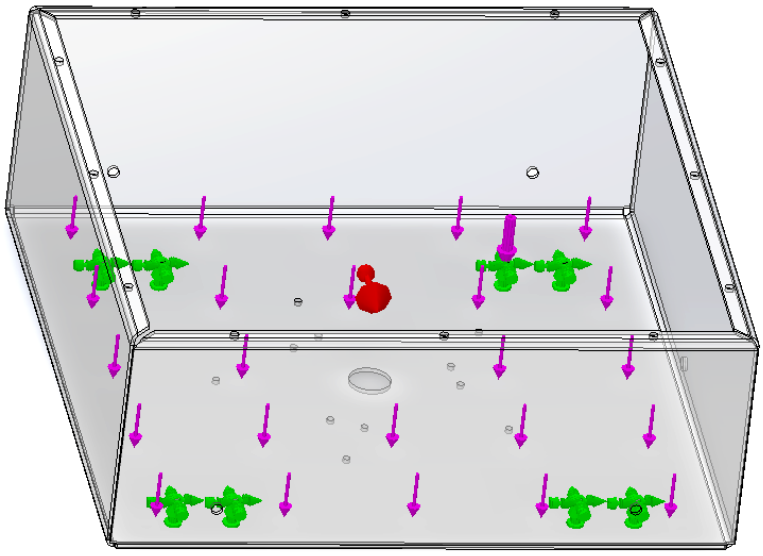


FIGURE 16 FORCE AND GRAVITY

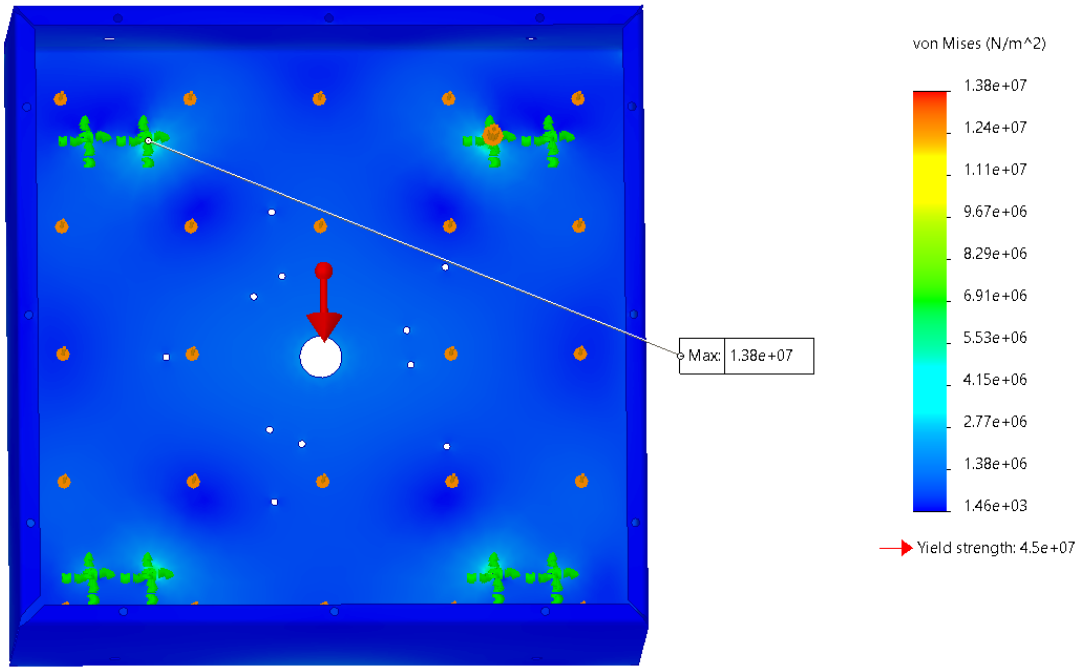


FIGURE 18 STRESSES ON BODY

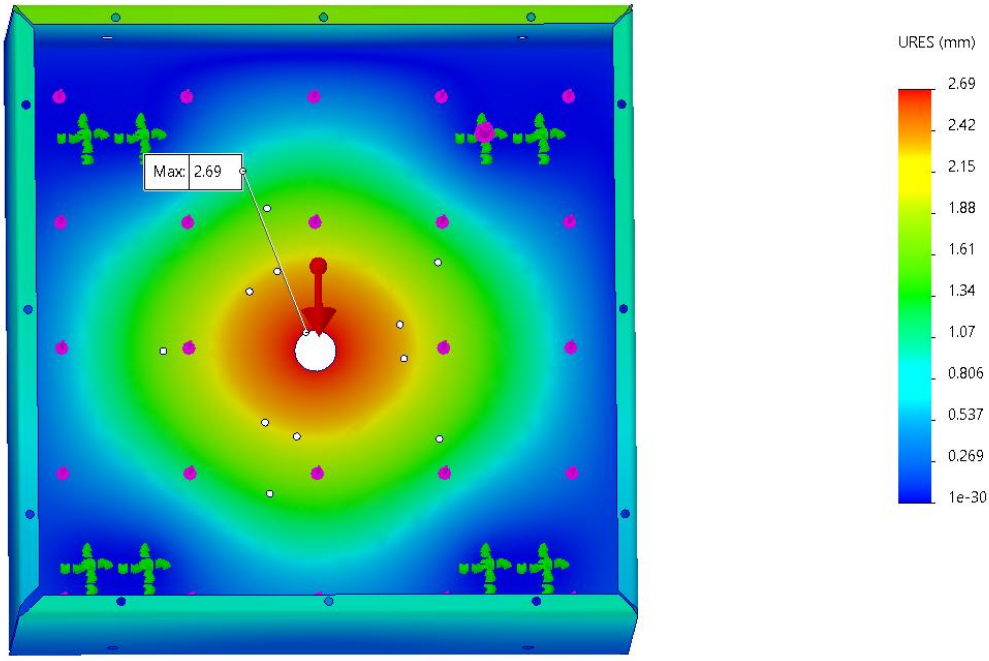


FIGURE 17 DISPLACEMENT IN BODY

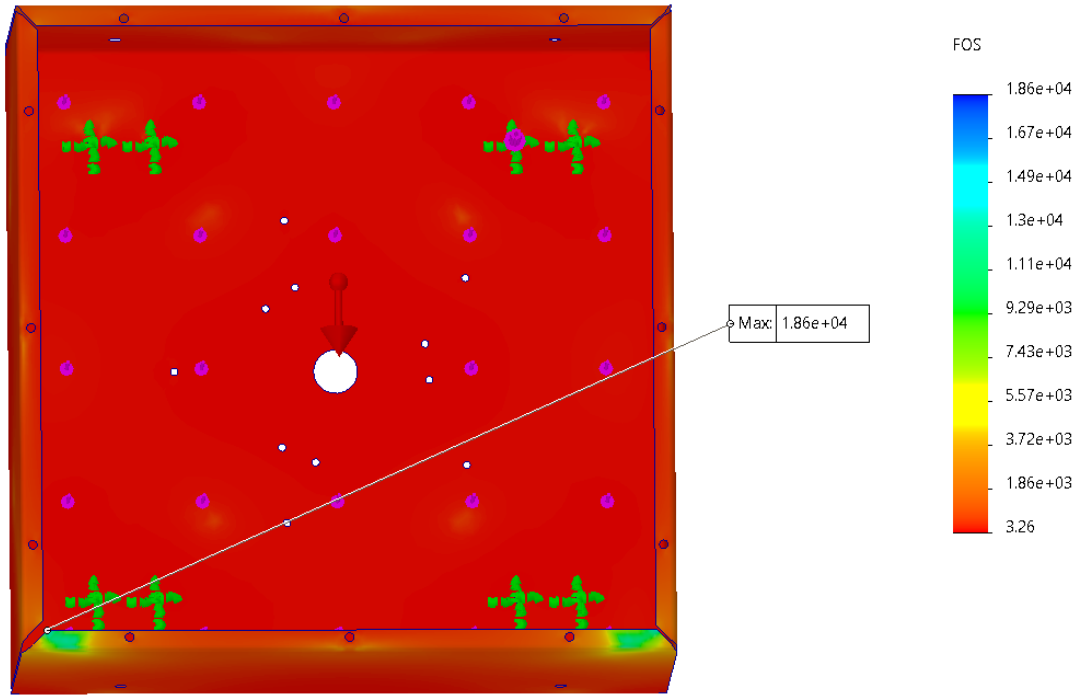


FIGURE 19 F.O.S FOR BODY

2.4.2 motor holder

The motor mount is made of sheet metal, which is an excellent choice due to its strength and durability. Sheet metal offers high resistance to mechanical stresses, ensuring the mount can securely support the motor during operation without deformation. Its lightweight nature also contributes to the overall efficiency of the robot, minimizing unnecessary weight. Additionally, sheet metal is easy to fabricate, allowing for precise customization to fit the motor's dimensions and requirements. To ensure the mount's reliability, a stress analysis will be performed to evaluate its performance under load conditions, ensuring safety and optimal design.

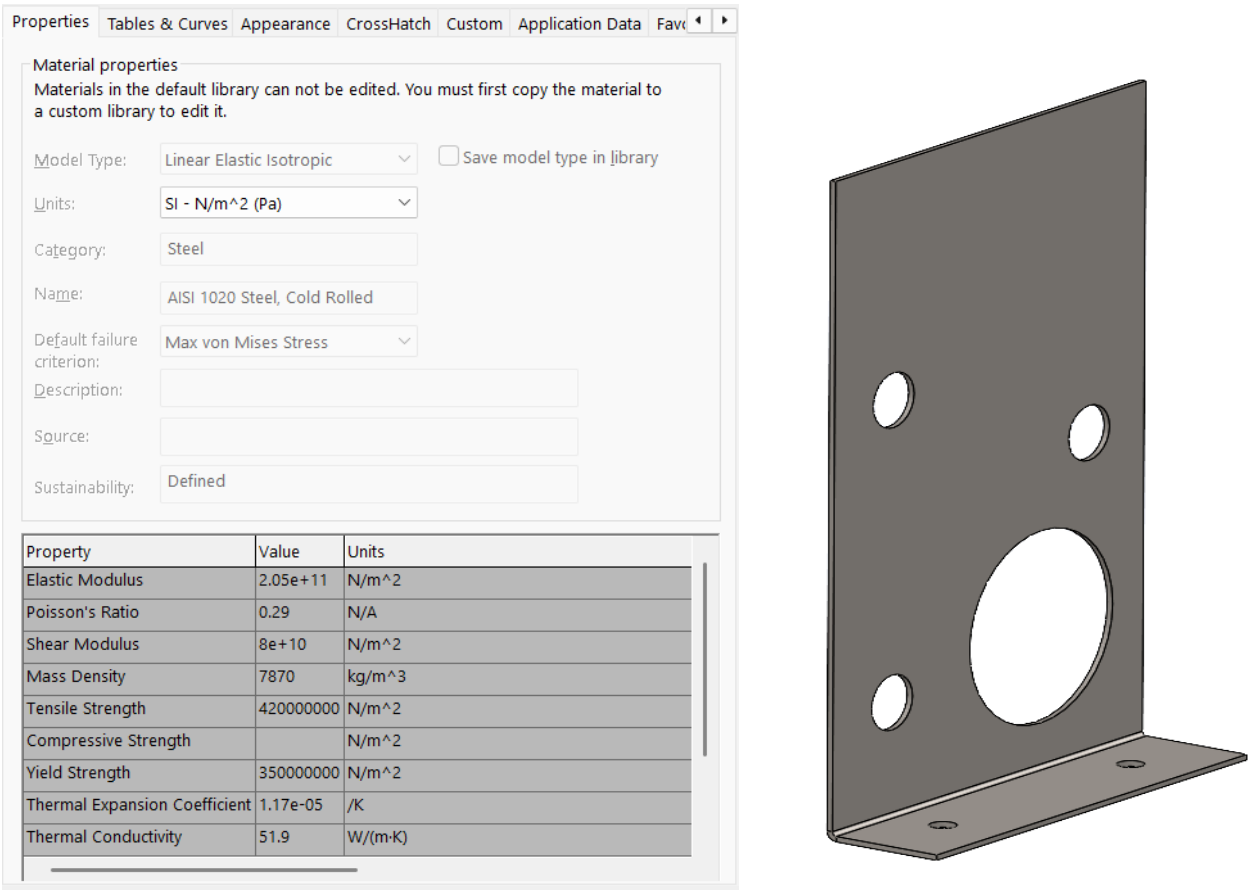


FIGURE 20 MATERIAL OF HOLDER

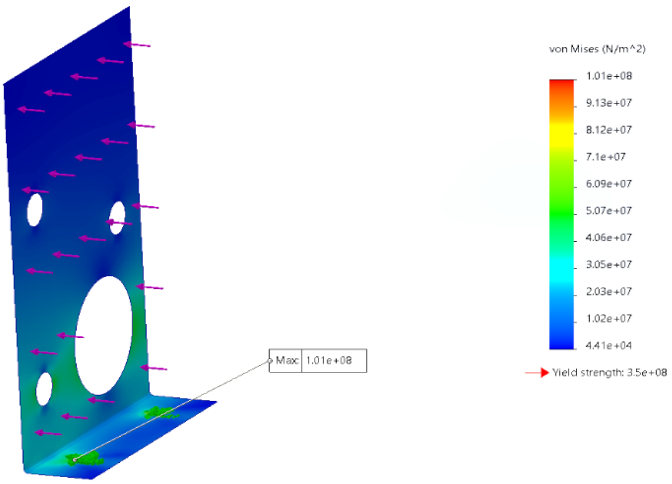


FIGURE 21 STRESS IN MOTOR HOLDER

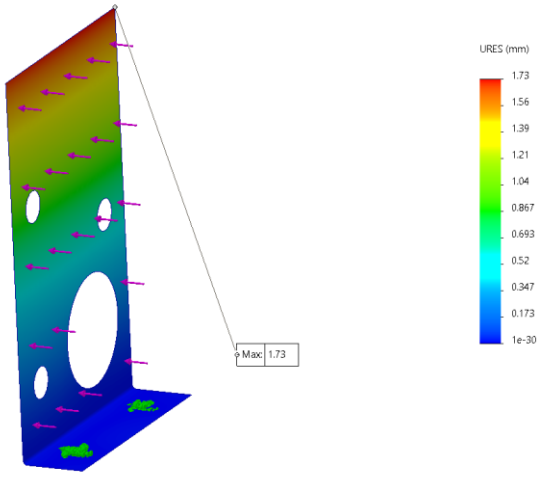


FIGURE 22 DISPLACEMENT IN MOTOR HOLDER

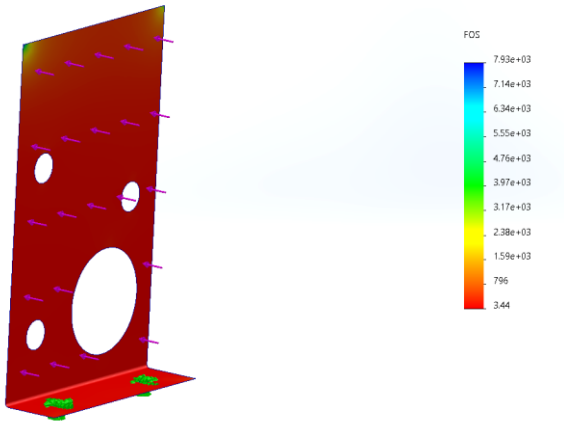


FIGURE 23 F.O.S FOR MOTOR HOLDER

2.5 sealing

Sealing is an essential process in various engineering applications, where it involves creating a barrier to prevent the escape of fluids (gases or liquids) or contamination from the outside environment. In robotics, sealing systems are critical for maintaining efficient performance, especially when adhesion or suction is required to interact with surfaces such as walls or windows. Effective sealing can enhance the overall function of a system by ensuring that it adheres securely to surfaces while minimizing energy loss, leakage, and wear.

In the context of your window cleaning robot, the sealing system is responsible for ensuring that the robot remains firmly attached to the glass or wall surface by creating a vacuum-like effect. This allows the robot to move efficiently and perform its cleaning tasks without falling off or losing suction power. The quality of the seal directly affects the robot's performance and stability.

Importance of Sealing in Robotics

In robotics, sealing systems are crucial for maintaining the performance of mobile robots, particularly those designed for specialized tasks like cleaning. In your window cleaning robot, a proper sealing system ensures the robot can generate sufficient suction to stay attached to the glass while cleaning. Poor sealing could lead to loss of suction, affecting the robot's stability and performance. Therefore, selecting the right sealing material and design is essential to achieving reliable operation, minimizing energy consumption, and ensuring long-term durability.

Types of Sealing

1. **Mechanical Seals**:
   Mechanical seals are designed to prevent leakage between two parts that rotate relative to each other, commonly used in pumps, motors, and rotating machinery. These seals typically include a combination of elastomers and metallic components to ensure tight sealing under pressure.

2. **Gasket Seals**:
   Gasket seals are used to prevent fluid leakage between two static surfaces. Gaskets are commonly made from materials such as rubber, silicone, or metal. They are placed between flanges, covers, or joints to ensure tight sealing, often used in engines, pipes, and industrial equipment.

3. **O-Rings**:
   O-rings are circular seals that fit into a groove and create a seal between two surfaces. These are made from elastomers, which allow them to deform and fill any gaps or irregularities in the mating surfaces. O-rings are widely used in hydraulic, pneumatic, and low-pressure applications.

4. **Vacuum Seals**:
   Vacuum seals are specifically designed for systems that rely on creating a vacuum or suction. These seals are commonly used in vacuum pumps, packaging systems, and robots that require adhesion to walls or windows, such as in your cleaning robot project. They are designed to handle both high and low-pressure environments and prevent air from leaking into the system.

FIGURE 24 EXAMPEL FOR SEALING

2.6 Closed Impeller and Flow Simulation with CFD in SolidWorks

Closed Impeller

A closed impeller is a specialized type of centrifugal impeller characterized by its enclosed blade structure. Unlike open impellers with exposed blades, a closed impeller features covers on both sides of its blades, creating a fully enclosed flow path. This design enables more efficient handling of fluids and enhances performance in applications demanding high pressure and flow rates.

The primary function of a closed impeller is to convert the rotational energy from a motor into fluid energy, resulting in a high-velocity stream of liquid or gas. The enclosed design minimizes fluid energy loss and promotes stable flow. Closed impellers are particularly effective in applications where precise control and high efficiency are required, such as in vacuum systems, pumps, and ventilation systems.

Flow Simulation with CFD in SolidWorks

Computational Fluid Dynamics (CFD) in SolidWorks is an advanced tool for simulating and analyzing fluid behavior within a given system. CFD enables engineers to evaluate how fluids interact with their designs, providing valuable insights into performance, efficiency, and potential issues.

When applying CFD for a closed impeller in SolidWorks, the process involves several key steps:

1. **Model Setup**: Import or create the 3D model of the closed impeller within SolidWorks, ensuring that all geometric details, including blade profiles and the enclosed housing, are accurately represented.
2. **Define Fluid Domain**: Establish the fluid domain around the impeller by setting up the regions where the fluid will flow. This includes defining the boundaries of the simulation, such as inlet and outlet conditions.
3. **Set Fluid Properties**: Input the properties of the fluid, such as density, viscosity, and temperature. CFD allows for the specification of both liquid and gas properties, and their behavior under different conditions.

4. **Generate Mesh**: Create a computational mesh to discretize the fluid domain. A finer mesh around the impeller blades is essential to capture detailed flow patterns and ensure accurate results.
5. **Apply Boundary Conditions**: Define the operational conditions for the simulation, including inlet velocity, pressure, and rotational speed of the impeller. Also, set any external forces or constraints affecting the system.
6. **Run the Simulation**: Execute the CFD simulation to analyze fluid dynamics. SolidWorks performs calculations to determine how the fluid interacts with the closed impeller, including velocity distribution, pressure changes, and flow patterns.
7. **Analyze Results**: Review the simulation results using SolidWorks' visualization tools. This includes examining velocity vectors, pressure contours, and flow trajectories to evaluate the impeller's performance. Identify any issues such as flow recirculation or high turbulence areas.
8. **Optimize Design**: Based on the CFD results, make design adjustments to improve performance. This might involve modifying blade shapes, adjusting inlet and outlet conditions, or refining overall geometry.

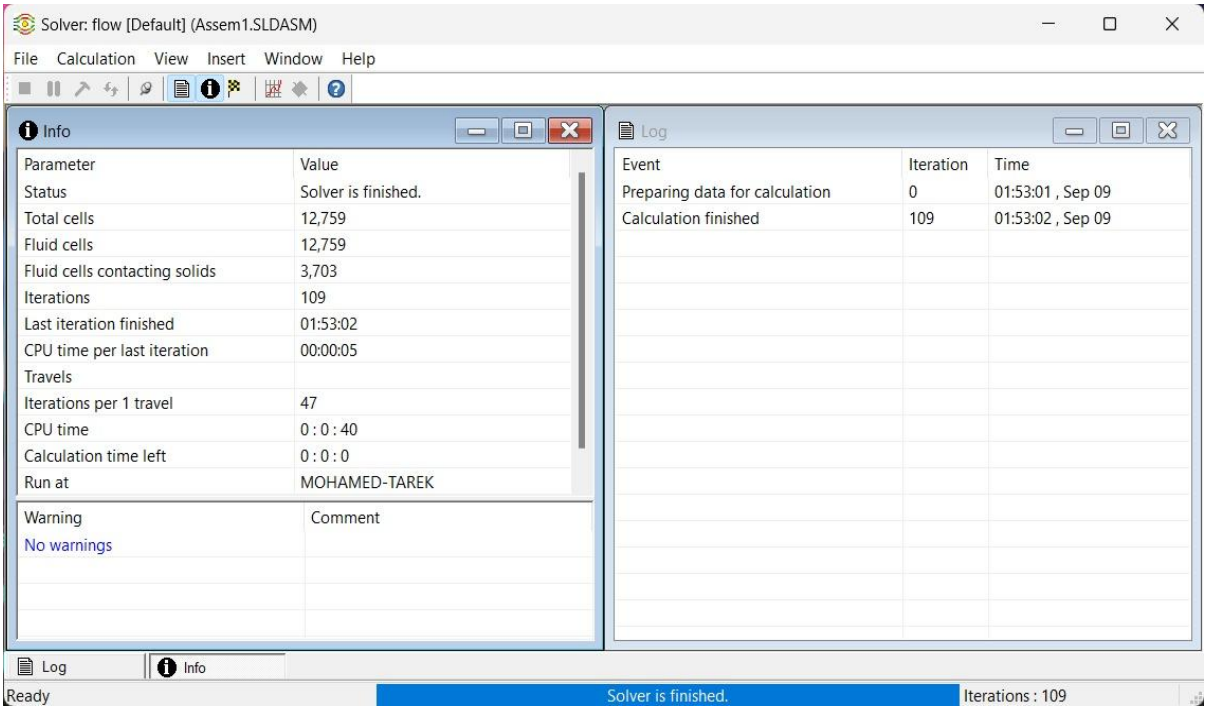


FIGURE 25

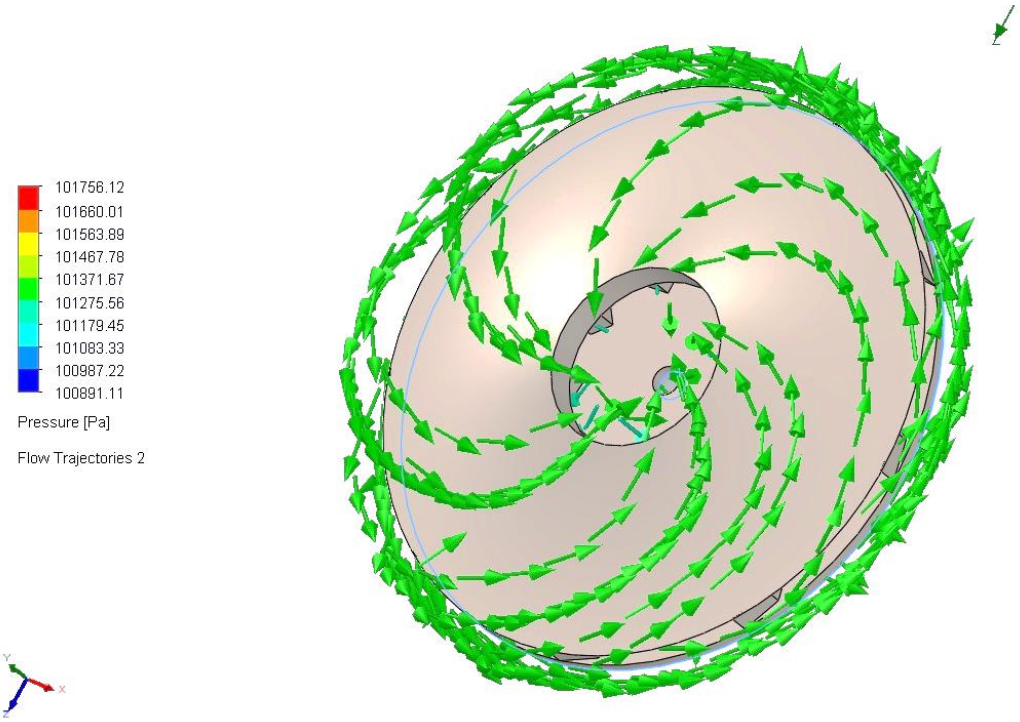


FIGURE 26

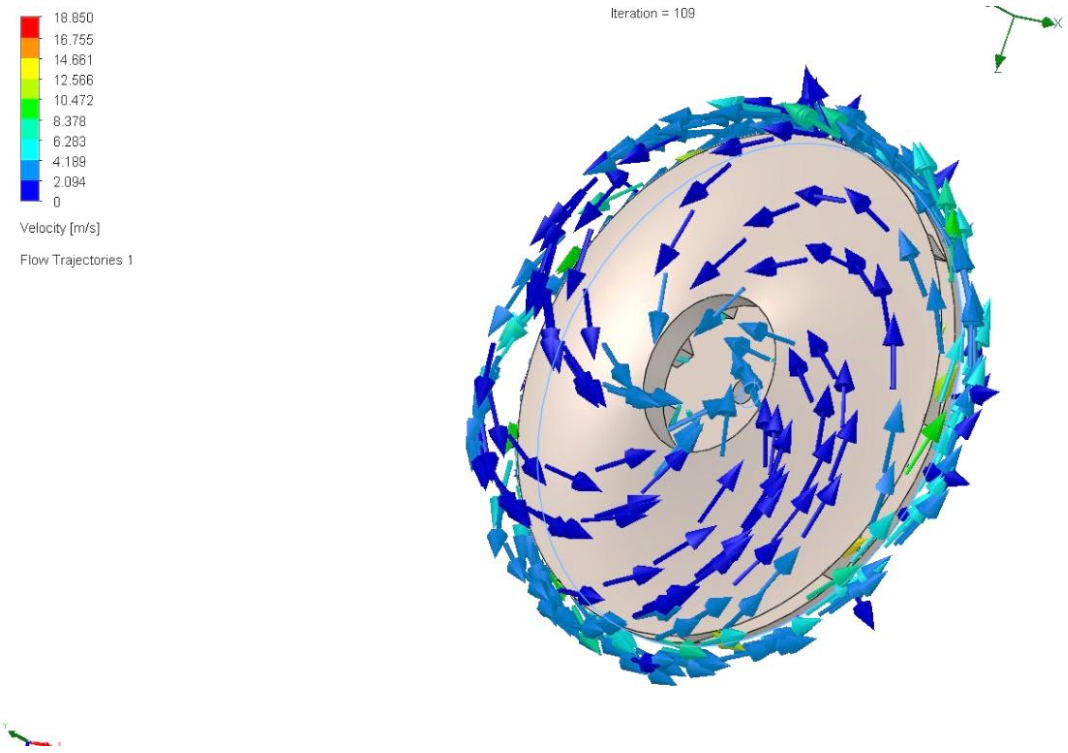


FIGURE 27

2.7 Designing the Sealing System for Adhesion

To achieve reliable adhesion to the glass surface, we need to analyze the sealing system's performance based on the motor's suction power, the sealing dimensions, and the pressure difference it creates.

Given Data:

1. Diameter of the sealing (∂): 16 cm = 0.16 m
2. Height of the sealing (h): 1.2 cm = 0.012 m
3. Pressure difference caused by the suction motor (ΔP): 90 kPa = 90,000 Pa

Key Parameters to Analyze:

1. **Sealing Area (Δ):** The area affected by the pressure difference can be calculated as:

$$A = \pi \left(\frac{d}{2}\right)^2$$

$$A = \pi \left(\frac{16}{2}\right)^2 = \pi(0.08)^2 \approx 0.0201\ m^2$$

2. **Adhesion Force (F):** The force holding the robot against the wall is the product of the pressure difference and the sealing area:

$$F = \Delta P \times A$$

$$F = 90000 \times 0.0201 \approx 1809\ N$$

This force is sufficient to support the weight of the robot (8 kg, approximately 78.48 N) and provide extra holding force for stability.

3. **Sealing Design Considerations:**
    - The sealing material should be soft and flexible (e.g., rubber or silicone) to ensure proper contact with uneven glass surfaces.
    - The height (h = 1.2 cm) should balance between flexibility and stability, ensuring it doesn't deform excessively under suction.
    - The sealing should create an airtight environment to maintain the pressure difference effectively.

4. **Air Leakage Analysis:** To minimize air leakage, the sealing should:
    - Be uniform and continuous along its perimeter.
    - Have a high-quality finish to prevent micro-leaks.
    - Be maintained properly to avoid wear and tear over time.

5. **Sealing Perimeter (P):** The perimeter of the sealing, which needs to be airtight, is given by:

$$P = \pi \times d$$

$$P = \pi \times 0.16 \approx 0.502m\ (50.2cm)$$

Air Leakage Rate Calculation

Key Parameters:

1. Pressure difference (ΔP): 90,000 Pa
2. Sealing Perimeter (P): $P = \pi \times 0.16 \approx 0.502m\ (50.2cm)$
3. Sealing height (h): 1.2 cm = 0.012 m
4. Air viscosity (μ): Typically, for air at room temperature, $\mu \approx 1.18 \times 10^{-5}\ Pa.s$
5. Density of air (ρ): 1.2 kg/m^3 (at standard conditions).

Flow Through the Leakage Path: Assume the leakage occurs between the sealing and the surface due to an imperfect seal. The flow is typically modeled as laminar flow through a rectangular gap.

Gap Geometry:

The leakage path is characterized by the gap height (h) and the sealing perimeter (P).

Flow Area (A_{leak}): The cross-sectional area for leakage is:

$$A_{leak} = P.h$$

$$A_{leak} = 0.502 * 0.012 = 0.006024\ \text{m}^2$$

Leakage Rate (Q) : Using equation for laminar flow through rectangular slit**:**

$$Q = \frac{\Delta P.\ A^2}{12\mu P}$$

$\Delta P = 90000\ Pa$

A_{leak}=$0.006024\ \text{m}^2$

$\mu = 1.81 * 10^{-5}$

$P = 0.502\ m$

$$Q = \frac{90000\ Pa.(0.006024\ )^2}{12 * 1.81 * 10^{-5} * 0.502\ m} = 29.96\ m^3/s$$

2. Total Normal Force (F_{normal})

The normal force combines the robot's weight and the suction force:

$$F_{normal} = W \cdot g + F_{suction}$$

$$F_{normal} = (8 \times 9,81) + 1,809 \approx 1887.48N$$

3. Friction Force (F_f)

The friction force is calculated using the normal force and the coefficient of friction:

$$F_f = \mu \cdot F_{normal}$$

$$F_f = 0.2 \times 1887.48 \approx 377.5\ N$$

4. Total Force Required to Move the Robot(F_{total})

The total force includes the friction force, and the force needed to accelerate the robot:

$$F_{total} = F_f + F_a$$

$$F_{total} = 377.5 + 4 = 381.5N$$

5. Torque Required for Each Motor

The torque is calculated based on the total force and the wheel radius:

$$\tau = \frac{F_{total} \cdot r}{N}$$

Where:

- R =0.05 (wheel radius).
- N =2 (two motors driving the rear wheels).

$$\tau = \frac{381.5 \times 0.05}{2} \approx 9.54N.m$$

The torque required for each motor to move the robot, considering the suction force, is:

$$\tau \approx 9.54N.m$$

Chapter 3
MOBILE ROBOT MODELING

Chapter 3
MOBILE ROBOT MODELING

Design, development, modification and control of a mechatronic system require an understanding and a suitable representation of a system; specifically, a “model” of the system is required. Any model is an idealization of the actual system. A mechatronic or robotic system may consist of several different types of components, and it is termed as a mixed system. One should use analogous procedures for modeling such components. In this manner the component models can be conveniently integrated to obtain the overall model. Modeling of a differential drive mobile robot platform consists of kinematic and dynamic modeling in addition to the modeling of the system actuators. Kinematic modeling deals with the geometric relationships that govern the system and studies the mathematics of motion without considering the affecting forces. Dynamic modeling on the other hand is the study of the motion in which forces and energies are modeled and included. Actuator modeling is needed to find the relationship between the control signal and the mechanical system’s input. Each part of this system’s modeling will be explained separately throughout this chapter. After getting an accurate model, we can simulate the system using an appropriate computer package. The simulation process will be explained thoroughly in one section of this chapter. The first step for mechanical modeling is to define appropriate coordinate systems for the platform which will be described in the next section.

3.1 COORDINATE SYSTEMS

The main function of the coordinate systems is to represent the position of the robot. The following two coordinate systems are used for mobile robot modeling and control purposes:

- Inertial Frame: $\{X_I, Y_I\}$ This is the fixed coordinate system in the plane of the robot.
- Robot Frame: $\{X_R, Y_R\}$ This is the coordinate system attached to the robot.

The above two coordinate systems are shown in Figure 48:

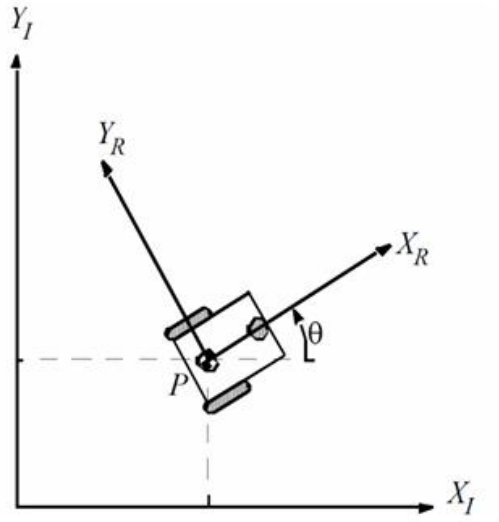


FIGURE 28

Introducing these coordinate systems is helpful in the kinematic modeling of the robot which will be explained in the next section. The important issue that needs to be explained is the mapping between these two frames. The robot position in the inertial and robot frame can be defined as follows:

$$q_I = (X_I . Y_I . \theta_I)^T$$

$$q_R = (X_R . Y_R . \theta_R)^T$$

The mapping between these two frames is through the standard orthogonal rotation transformation:

$$\dot{q}_R = R(\theta)\dot{q}_I$$

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Using the above equations, we have the relation between the robot velocities in the local frame and the inertial frame which is very important in the robot kinematics.

3.2 KINEMATIC MODELING OF THE MOBILE ROBOT

The goal of the robot kinematic modeling is to find the robot speed in the inertial frame as a function of the wheel's speeds and the geometric parameters of the robot (configuration coordinates). In other words, we want to establish the robot speed $\dot{q} = [\dot{x} \quad \dot{y} \quad \dot{\theta}]^T$ as a function of the wheel speeds $\dot{\emptyset}_R$ and $\dot{\emptyset}_L$ and the robot geometric parameters or we want to find the relationship between control parameters ($\dot{\emptyset}_R$ and $\dot{\emptyset}_L$) and the behavior of the system in the state space. The robot kinematics generally has two main analyses, one Forward kinematics and one Inverse kinematics:

- **FORWARD KINEMATICS:**

$$\dot{q} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = fn(\dot{\emptyset}_R, \dot{\emptyset}_L, L, R_a, \theta)$$

- **INVERSE KINEMATICS:**

$$\begin{pmatrix} \dot{\emptyset}_R \\ \dot{\emptyset}_L \end{pmatrix} = fn(\dot{x}, \dot{y}, \dot{\theta})$$

The Differential drive mobile robot forward kinematics will be discussed in the next section.

3.2.1 Forward kinematic model

Assume a differential drive mobile robot setup which has two wheels with the radius of R_a placed with a distance L from the robot center as shown in Figure 49:

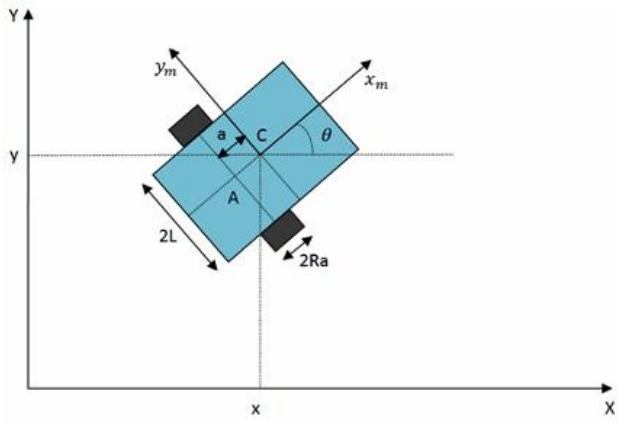


FIGURE 29 THE DIFFERENTIAL DRIVE MOBILE ROBOT

The following notations will be used in this thesis:

A: The intersection of the axis of symmetry with the driving wheels axis.

C: The center of mass of the platform.

a: The distance between the center of mass and driving wheels axis in x-direction.

L: The distance between each driving wheel and the robot axis of symmetry in y-direction.

R_a: The radius of each driving wheel.

$\dot{\phi}_R$: The rotational velocity of the right wheel.

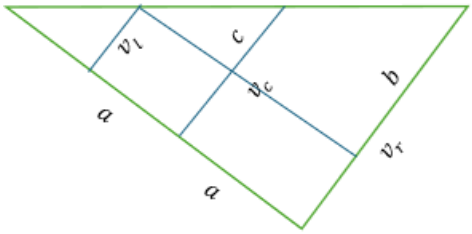
$\dot{\phi}_L$: The rotational velocity of the left wheel.

v: The translational velocity of the platform in the local frame.

ω: The rotational velocity of the platform in the local and global frames.

The forward kinematic problem can be described as the problem of finding the following function:

$$\dot{q} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = fn\left(\dot{\phi}_R, \dot{\phi}_L, L, R_a, \theta\right)$$



From Triangle Similarity:

$\frac{a}{2a} = \frac{c}{b}$ $\quad\quad$ $\therefore c = \frac{1}{2}b$

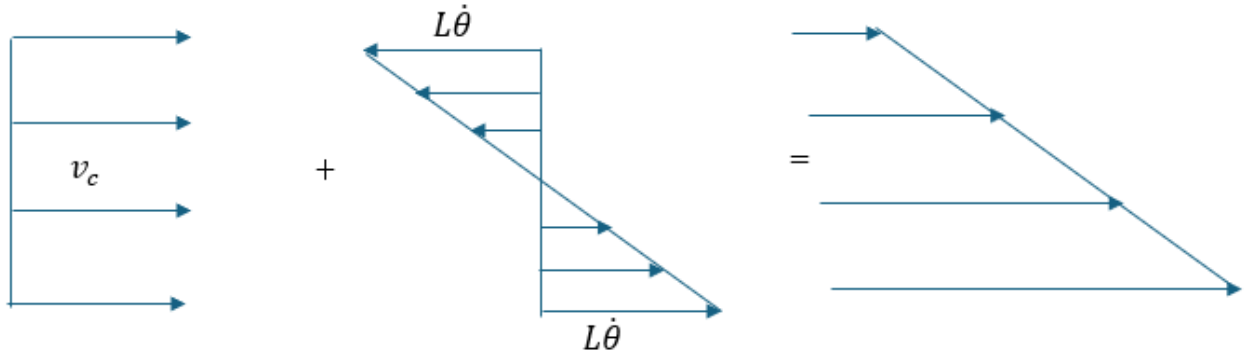
$v_c = v_l + c$ $\quad\quad$ $\therefore c = v_c - v_l \longrightarrow 1$

$v_r = b + v_l$ $\quad\quad$ $\therefore b = v_r - v_l \longrightarrow 2$

$\therefore c = \frac{v_r - v_l}{2} \longrightarrow 3$

from 1 & 3 :

$v_c - v_l = \frac{v_r - v_l}{2}$ $\quad\quad$ $\therefore v_c = \frac{v_r + v_l}{2}$



Velocity direction body | Velocity direction wheel | Overall velocity direction

$$v_r = v_c + L\dot{\theta} \longrightarrow 3$$

$$v_l = v_c - L\dot{\theta} \longrightarrow 4$$

$3 + 4$

$$v_c = \frac{v_r + v_l}{2}$$

$3 - 4$

$$\dot{\theta} = \frac{v_r - v_l}{2L}$$

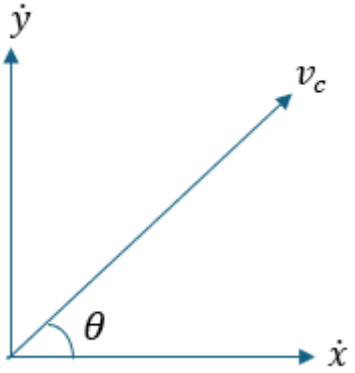
$$v_r = R_a \dot{\emptyset}_r$$

$$v_l = R_a \dot{\emptyset}_l$$

$$\dot{x} = v_c \cos\theta = \left(\frac{v_r + v_l}{2}\right)\cos\theta = \frac{R_a}{2}\left(\dot{\emptyset}_r + \dot{\emptyset}_l\right)\cos\theta$$

$$\dot{y} = v_c \sin\theta = \left(\frac{v_r + v_l}{2}\right)\sin\theta = \frac{R_a}{2}\left(\dot{\emptyset}_r + \dot{\emptyset}_l\right)\sin\theta$$

$$\dot{\theta} = \frac{v_r - v_l}{2L} = \frac{R_a}{2L}\left(\dot{\emptyset}_r - \dot{\emptyset}_l\right)$$



$$\dot{q} = Jv$$

$$\therefore \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \frac{1}{2}\cos\theta & \frac{1}{2}\cos\theta \\ \frac{1}{2}\sin\theta & \frac{1}{2}\sin\theta \\ \frac{1}{2L} & -\frac{1}{2L} \end{pmatrix} \begin{pmatrix} v_r \\ v_l \end{pmatrix}$$

$$\therefore \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \frac{R_a}{2}\cos\theta & \frac{R_a}{2}\cos\theta \\ \frac{R_a}{2}\sin\theta & \frac{R_a}{2}\sin\theta \\ \frac{R_a}{2L} & -\frac{R_a}{2L} \end{pmatrix} \begin{pmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{pmatrix}$$

3.2.2 Inverse kinematic model:

The inverse kinematic problem can be described as the problem of finding the following function:

$$\begin{pmatrix} \dot{\phi}_R \\ \dot{\phi}_L \end{pmatrix} = fn(\dot{x}, \dot{y}, \dot{\theta})$$

$$v = J\dot{q}$$

if : $v_r = R_a\dot{\phi}_r$ $\qquad$ $v_l = R_a\dot{\phi}_l$

$v_r = v_c + L\dot{\theta}$ $\qquad$ $v_l = v_c - L\dot{\theta}$

$\therefore R_a\dot{\phi}_r = v_c + L\dot{\theta}$ $\qquad$ $\therefore R_a\dot{\phi}_l = v_c - L\dot{\theta}$

$$v_c = \dot{x}\cos\theta + \dot{y}\sin\theta$$

$$\dot{\phi}_r = \frac{1}{R_a}\left(\dot{x}\cos\theta + \dot{y}\sin\theta + L\dot{\theta}\right)$$

$$\dot{\phi}_l = \frac{1}{R_a}\left(\dot{x}\cos\theta + \dot{y}\sin\theta - L\dot{\theta}\right)$$

$$\begin{pmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{pmatrix} = \frac{1}{R_a}\begin{pmatrix} \cos\theta & \sin\theta & L \\ \cos\theta & \sin\theta & -L \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix}$$

Instantaneous radius :

$$R = \frac{v_c}{\dot{\theta}} = L\left(\frac{v_r + v_l}{v_r - v_l}\right) \quad , \qquad v_r \geq v_l$$

Instantaneous coefficient :

$$K = \frac{1}{R}$$

3.3 KINETIC MODELING OF THE MOBILE ROBOT

3.3.1-wheel speed to vehicle speed

Assume the center of the wheel is the same center of the mobile robot $(a = 0)$.

velocity at center:

$$V_A = \begin{pmatrix} \dot{x} \\ \dot{y} \\ 0 \end{pmatrix}$$

velocity at right wheel:

$$\overrightarrow{V_r} = \overrightarrow{V_A} + \vec{\omega} \times R\overrightarrow{r_r}$$

where:

- $\vec{\omega} = \dot{\theta}.\hat{k} \quad k\ is\ unit\ vector$

$$\dot{\theta}.\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \dot{\theta} \end{pmatrix}$$

- R about z-axis

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- $\overrightarrow{r_r}$ distance from the center of mobile robot to right wheel

$$\begin{pmatrix} 0 \\ -L \\ 0 \end{pmatrix}$$

$$\therefore R.\overrightarrow{r_r} = \begin{pmatrix} cos\,\theta & -sin\,\theta & 0 \\ sin\,\theta & cos\,\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ -L \\ 0 \end{pmatrix} = \begin{pmatrix} L\,sin\,\theta \\ -L\,cos\,\theta \\ 0 \end{pmatrix}$$

$$\therefore \vec{\omega} \times R\overrightarrow{r_r} = \begin{pmatrix} +i & -j & +k \\ 0 & 0 & \dot{\theta} \\ L\,sin\,\theta & -L\,cos\,\theta & 0 \end{pmatrix} = \begin{pmatrix} L\dot{\theta}\,cos\,\theta \\ L\dot{\theta}\,sin\,\theta \\ 0 \end{pmatrix}$$

$$\therefore \overrightarrow{V_r} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ 0 \end{pmatrix} + \begin{pmatrix} L\dot{\theta}\,cos\,\theta \\ L\dot{\theta}\,sin\,\theta \\ 0 \end{pmatrix} = \begin{pmatrix} \dot{x} + L\dot{\theta}\,cos\,\theta \\ \dot{y} + L\dot{\theta}\,sin\,\theta \\ 0 \end{pmatrix}$$

using a similar analysis, the velocity of the second wheel is :

$$\overrightarrow{V_l} = \overrightarrow{V_A} + \vec{\omega} \times R\overrightarrow{r_l}$$

where:

$$\vec{\omega} = \dot{\theta}.\hat{k} \quad k\ is\ unit\ vector$$

$$\dot{\theta}.\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \dot{\theta} \end{pmatrix}$$

- R about z-axis

$$\begin{pmatrix} cos\,\theta & -sin\,\theta & 0 \\ sin\,\theta & cos\,\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- $\overrightarrow{r_l}$ distance from the center of mobile robot to left wheel

$$\begin{pmatrix} 0 \\ L \\ 0 \end{pmatrix}$$

$$\therefore R.\overrightarrow{r_l} = \begin{pmatrix} cos\,\theta & -sin\,\theta & 0 \\ sin\,\theta & cos\,\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ L \\ 0 \end{pmatrix} = \begin{pmatrix} -L\,sin\,\theta \\ L\,cos\,\theta \\ 0 \end{pmatrix}$$

$$\therefore \vec{\omega} \times \overrightarrow{r_l} = \begin{pmatrix} +i & -j & +k \\ 0 & 0 & \dot{\theta} \\ -L\,sin\,\theta & L\,cos\,\theta & 0 \end{pmatrix} = \begin{pmatrix} -L\dot{\theta}\,cos\,\theta \\ -L\dot{\theta}\,sin\,\theta \\ 0 \end{pmatrix}$$

$$\therefore \overrightarrow{V_l} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ 0 \end{pmatrix} + \begin{pmatrix} -L\dot{\theta}\,cos\,\theta \\ -L\dot{\theta}\,sin\,\theta \\ 0 \end{pmatrix} = \begin{pmatrix} \dot{x} - L\dot{\theta}\,cos\,\theta \\ \dot{y} - L\dot{\theta}\,sin\,\theta \\ 0 \end{pmatrix}$$

3.4 The kinematic constraints

The following assumptions about the wheel motion will cause the robot kinematic constraints:

- Movement on a horizontal plane.
- Point contact between the wheels and ground.
- Wheels are not deformable.
- Pure rolling which means that we have the instantaneous center of zero velocity at the contact point of the wheels and ground.
- the contact point of the wheels and ground.
- No slipping, skidding or sliding.
- No friction for rotation around contact points.
- Steering axes orthogonal to the surface.
- Wheels are connected by a rigid frame (chassis).

Considering the above assumptions about the wheel motion, the robot will have a special kind of constraint called Nonholonomic constraint. A nonholonomic constraint is a constraint on the feasible velocities of a body. For the case of the differential drive mobile robot, it can simply mean that the robot can move in some directions (Forward and backward) but not others (Sideward) as is shown in Figure 50:

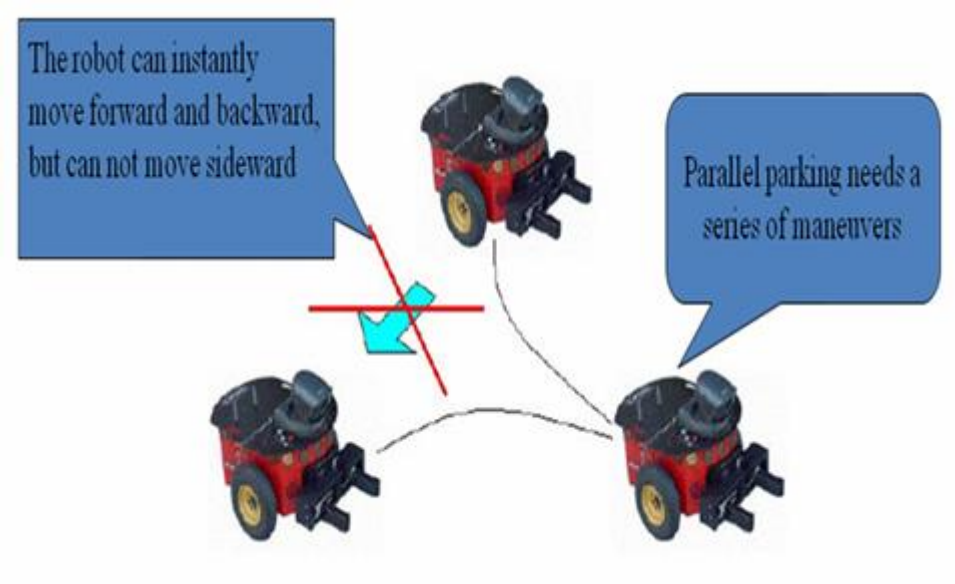


FIGURE 30 THE NONHOLONOMIC CONSTRAINT IN THE ROBOT MOTION

Having this kind of constraints in a system will cause some challenges in the motion planning and control of such systems which will be explained in the control algorithm section of this thesis. The equations of the nonholonomic constraints for the differential drive mobile robot shown in figure 31 are as follows:

- No lateral slip constraint:

$$\dot{y}_c \cos\theta - \dot{x}_c \sin\theta - \dot{\theta} a = 0$$

$\dot{y}_c$ and $\dot{x}_c$ are the robot velocity components in the inertial frame. This constraint means that the velocity of the robot center point will be in the direction of the axis of symmetry and the motion in the orthogonal plane will be zero.

- Pure rolling constraint:

$$\dot{x}_c \cos\theta + \dot{y} \sin\theta + L\dot{\theta} = R_a \dot{\emptyset}_R$$

$$\dot{x}_c \cos\theta + \dot{y} \sin\theta - L\dot{\theta} = R_a \dot{\emptyset}_L$$

This constraint shows that the driving wheels do not slip. The three constraints can be written in the following form:

$$A(q)\dot{q} = 0$$

In which:

$$A(q) = \begin{pmatrix} -\sin\theta & \cos\theta & a & 0 & 0 \\ \cos\theta & \sin\theta & L & -R_a & 0 \\ \cos\theta & \sin\theta & -L & 0 & -R_a \end{pmatrix}$$

$$q = \begin{pmatrix} x \\ y \\ \theta \\ \emptyset_r \\ \emptyset_l \end{pmatrix}, \qquad \dot{q} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\emptyset}_r \\ \dot{\emptyset}_l \end{pmatrix}$$

The above representation of the nonholonomic constraint is useful when we want to take the constraints in to account in the dynamic modeling. The system dynamic modeling which is the other important part of the robot modeling will be discussed in the next section.

3.5 DYNAMIC MODELING OF THE MOBILE ROBOT

Formulation of equations of motion, or dynamics, of a robot is essential in the analysis, design and control of a robot. Dynamic modeling in general is the study of the system's motion in which forces are modeled and it can include energies and the speeds associated with the motions. The main difference between dynamic and kinematic modeling is that in kinematics we study the motion without considering the forces that affect the motion and we just deal with the geometric relationships that govern the system.

A mobile robot system having an n-dimensional configuration space L with generalized coordinates $(q_1, q_2, \dots, q_n)$ and subject to m constraints can be described by the following general dynamic equation:

$$M(q)\ddot{q} + V(q,\dot{q}) + F(\dot{q}) + G(q) + \tau d = B(q)\tau - A^T(q)\lambda$$

Where:

$M(q)$: The symmetric positive definite inertia matrix.

$V(q,\dot{q})$: The centripetal and Coriolis matrix.

$F(\dot{q})$: The friction matrix.

$G(q)$: The gravitational vector.

τ_d: Denoted bounded unknown disturbances including unstructured unmodeled dynamics.

$B(q)$: The input transformation matrix.

τ: The input vector.

$A^T(q)$: The matrix associated with the constraints.

λ: The vector of the constraint forces.

In the following two sections, the energy-based Lagrangian approach and the direct, Newton-Euler, vector mechanics approach are outlined for expressing the dynamics of this robot. The first approach is relatively more convenient to formulate and implement but the tradeoff is that physical insight and part of the useful information are lost in this process.

3.6 Lagrangian Dynamics approach

Analytical dynamics is an approach in dynamics which treats the system as a whole dealing with scalar quantities such as the kinetic and potential energies of the system. Lagrange (1736-1813) proposed an approach which provides a powerful and versatile method for the formulation of the equations of motion for any dynamical system. Lagrange's equations are differential equations in which one considers the energies of the system and the work done instantaneously in time. The derivation of the Lagrange equation for holonomic systems requires that the generalized coordinates be independent. For a nonholonomic system, however, there must be more number of generalized coordinates than the number of degrees of freedom which is because of the constraints on the motion of the system.

General form of Lagrange equation:

$$\frac{d}{dt}\left(\frac{dl}{d\dot{q}_i}\right) - \frac{\partial l}{\partial q_i} = \sum_{j=1}^{n} \lambda_j a_{ji} + Q_i \quad , \quad i = 1,2,\dots\dots,n$$

$L = K.E - P.E$ is the Lagrangian which is the difference between the systems kinetic and potential energy.

- **KINETIC ENERGY**

$$K.E = K.E_{chassis} + K.E_{right\ wheel} + K.E_{left\ wheel} = \frac{1}{2}mv^2 + \frac{1}{2}Iw^2$$

1) Chassis :

$$K.E_{chassis} = \frac{1}{2}m_c\|v_c\|^2 + \frac{1}{2}I_c{w_c}^2$$

- $\overrightarrow{v_c} = \overrightarrow{v_A} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ 0 \end{pmatrix}$ — the center of wheel is the same as the center of chassis; symmetric
- $w_c = \dot{\theta}.\hat{k} = \begin{pmatrix} 0 \\ 0 \\ \dot{\theta} \end{pmatrix}$

$$K.F_{chassis} = \frac{1}{2}m_c\begin{pmatrix} \dot{x} & \dot{y} & 0 \end{pmatrix}\begin{pmatrix} \dot{x} \\ \dot{y} \\ 0 \end{pmatrix} + \frac{1}{2}I_c\begin{pmatrix} 0 & 0 & \dot{\theta} \end{pmatrix}\begin{pmatrix} 0 \\ 0 \\ \dot{\theta} \end{pmatrix}$$

$$\therefore K.F_{chassis} = \frac{1}{2}m_c(\dot{x}^2 + \dot{y}^2) + \frac{1}{2}I_c(\dot{\theta}^2)$$

$$= \frac{1}{2}\left(m_c(\dot{x}^2 + \dot{y}^2) + I_c(\dot{\theta}^2)\right)$$

2) Right wheel

$$K.E_{right\ wheel} = \frac{1}{2} m_w \|v_r\|^2 + \frac{1}{2} I_w {\Omega_w}^2$$

$$\Omega_w = \dot{\phi}_r \begin{pmatrix} \sin\theta \\ \cos\theta \\ 0 \end{pmatrix} + \dot{\theta} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \dot{\phi}_r \sin\theta \\ \dot{\phi}_r \cos\theta \\ \dot{\theta} \end{pmatrix}$$

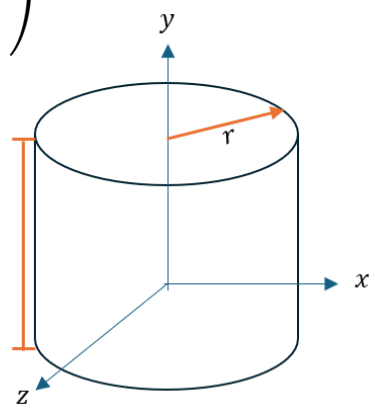
$$I_w = R_{12}\, I_w^b\, R_{12}^T$$

$$I_w^b = \begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix},$$

$$I_{xx} = \frac{m}{12}(3r^2 + t^2)$$

$$I_{yy} = \frac{m}{2} r^2$$

$$I_{zz} = \frac{m}{12}(3r^2 + t^2)$$



$$I_w = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} I_{xx}\cos\theta^2 + I_{yy}\sin\theta^2 & I_{xx}\cos\theta\sin\theta - I_{yy}\cos\theta\sin\theta & 0 \\ I_{xx}\cos\theta\sin\theta - I_{yy}\cos\theta\sin\theta & I_{xx}\sin\theta^2 + I_{yy}\cos\theta^2 & 0 \\ 0 & 0 & I_{zz} \end{pmatrix}$$

$$V_r = \begin{pmatrix} \dot{x} + L\dot{\theta}\cos\theta \\ \dot{y} + L\dot{\theta}\sin\theta \\ 0 \end{pmatrix}$$

$$\therefore K.E_{right\ wheel} = \frac{1}{2} m_w \|v_r\|^2 + \frac{1}{2} \Omega_w^T I_w {\Omega_w}^2$$

$$\therefore K.E_{right\ wheel}$$

$$= \frac{1}{2} m_w \left(\dot{x}^2 + \dot{y}^2 + L^2\dot{\theta}^2 + 2L\dot{\theta}(\dot{x}\cos\theta + \dot{y}\sin\theta)\right) + \frac{I_{zz}}{2}\dot{\theta}^2 + \frac{I_{yy}}{2}\dot{\phi}_r^2$$

3) Left wheel

$$K.E_{left\ wheel} = \frac{1}{2} m_w \|v_l\|^2 + \frac{1}{2} \Omega_w^T I_w \Omega_w$$

$$\Omega_w = \dot{\phi}_l \begin{pmatrix} \sin\theta \\ \cos\theta \\ 0 \end{pmatrix} + \dot{\theta} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \dot{\phi}_l \sin\theta \\ \dot{\phi}_l \cos\theta \\ \dot{\theta} \end{pmatrix}$$

$$I_w = R_{12}\, I_w^b\, R_{12}^T$$

$$I_w^b = \begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix},$$

$$I_{xx} = \frac{m}{12}(3r^2 + t^2)$$

$$I_{yy} = \frac{m}{2} r^2$$

$$I_{zz} = \frac{m}{12}(3k^2 + t^2)$$

$$I_w = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} I_{xx}\cos\theta^2 + I_{yy}\sin\theta^2 & I_{xx}\cos\theta\sin\theta - I_{yy}\cos\theta\sin\theta & 0 \\ I_{xx}\cos\theta\sin\theta - I_{yy}\cos\theta\sin\theta & I_{xx}\sin\theta^2 + I_{yy}\cos\theta^2 & 0 \\ 0 & 0 & I_{zz} \end{pmatrix}$$

$$V_l = \begin{pmatrix} \dot{x} - L\dot{\theta}\cos\theta \\ \dot{y} - L\dot{\theta}\sin\theta \\ 0 \end{pmatrix}$$

$$\therefore K.E_{left\ wheel}$$

$$= \frac{1}{2} m_w \left(\dot{x}^2 + \dot{y}^2 + L^2\dot{\theta}^2 - 2L\dot{\theta}(\dot{x}\cos\theta + \dot{y}\sin\theta)\right)$$

$$+ \frac{I_{zz}}{2}\dot{\theta}^2 + \frac{I_{yy}}{2}\dot{\phi}_l^2$$

The same as the right wheel.

3.7 Total Kinetic Energy

$$K.E_{total} = K.E_{chassis} + K.E_{right\ wheel} + K.E_{left\ wheel}$$

$$K.E_{total} = \frac{1}{2}\left(m_c(\dot{x}^2 + \dot{y}^2) + I_c(\dot{\theta}^2)\right) + \frac{1}{2}m_w\left(\dot{x}^2 + \dot{y}^2 + L^2\dot{\theta}^2 + 2L\dot{\theta}(\dot{x}\cos\theta + \dot{y}\sin\theta)\right) + \frac{I_{zz}}{2}\dot{\theta}^2 + \frac{I_{yy}}{2}\dot{\emptyset}_r^2 + \frac{1}{2}m_w\left(\dot{x}^2 + \dot{y}^2 + L^2\dot{\theta}^2 - 2L\dot{\theta}(\dot{x}\cos\theta + \dot{y}\sin\theta)\right) + \frac{I_{zz}}{2}\dot{\theta}^2 + \frac{I_{yy}}{2}\dot{\emptyset}_l^2$$

$$K.E_{total} = \frac{1}{2}(m_c + 2m_w)(\dot{x}^2 + \dot{y}^2) + \frac{1}{2}(I_c + 2Lm_w + 2I_{zz})\dot{\theta}^2 + \frac{I_{yy}}{2}(\dot{\emptyset}_r^2 + \dot{\emptyset}_l^2)$$

$$let: m_t = m_c + 2m_w$$

$$I_t = I_c + 2Lm_w + 2I_{zz}$$

$$\therefore K.E_{total} = \frac{1}{2}m_t(\dot{x}^2 + \dot{y}^2) + \frac{1}{2}I_t\dot{\theta}^2 + \frac{1}{2}I_{yy}(\dot{\emptyset}_r^2 + \dot{\emptyset}_l^2)$$

- **POTENTIAL ENERGY**

$$P.E_{total} = P.E_{chassis} + P.E_{right\ wheel} + P.E_{left\ wheel}$$

$$P.E_{total} = (m_c + 2m_w)gy\sin\gamma = m_t gy\sin\gamma$$

Where γ is the inclination angle of the wall.

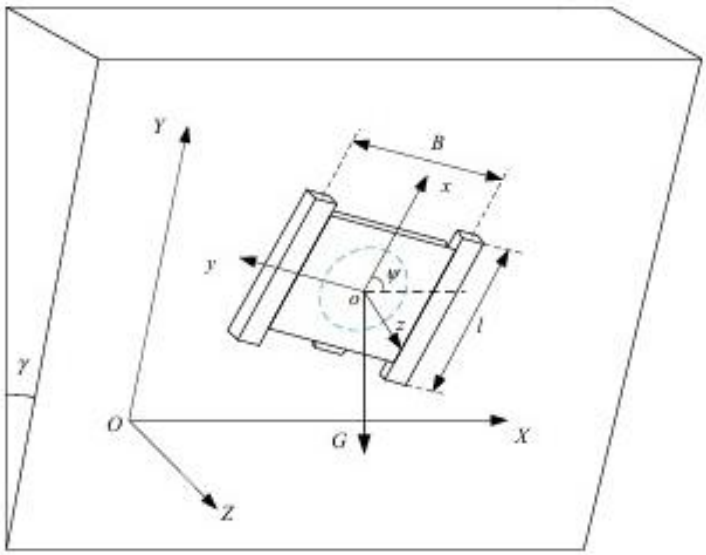


FIGURE 31

$$L = K.E - P.E$$
$$= \frac{1}{2} m_t (\dot{x}^2 + \dot{y}^2) + \frac{1}{2} I_t \dot{\theta}^2 + \frac{1}{2} I_{yy} (\dot{\phi}_r^2 + \dot{\phi}_l^2)$$
$$- m_t g y \sin\gamma$$

The generalized coordinates to use in the Lagrange formulation are as follows:

$$q = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}, \qquad \dot{q} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix}$$

The step by step approach to find the dynamic equations using the above generalized coordinates and Lagrangian is as follows:

$$\frac{\partial L}{\partial \dot{x}} = m\dot{x}$$

$$\frac{\partial L}{\partial \dot{y}} = m\dot{y}$$

$$\frac{\partial L}{\partial \dot{\theta}} = I\dot{\theta}$$

The time derivatives of the above three equations are as follows:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) = m\ddot{x}$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{y}}\right) = m\ddot{y}$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) = I\ddot{\theta}$$

The rest of the derivatives to complete the Lagrange equations are as follows:

$$\frac{\partial L}{\partial x} = 0$$

$$\frac{\partial L}{\partial y} = -m_t g \sin\gamma$$

$$\frac{\partial L}{\partial \theta} = 0$$

Substituting the above term in the Lagrange equation, we have:

$$m\ddot{x} = F_x + C_x$$

$$m\ddot{y} = F_y + C_y$$

$$I\ddot{\theta} = \tau + C_\theta$$

Where:

- F_x is the actuator force in the x-direction.
- F_y is the actuator force in the y-direction.
- τ is the actuator rotational torque on the robot.
- C_x, C_y, C_θ is the constraint forces in the x, y, θ directions.

$$\begin{pmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & I \end{pmatrix} \ddot{q} = \begin{pmatrix} F_x \\ F_y \\ \tau \end{pmatrix} + \begin{pmatrix} C_x \\ C_y \\ C_\theta \end{pmatrix}$$

We can relate the forces in the x, y and θ directions to the actuator torques on each wheel according to the free body diagram of the robot shown in Figure 52:

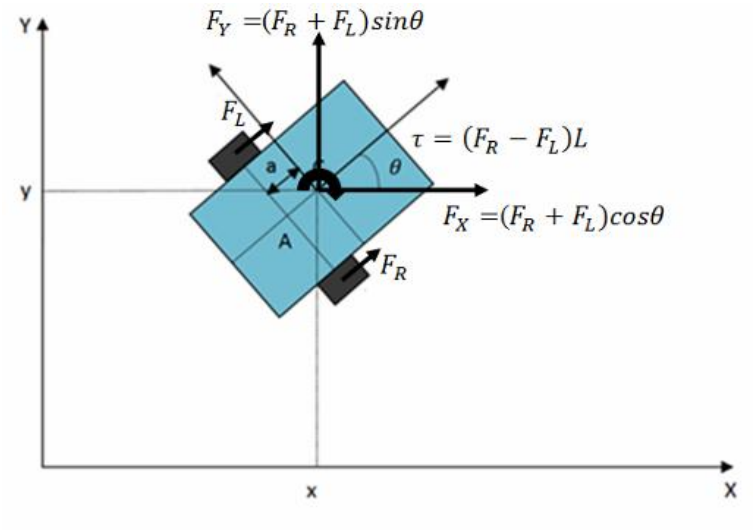


FIGURE 32 THE ROBOTS FREE BODY DIAGRAM

$$F_R = \frac{\tau_R}{R_a}$$

$$F_L = \frac{\tau_L}{R_a}$$

$$F_x = (F_R + F_L)\cos\theta = \frac{\tau_R + \tau_L}{R_a}\cos\theta$$

$$F_y = (F_R + F_L)\sin\theta = \frac{\tau_R + \tau_L}{R_a}\sin\theta$$

$$\tau = (F_R - F_L)L = \left(\frac{\tau_R - \tau_L}{R_a}\right)L$$

According to the above equations, we can write the input force matrix in the system's dynamic equation as follows:

$$\therefore \begin{pmatrix} F_x \\ F_y \\ \tau \end{pmatrix} = \frac{1}{R_a}\begin{pmatrix} \cos\theta & \cos\theta \\ \sin\theta & \sin\theta \\ L & -L \end{pmatrix}\begin{pmatrix} \tau_R \\ \tau_L \end{pmatrix}, \quad B(q) = \frac{1}{R_a}\begin{pmatrix} \cos\theta & \cos\theta \\ \sin\theta & \sin\theta \\ L & -L \end{pmatrix}$$

As it is mentioned in the previous section the nonholonomic constraints of this system are:

$$\dot{y}_c \cos\theta - \dot{x}_c \sin\theta - \dot{\theta} a = 0$$

$$\dot{x}_c \cos\theta + \dot{y} \sin\theta + L\dot{\theta} = R_a \dot{\emptyset}_R$$

$$\dot{x}_c \cos\theta + \dot{y} \sin\theta - L\dot{\theta} = R_a \dot{\emptyset}_L$$

According to these constraint equations the constraint forces will become:

$$C_x = m(\dot{x}_c \cos\theta + \dot{y}_c \sin\theta)\dot{\theta} \sin\theta$$

$$C_y = -m(\dot{x}_c \cos\theta + \dot{y}_c \sin\theta)\dot{\theta} \cos\theta$$

$$C_z = ma(\dot{x}_c \cos\theta + \dot{y}_c \sin\theta)\dot{\theta}$$

The matrix representation of the above constraint forces is as follows:

$$\begin{pmatrix} C_x \\ C_y \\ C_z \end{pmatrix} = \begin{pmatrix} m(\dot{x}_c \cos\theta + \dot{y}_c \sin\theta)\dot{\theta} \sin\theta \\ -m(\dot{x}_c \cos\theta + \dot{y}_c \sin\theta)\dot{\theta} \cos\theta \\ ma(\dot{x}_c \cos\theta + \dot{y}_c \sin\theta)\dot{\theta} \end{pmatrix} = A^T(q)\lambda$$

Where: $A^T = \begin{pmatrix} -\sin\theta \\ \cos\theta \\ a \end{pmatrix}$ $\lambda = -m(\dot{x}_c \cos\theta + \dot{y}_c \sin\theta)\dot{\theta}$

3.7.1 Friction Force:

Let ω_1 and ω_2 be the angular of the sprockets and r be the pitch radius of the sprockets. The slips for the tracks on both sides can be calculated by:

$$i_1 = 1 - \frac{\dot{x} - L\dot{\theta}}{r\omega_1}$$

$$i_2 = 1 - \frac{\dot{x} + L\dot{\theta}}{r\omega_2}$$

Let F_1 and F_2 be the longitudinal tractive force of the tracks:

$$F_1 = \mu F_{N_1}\left(1 - \frac{K}{i_1 l}\left(1 - e^{-\frac{i_1 l}{K}}\right)\right)$$

$$F_2 = \mu F_{N_2}\left(1 - \frac{K}{i_2 l}\left(1 - e^{-\frac{i_2 l}{K}}\right)\right)$$

F_y is the lateral resistance force of the track:

$$F_y = 2\sin\dot{\theta}\,\mu_t a\frac{F_{N_1} + F_{N_2}}{l}$$

Where μ_t is lateral resistance coefficient. The turning moment M generated by the longitudinal force is:

$$M = -(F_1 - R_1)L + (F_2 - R_2)L$$

The turning resistance moment caused by the lateral resistance can be expressed as:

$$M_r = \sin\dot{\theta}\,\mu_t\frac{F_{N_1} + F_{N_2}}{l}\left(\frac{l^2}{4} - a^2\right)$$

The dynamics model of the wall-climbing robot is formulated in $x0y$ as:

$$m\ddot{x} = m\dot{y}\dot{\theta} + (F_1 + F_2) - (R_1 + R_2) - G\cos\gamma\sin\theta$$

$$m\ddot{y} = -m\dot{x}\dot{\theta} + F_y - G\cos\gamma\sin\theta$$

$$I_z\ddot{\theta} = M - M_r$$

Convert to inertial coordinate system:

$$m\ddot{X} = m\dot{Y}\dot{\theta} + (F_1 + F_2)\cos\theta - (R_1 + R_2)\cos\theta - F_y \sin\theta$$

$$m\ddot{Y} = m\dot{X}\dot{\theta} + (F_1 + F_2)\sin\theta - (R_1 + R_2)\sin\theta + F_y \cos\theta - G\cos\gamma$$

$$I_z\ddot{\theta} = M - M_r$$

$$\therefore F(q) = \begin{pmatrix} -F_y \sin\theta + (R_1 + R_2)\cos\theta \\ -F_y \cos\theta + (R_1 + R_2)\sin\theta \\ M_r + (R_1 - R_2)L \end{pmatrix}$$

$$G(q) = \begin{pmatrix} 0 \\ mg\sin\gamma \\ 0 \end{pmatrix}$$

$$V(q) = 0$$

$\tau_d = 0$ considered to be zero in this derivation

$$\tau = \begin{pmatrix} \tau_R \\ \tau_L \end{pmatrix}$$

$$M(q)\ddot{q} + V(q,\dot{q}) + F(\dot{q}) + G(q) + \tau d = B(q)\tau - A^T(q)\lambda$$

$$\therefore \begin{pmatrix} M & 0 & 0 \\ 0 & M & 0 \\ 0 & 0 & I \end{pmatrix}\begin{pmatrix} \ddot{x}_c \\ \ddot{y}_c \\ \ddot{\theta} \end{pmatrix} + \begin{pmatrix} -F_y \sin\theta + (R_1 + R_2)\cos\theta \\ -F_y \cos\theta + (R_1 + R_2)\sin\theta \\ M_r + (R_1 - R_2)L \end{pmatrix} + \begin{pmatrix} 0 \\ mg\sin\gamma \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} \cos\theta & \cos\theta \\ \sin\theta & \sin\theta \\ L & -L \end{pmatrix}\begin{pmatrix} \tau_R \\ \tau_L \end{pmatrix} - \begin{pmatrix} -\sin\theta \\ \cos\theta \\ a \end{pmatrix}\left(-M(\dot{x}_c\cos\theta + \dot{y}_c\sin\theta)\dot{\theta}\right)$$

The above system can be transformed into a more proper representation for control and simulation purposes. In this transformation, we are trying to find a way to eliminate the constraint term from the equation. The following two matrices are defined to do this transformation:

$$V(q) = \begin{pmatrix} V \\ \omega \end{pmatrix} = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} = \begin{pmatrix} \dot{x}_R \\ \dot{\theta} \end{pmatrix}$$

$$J(q) = \begin{pmatrix} \cos\theta & -a\sin\theta \\ \sin\theta & a\cos\theta \\ 0 & 1 \end{pmatrix}$$

By looking at the forward kinematic equation, one can realize that the $J(q)$ matrix is the modified forward kinematic matrix which has two velocity terms related to the distance between the robot centroid and wheel axis. Therefore, we can write the following equation for the system:

$$\dot{q} = \begin{pmatrix} \dot{x}_c \\ \dot{y}_c \\ \dot{\theta} \end{pmatrix} = J(q)V(q) = \begin{pmatrix} \cos\theta & -a\sin\theta \\ \sin\theta & a\cos\theta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} V \\ \omega \end{pmatrix}$$

It can easily be proved that the $J(q)$ has the following relation with $A(q)$ matrix:

$$J^T(q)A^T(q) = 0$$

The above equation is useful when we want to eliminate the constraint term from the main dynamic equation as you will see in the next step. Differentiating the equation, we have:

$$\dot{q} = J(q)v(t)$$

$$\ddot{q} = \dot{J}(q)v(t) + J(q)\dot{v}(t)$$

Substituting the above equation will result the following equation:

$$M(q)\left(\dot{J}(q)v(t) + J(q)\dot{v}(t)\right) + V_m(q,\dot{q})J(q)v(t) + F(\dot{q}) + G(q) + \tau_d = B(q)\tau - A^T(q)\lambda$$

$$M(q)\dot{J}(q)v(t) + M(q)J(q)\dot{v}(t) + V_m(q,\dot{q})J(q)v(t) + F(\dot{q}) + G(q) + \tau_d = B(q)\tau - A^T(q)\lambda$$

The next step to eliminate the constraint matrix $A^T(q)\lambda$ is to multiply the equation by $J^T(q)$ as follows:

$$\left(J^T(q)M(q)\dot{J}(q)\right)\dot{v}(t)+\left(J^T(q)M(q)\dot{J}(q)+J^T(q){V_m}^T(q,\dot{q})J(q)\right)v(t)+J^T(q)F(\dot{q})+J^T(q)\,G(q)$$
$$+J^T(q)\,\tau_d=J^T(q)\,B(q)\tau-J^T(q)\,A^T(q)\lambda$$

As it can be seen from the above equation, we have $J^T(q)\,A^T(q)$ which is zero. Therefore, the constraint term is eliminated, and the new dynamic equation is:

$$\left(J^T(q)M(q)\dot{J}(q)\right)\dot{v}(t)+\left(J^T(q)M(q)\dot{J}(q)+J^T(q){V_m}^T(q,\dot{q})J(q)\right)v(t)$$
$$+J^T(q)F(\dot{q})+J^T(q)\,G(q)+J^T(q)\,\tau_d=J^T(q)\,B(q)\tau$$

By the following appropriate definitions, we can rewrite equation as follows:

$$\bar{M}(q)\dot{v}(t)+\bar{V}_m(q,\dot{q})v(t)+\bar{F}(\dot{q})+\bar{G}(q)+\bar{\tau}_d=\bar{\tau}$$

Where:

$$\bar{\tau}=\bar{B}(q)\tau$$

$$\bar{M}(q)=J^TMJ=\begin{pmatrix}M & 0\\ 0 & Ma^2+I\end{pmatrix}\quad,\quad \bar{V}_m(q,\dot{q})=J^TM\dot{J}=\begin{pmatrix}0 & Ma\dot{\theta}\\ Ma\dot{\theta} & Ma\dot{a}\end{pmatrix}$$

$$\bar{F}(q,\dot{q})=J^TF=\begin{pmatrix}R_1+R_2\\ (R_1-R_2)L-F_ya+M_r\end{pmatrix}$$

$$\bar{G}(q)=J^TG=\begin{pmatrix}Mg\sin\gamma\sin\theta\\ -Mga\sin\gamma\cos\theta\end{pmatrix}\quad,\ \bar{B}(q)=J^TB=\begin{pmatrix}1 & 1\\ -L & L\end{pmatrix}$$

$$\begin{pmatrix}M & 0\\ 0 & Ma^2+I\end{pmatrix}\dot{v}(t)+\begin{pmatrix}0 & Ma\dot{\theta}\\ Ma\dot{\theta} & Ma\dot{a}\end{pmatrix}v(t)+\begin{pmatrix}R_1+R_2\\ (R_1-R_2)L\end{pmatrix}$$
$$+\begin{pmatrix}Mg\sin\gamma\sin\theta\\ -Mga\sin\gamma\cos\theta\end{pmatrix}=\begin{pmatrix}1 & 1\\ -L & L\end{pmatrix}\begin{pmatrix}\tau_R\\ \tau_L\end{pmatrix}$$

3.7.2 Newton-Euler Dynamics approach

The first and one of the most important steps in Newtonian dynamic modeling is to draw the free body diagram of the system and analyzing the forces on it. The free body diagram of the differential drive mobile robot is shown in Figure:

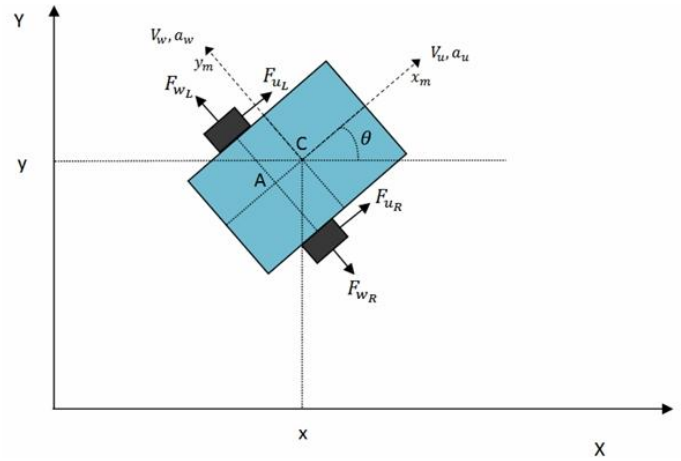


FIGURE 33 THE ROBOT'S FREE BODY DIAGRAM FOR NEWTONIAN DYNAMIC MODELING

The following notations are introduced in this figure and will be used for the Newtonian dynamic modeling:

(v_u, v_w): Represents the velocity of the vehicle in the local frame. v_u Is the longitudinal velocity and v_w is the lateral velocity.

(a_u, a_w): represent the acceleration of the vehicle's center of mass**.**

(f_{u_l}, f_{u_r}): are the longitudinal forces exerted on the vehicle by the left and right wheels.

(f_{w_l}, f_{w_r}): are the lateral forces exerted on the vehicle by the left and right wheels.

As it can be seen from the above free body diagram, the only forces acting on the robot are actuator forces acting on the robot wheels. We start the derivation by representing the robot position using polar coordinates. Assuming that the robot is a rigid body, its position can be represented using its angle and radius:

$$\hat{r} = re^{i\theta}$$

Differentiating the above position vector will give us the velocity and acceleration of the robot:

$$\dot{\hat{r}} = \dot{r}e^{i\theta} + r\dot{\theta}ie^{i\theta}$$

$$\ddot{\hat{r}} = \ddot{r}e^{i\theta} + \dot{r}\dot{\theta}ie^{i\theta} - r\dot{\theta}^2e^{i\theta} + \dot{r}\dot{\theta}ie^{i\theta} + r\ddot{\theta}ie^{i\theta}$$

Simplifying and writing the velocity and acceleration terms in radial and tangential terms, we have:

$$\dot{\hat{r}} = [\dot{r}]e^{i\theta} + [r\dot{\theta}]ie^{i\theta+\frac{\pi}{2}}$$

$$\ddot{\hat{r}} = [\ddot{r} - r\dot{\theta}^2]e^{i\theta} + [2\dot{r}\dot{\theta} + r\ddot{\theta}]ie^{i\theta+\frac{\pi}{2}}$$

The radial and tangential velocity and acceleration terms are defined as follows:

$$v_u = \dot{r}$$

$$v_w = r\dot{\theta}$$

$$a_u = \ddot{r} - r\dot{\theta}^2$$

$$a_w = 2\dot{r}\dot{\theta} + r\ddot{\theta}$$

From the above four equations, we can write the following relations between the radial and tangential velocity and acceleration of the robot:

$$a_u = \dot{v}_u - v_w\dot{\theta}$$

$$a_w = \dot{v}_w - v_u\dot{\theta}$$

The next step is to write the Newton's second law in radial (u) and tangential (w) directions and finding the relation between the forces and accelerations:

$$\sum F_u = ma_u$$

$$ma_u = F_{u_l} + F_{u_r}$$

$$\sum F_w = ma_w$$

$$ma_w = F_{w_l} + F_{w_r}$$

Substituting the acceleration terms from the equations:

$$\dot{v}_u = v_w\dot{\theta} + \frac{F_{u_l} + F_{u_r}}{m}$$

$$\dot{v}_w = -v_u\dot{\theta} + \frac{F_{w_l} + F_{w_r}}{m}$$

The above two equations show the acceleration of the robot in terms of the actuating forces and the velocity terms. The Newton's second law in rotation about the center of mass:

$$\sum M_C = I_C \ddot{\theta}$$

$$\ddot{\theta} = -\frac{(F_{w_l} + F_{w_r})a}{I_C} + \frac{(F_{u_l} - F_{u_r})L}{I_C}$$

The absence of slipping in the longitudinal (pure rolling) and lateral (no sliding) directions creates independence between the longitudinal, lateral and angular velocities and simplifies the dynamic equations.

In the absence of lateral slippage (no sliding constraint), the lateral velocity of the midpoint if the drive wheels is zero. Therefore, the lateral velocity of the center of mass will be:

$$v_w = a\dot{\theta}$$

And the lateral acceleration of the center of mass will be:

$$\dot{v}_w = a\ddot{\theta}$$

Combining equations:

$$a\ddot{\theta} = -v_u\dot{\theta} + \frac{F_{w_l} + F_{w_r}}{m}$$

$$m\left(a\ddot{\theta} + v_u\dot{\theta}\right) = F_{w_l} + F_{w_r}$$

Combining equations and solving for $\ddot{\theta}$ we have:

$$\ddot{\theta} = \frac{\left(F_{u_l} - F_{u_r}\right)L}{ma^2 + I_C} - \frac{mav_u\dot{\theta}}{ma^2 + I_C}$$

Combining equations:

$$\dot{v}_u = a\dot{\theta}^2 + \frac{F_{u_l} + F_{u_r}}{m}$$

The above two equations are the dynamic equations of the robot considering the nonholonomic constraints. The above two equations can easily be transformed to the matrix form using the same notations and matrices in the lagrangian approach:

$$\tau_r = \frac{F_r}{R_a}, \qquad \tau_l = \frac{F_l}{R_a}$$

$$(ma^2 + I_C)\ddot{\theta} + mav_u\dot{\theta} = \frac{(\tau_r - \tau_l)L}{R_a}$$

$$m\dot{v} - ma\dot{\theta}^2 = \frac{\tau_r + \tau_l}{R_a}$$

The matrix form of the above two equation is shown in the following equation:

$$\begin{pmatrix} m & 0 \\ 0 & ma^2 + I_C \end{pmatrix}\begin{pmatrix} \dot{v} \\ \ddot{\theta} \end{pmatrix} + \begin{pmatrix} 0 & -ma\dot{\theta} \\ ma\dot{\theta} & 0 \end{pmatrix}\begin{pmatrix} v \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \frac{1}{R_a} & \frac{1}{R_a} \\ \frac{L}{R_a} & \frac{-L}{R_a} \end{pmatrix}\begin{pmatrix} \tau_r \\ \tau_l \end{pmatrix}$$

As it can be seen from the above equation, both methods will reach the same dynamic equation for the mobile robot.

Differentiating the forward kinematic equations we have:

$$\ddot{X}_r = \dot{v}_u = \frac{\ddot{\phi}_r + \ddot{\phi}_l}{2}R_a$$

$$\ddot{\theta} = \frac{\ddot{\phi}_r - \ddot{\phi}_l}{2L}R_a$$

$$\frac{\ddot{\phi}_r - \ddot{\phi}_l}{2L}R_a = \frac{(F_{u_r} - F_{u_l})L}{ma^2 + I_A} - \frac{ma}{ma^2 + I_A}\left(\frac{\dot{\phi}_r + \dot{\phi}_l}{2}R_a\right)\left(\frac{\dot{\phi}_r - \dot{\phi}_l}{2L}R_a\right)$$

$$\frac{\ddot{\phi}_r + \ddot{\phi}_l}{2}R_a = a\left(\frac{\dot{\phi}_r - \dot{\phi}_l}{2L}R_a\right)^2 + \frac{F_{u_r} + F_{u_l}}{m}$$

$$F_{u_r} - F_{u_l} = \frac{R_a(ma^2 + I_A)}{2L^2}(\ddot{\phi}_r + \ddot{\phi}_l) + \frac{ma}{L}\left(\frac{\dot{\phi}_r + \dot{\phi}_l}{2}R_a\right)\left(\frac{\dot{\phi}_r - \dot{\phi}_l}{2L}R_a\right)$$

$$F_{u_r} + F_{u_l} = \frac{mR_a}{2}(\ddot{\phi}_r + \ddot{\phi}_l) - am\left(\frac{\dot{\phi}_r - \dot{\phi}_l}{2L}R_a\right)^2$$

Adding and simplifying the above two equation, we have:

$$F_{u_r} = \left(\frac{R_a(ma^2 + I_A)}{4L^2} + \frac{mR_a}{4}\right)\ddot{\phi}_r + \left(-\frac{R_a(ma^2 + I_A)}{4L^2} + \frac{mR_a}{4}\right)\ddot{\phi}_l - \left(\frac{maR_a{}^2}{4L^2}\right)\dot{\phi}_l{}^2 + \left(\frac{maR_a{}^2}{4L^2}\right)\dot{\phi}_r\dot{\phi}_l$$

$$F_{u_l} = \left(\frac{R_a(ma^2 + I_A)}{4L^2} + \frac{mR_a}{4}\right)\ddot{\phi}_l + \left(-\frac{R_a(ma^2 + I_A)}{4L^2} + \frac{mR_a}{4}\right)\ddot{\phi}_r - \left(\frac{maR_a{}^2}{4L^2}\right)\dot{\phi}_r{}^2 + \left(\frac{maR_a{}^2}{4L^2}\right)\dot{\phi}_r\dot{\phi}_l$$

Assuming that the system inputs are the DC motor torques, the system dynamic equations will be:

$$\tau_r = \left(\frac{R_a(ma^2 + I_A)}{4L^2} + \frac{mR_a{}^2}{4}\right)\ddot{\phi}_r + \left(-\frac{R_a{}^2(ma^2 + I_A)}{4L^2} + \frac{mR_a{}^2}{4}\right)\ddot{\phi}_l - \left(\frac{maR_a{}^3}{4L^2}\right)\dot{\phi}_l{}^2 + \left(\frac{maR_a{}^3}{4L^2}\right)\dot{\phi}_r\dot{\phi}_l$$

$$\tau_l = \left(\frac{R_a(ma^2 + I_A)}{4L^2} + \frac{mR_a{}^2}{4}\right)\ddot{\phi}_l + \left(-\frac{R_a{}^2(ma^2 + I_A)}{4L^2} + \frac{mR_a{}^2}{4}\right)\ddot{\phi}_r - \left(\frac{maR_a{}^3}{4L^2}\right)\dot{\phi}_r{}^2 + \left(\frac{maR_a{}^3}{4L^2}\right)\dot{\phi}_r\dot{\phi}_l$$

The following parameters were defined to simplify the rest of the calculations:

$$A = \frac{R_a(ma^2 + I_A)}{4L^2} + \frac{mR_a{}^2}{4}$$

$$B = -\frac{R_a(ma^2 + I_A)}{4L^2} + \frac{mR_a{}^2}{4}$$

$$C = \frac{maR_a{}^3}{4L^2}$$

Therefore, the dynamic equations can be written as follows:

$$\tau_r = A\ddot{\phi}_r + B\ddot{\phi}_l - C\dot{\phi}_l{}^2 + C\dot{\phi}_r\dot{\phi}_l$$

$$\tau_l = A\ddot{\phi}_l + B\ddot{\phi}_r - C\dot{\phi}_r{}^2 + C\dot{\phi}_r\dot{\phi}_l$$

3.7.3 Newton's law Dynamics approach

Dynamic model of mobile robot can be presented as it includes the effects of robot's overall mass m and overall moment of inertia J . It can be obtained using the second Newton's law for forces.

$$F_i = ma_i$$

In application of the previous equation to wheel motor's traction forces F_R, F_L it is possible to express the overall angular acceleration as:

$$\dot{v} = \frac{F_R}{m} + \frac{F_L}{m}$$

Traction forces F_R, F_L can be defined also as wheel torques, while the overall torque of the robot is defined as a sum of wheel partial torques $\tau_R = F_R L$, $\tau_R = -F_L L$ as:

$$\tau_{robot} = \tau_R + \tau_L$$

Where opposite direction of F_L is required to satisfy the last equation. Again, using the second Newton's law, it is possible to express τ_{robot} as function of $\dot{\omega}$ as:

$$\tau_{robot} = J\dot{\omega}$$

Where the J is robot's overall moment of inertia. Angular acceleration $\dot{\omega}$ can be also expressed as a function of traction forces

$$\dot{\omega} = \frac{F_R L}{J} - \frac{F_L L}{J}$$

Based on the relations, it is possible to obtain the mobile robot's dynamic model defined for wheel motor's traction forces F_R, F_L , that can be implemented in simulation model of mobile robot (Figure 53).

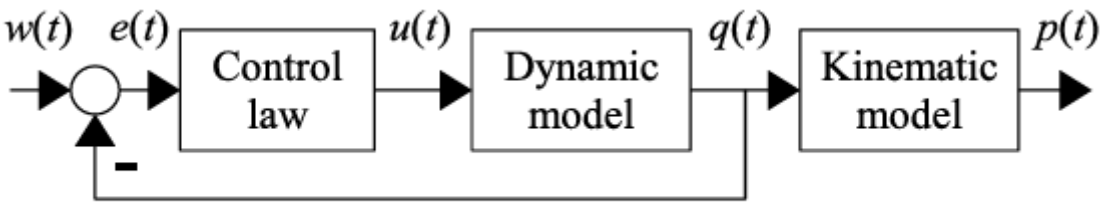


FIGURE 34 MOBILE ROBOT WITH INTERNAL CONTROL LOOP

3.8 MOBILE ROBOT CONTROL AND SIMULATION

SYSTEM SIMULATION

Simulation has been recognized as an important research tool since the beginning of the 20th century and now the simulation is a powerful visualization, planning, and strategic tool in different areas of research and development. The simulation has also a very important role in robotics. Different tools are used for the analysis of kinematics and dynamics of robotic systems, for off-line programming, to design different control algorithms, to design mechanical structure of robots, to design robotic cells and production lines, etc. However, a suitable computer package is needed to do the task of simulation for robotic systems. MATLAB and MATLAB Simulink are the two powerful software's which have been used broadly in the field of robotics research, and we used them in this project to do the simulations. The Simulink block diagram of the different parts of the robot model which has been derived in the previous sections of this chapter are shown and explained separately in this section.

Kinematic Model

This vehicle has two independently driven wheels which can be used to control forward and angular velocity.

Inputs:

- Left and right wheel speeds ω_L and ω_R, in rad/s

Outputs

- Linear velocity *v*, in m/s
- Angular velocity *ω*, in rad/s

Forward Kinematics

$$v = \frac{R}{2}(\omega_R + \omega_L), \qquad \omega = \frac{R}{L}(\omega_R - \omega_L)$$

Inverse Kinematics

$$\omega_L = \frac{1}{R}\left(v - \frac{\omega L}{2}\right), \quad \omega_R = \frac{1}{R}\left(v + \frac{\omega L}{2}\right)$$

The code of the simulation will be in the appendix.

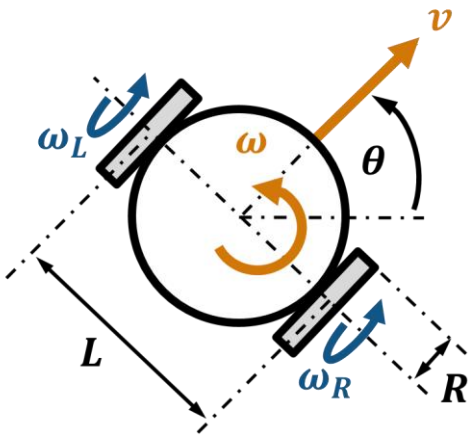


FIGURE 35 KINEMATIC MODEL

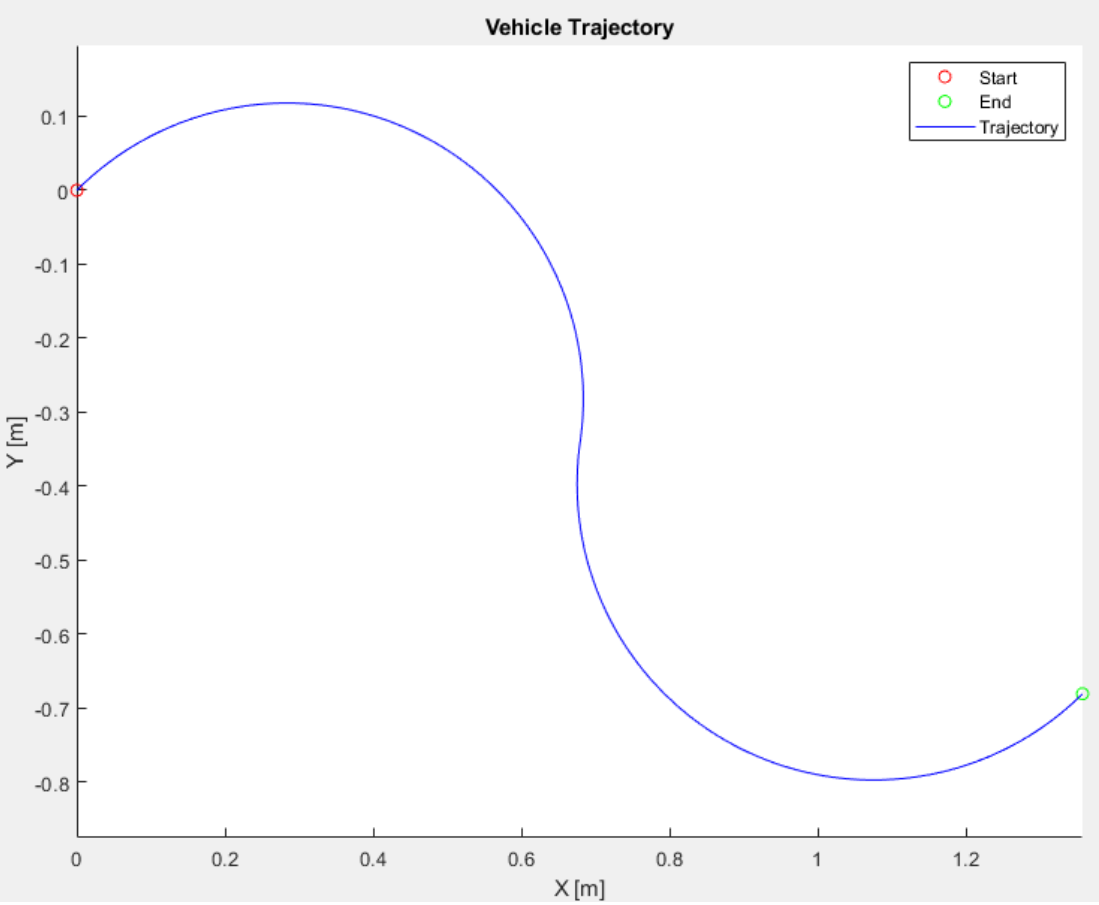


FIGURE 36 DIFFERENTIAL DRIVE SIMULATION USED TO CONTROL FORWARD AND ANGULAR VELOCITY

Simulink Usage

Simulink blocks are in the **Kinematic Models > Differential Drive** section of the block library.

- Use the **Differential Drive Forward Kinematics** and **Differential Drive Inverse Kinematics** blocks to convert between body velocities and wheel velocities.
- Use the **Differential Drive Simulation** block to simulate the pose given wheel speeds as inputs. You can configure the initial pose and simulation sample time.

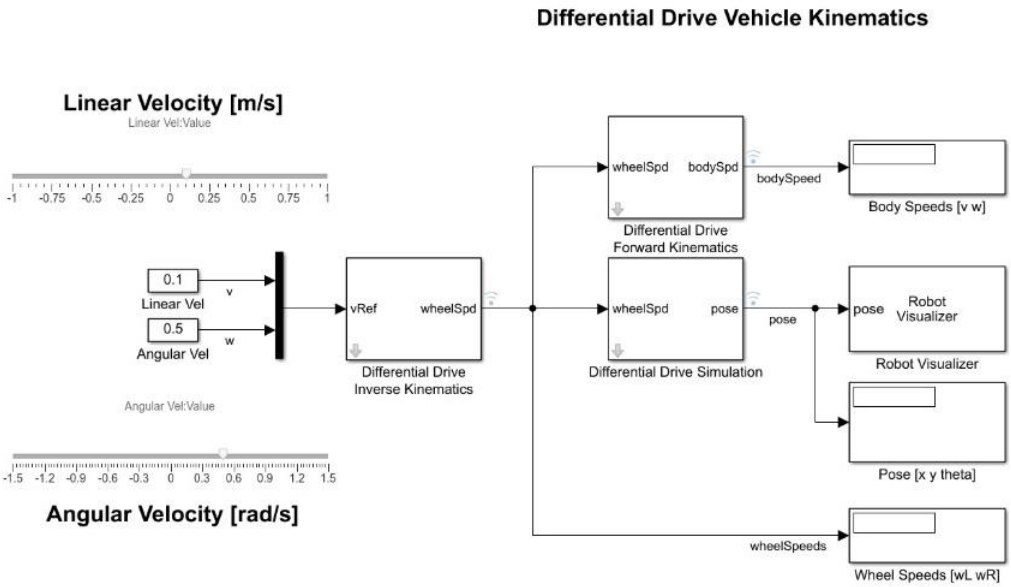


FIGURE 37

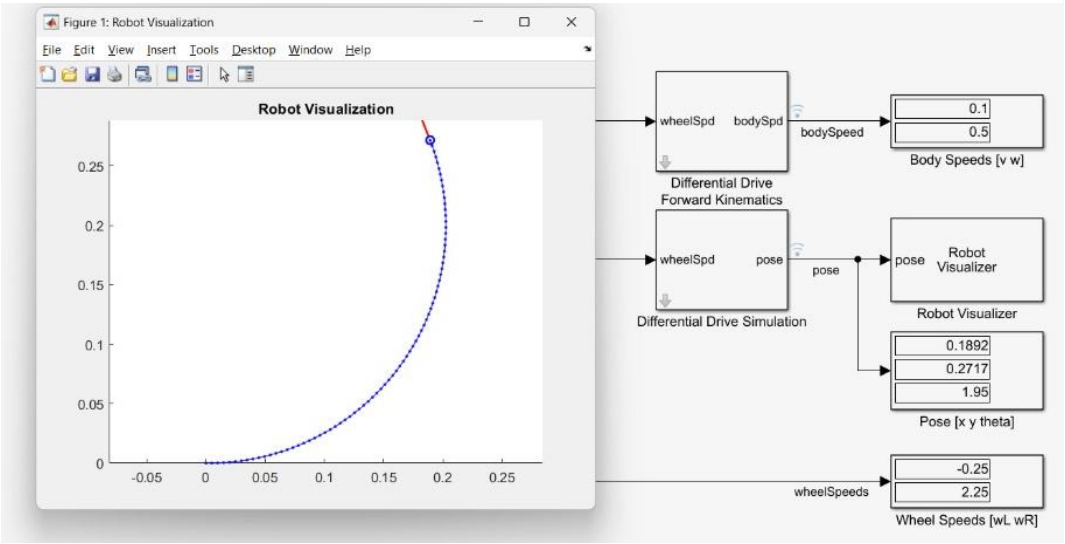


FIGURE 38

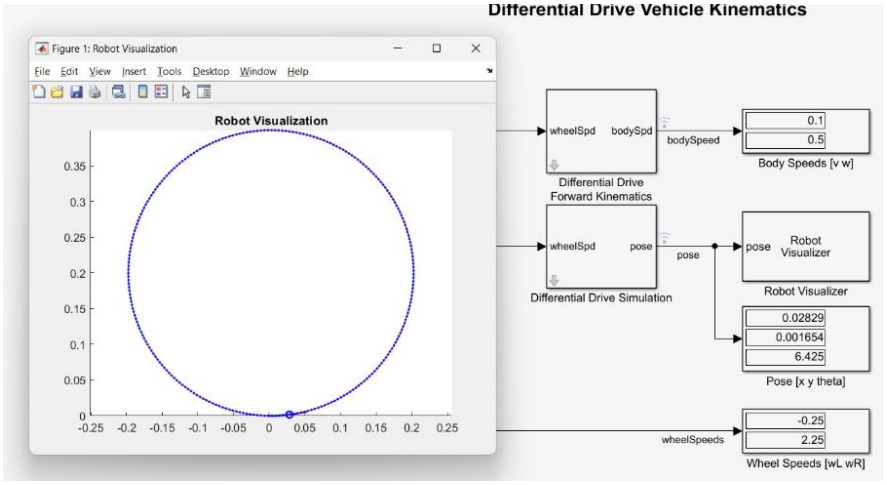


FIGURE 39

The MATLAB code simulates the motion of a wheeled mobile robot, representing the robot as a cuboid body with two cylindrical wheels. The code calculates the robot's dynamics, such as its position, velocity, and acceleration over time, and also computes the errors in these values for each wheel. Additionally, the code visualizes the robot's path in the X-Y plane.

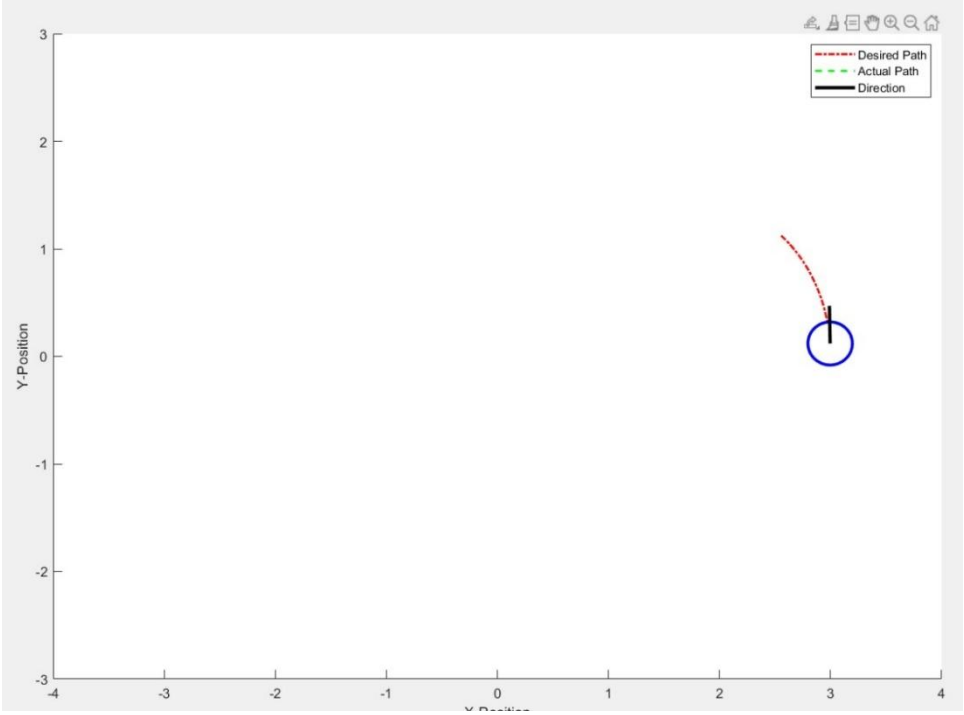


FIGURE 40

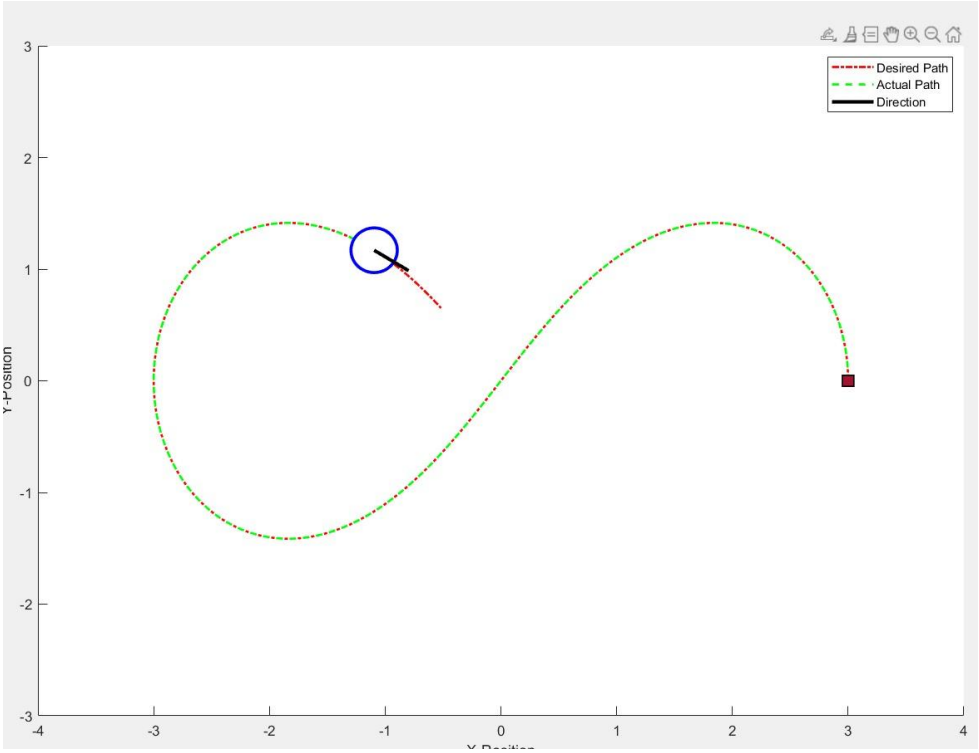


FIGURE 41

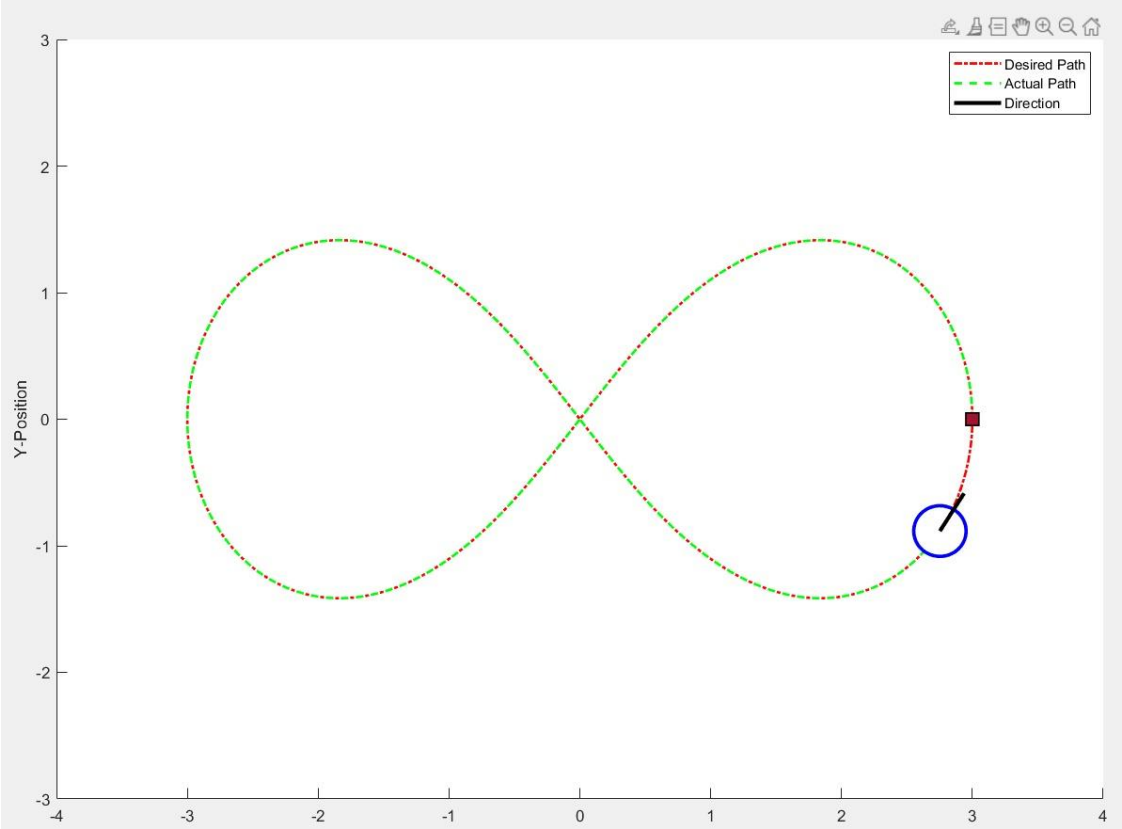


FIGURE 42

This code simulate the tracked wheel mobile robot with time t (updated coordinates only every t).

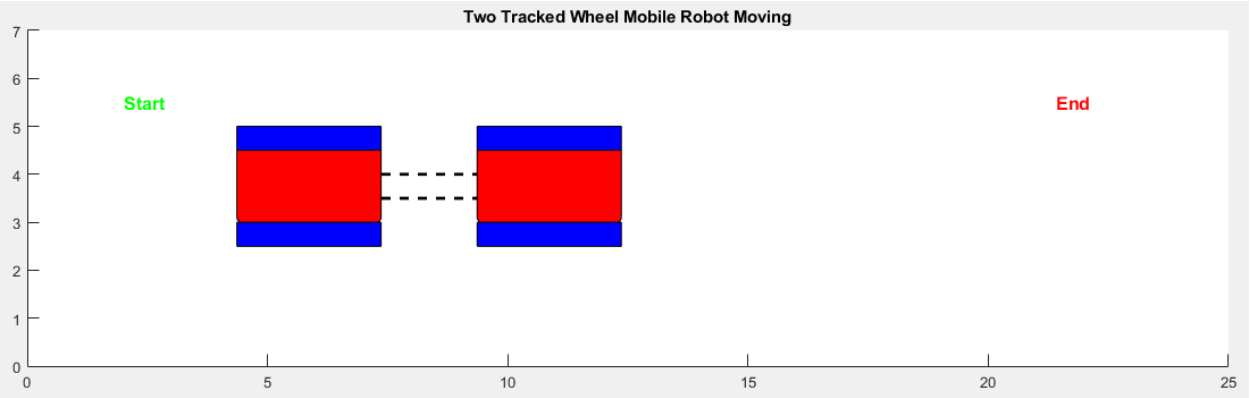


FIGURE 43 TWO TRACKED WHEEL MOBILE ROBOT MOVING

3.9 Trajectory tracking controller design

Control of mobile robots is a crucial area in robotics and engineering, as it involves the design and development of systems capable of effectively and accurately navigating robots in various environments, including those that may be complex or unknown. Mobile robots, such as wheeled or legged robots, are widely used in industrial applications, logistics, military systems, healthcare, and even in space missions.

Mobile robot control presents several technical challenges, including accurately determining the robot's position, directing it toward a specific target, avoiding obstacles, and maintaining stability during motion. To achieve this, a variety of control techniques are used, such as traditional control methods like Proportional-Integral-Derivative (PID), Model Predictive Control (MPC), and artificial intelligence techniques like neural networks and genetic algorithms.

Control systems for mobile robots are essential to ensure a balanced performance between efficiency, speed, and accuracy, especially in environments requiring quick responses and dynamic interaction with moving or changing elements. Therefore, designing an effective control system requires a deep understanding of the robot's mathematical models, in addition to developing advanced software algorithms that enable it.

TYPES OF CONTROL:

- **Computed Torque Control:** Calculates the required torque for each axis based on the robot's dynamic model to ensure precise control.
- **Backstepping Control:** A technique for controlling nonlinear systems by using multiple stages to achieve stability.
- **Reactive Control:** Provides immediate response to environmental information to ensure smooth and effective movement.
- **Path Following Control:** Ensures that the robot follows a specific path accurately using advanced techniques.
- **Model Predictive Control:** Uses a mathematical model to predict and optimize control based on future predictions.
- **Neural Network-Based Control:** Utilizes neural networks to enhance control strategies based on learning from data.
- **Adaptive Control:** Adjusts control strategies based on changes in the system or environment.
- **Intelligent Control:** Involves using artificial intelligence techniques to improve performance and decision-making.
- **Integrated Control:** Combines multiple control strategies such as guidance and planning to ensure overall performance.
- **Dynamic Analysis Control:** Analyzes the robot’s dynamic behavior to improve control strategies.

We will use **1. Computed Torque Control (CTC).**

A well-known robotic controller based on the nonlinear control technique feedback linearization. In a simple manner, it "theoretically" eliminates the system non-linearities represented in the $M, C, and\ G$ matrices in the robotic equation of motion:

$$M(q)\ddot{q} + C(q,\dot{q}) + G(q) = \tau$$

As shown in the figure below, it is easy to understand that the control consists of two loops, the inner loop which is responsible for eliminating the system nonlinearities, and the outer loop which is responsible for the motion control. The inner loop is the application of the feedback linearization technique while the outer loop is just a linear control technique known as PID control.

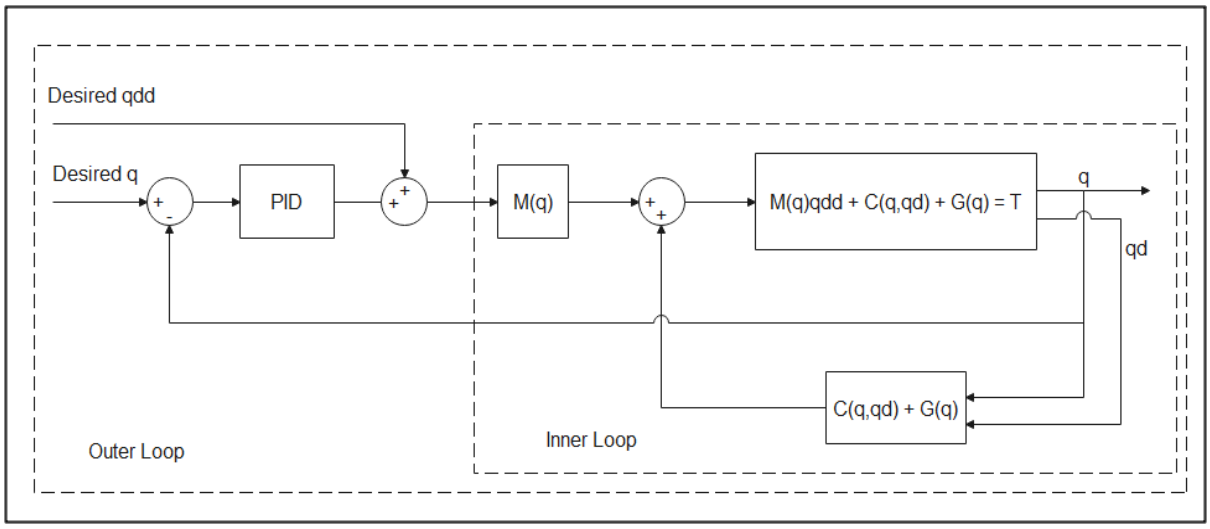


FIGURE 44 NONLINEAR CONTROL TECHNIQUE FEEDBACK LINEARIZATION

For the system here, the controlled variables are the velocities of the wheels not the angles. Consequently, the scheme above had been modulated to suit these changes as shown in the figure below.

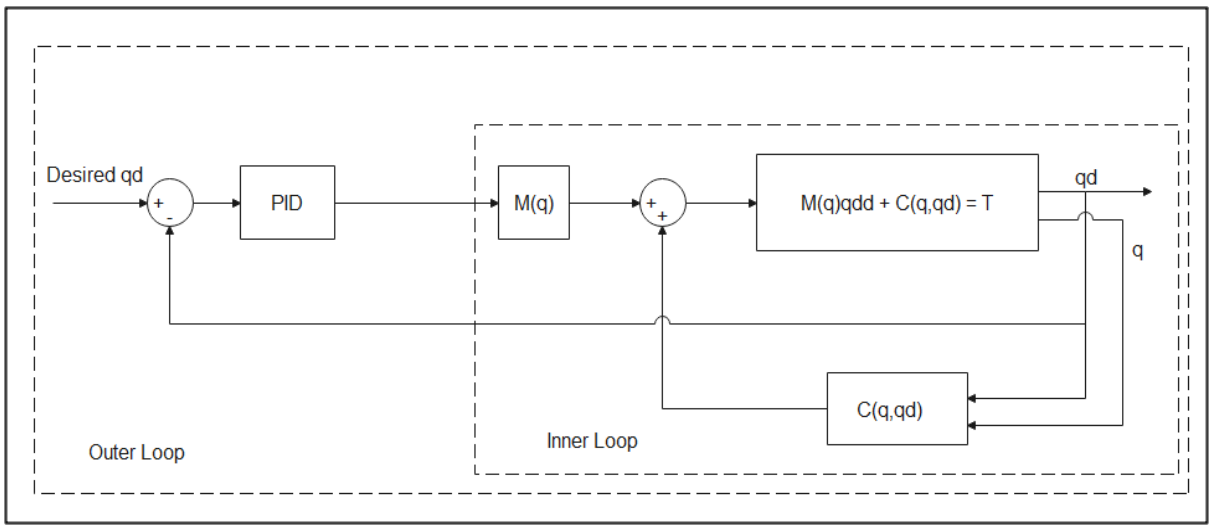


FIGURE 45 COMPUTED TORQUE CONTROL BLOCK DIAGRAM

a block diagram representing a **Computed Torque Control** system for a robot. The system relies on two main loops:

1. Outer Loop:

Here, the desired velocity (desired position or velocity) is compared to the robot's actual velocity.

A PID controller processes this error (the difference between the desired and actual velocity) and generates a control signal.

This signal is sent to the robot's dynamics to control its motion.

2. Inner Loop:

The inner part relies on the robot's dynamics model (Mass Matrix $M(q)$) with the equations that relate forces and torques to acceleration and velocity. These dynamics represent the robot's motion based on factors like mass and resistance.

$M(q)$ represents the mass matrix of the robot, while $C(q\ ,q_d)$ represents the Coriolis forces generated due to velocity.

This loop ensures that the robot moves smoothly based on the computed torque by compensating for the complex dynamics of the robot.

Objective: The goal of this diagram is to control the robot's movement and ensure its stability using a PID controller in the outer loop and adjusting the applied torque through the inner dynamic model (computed torque).

3.10 Simulink Simulation

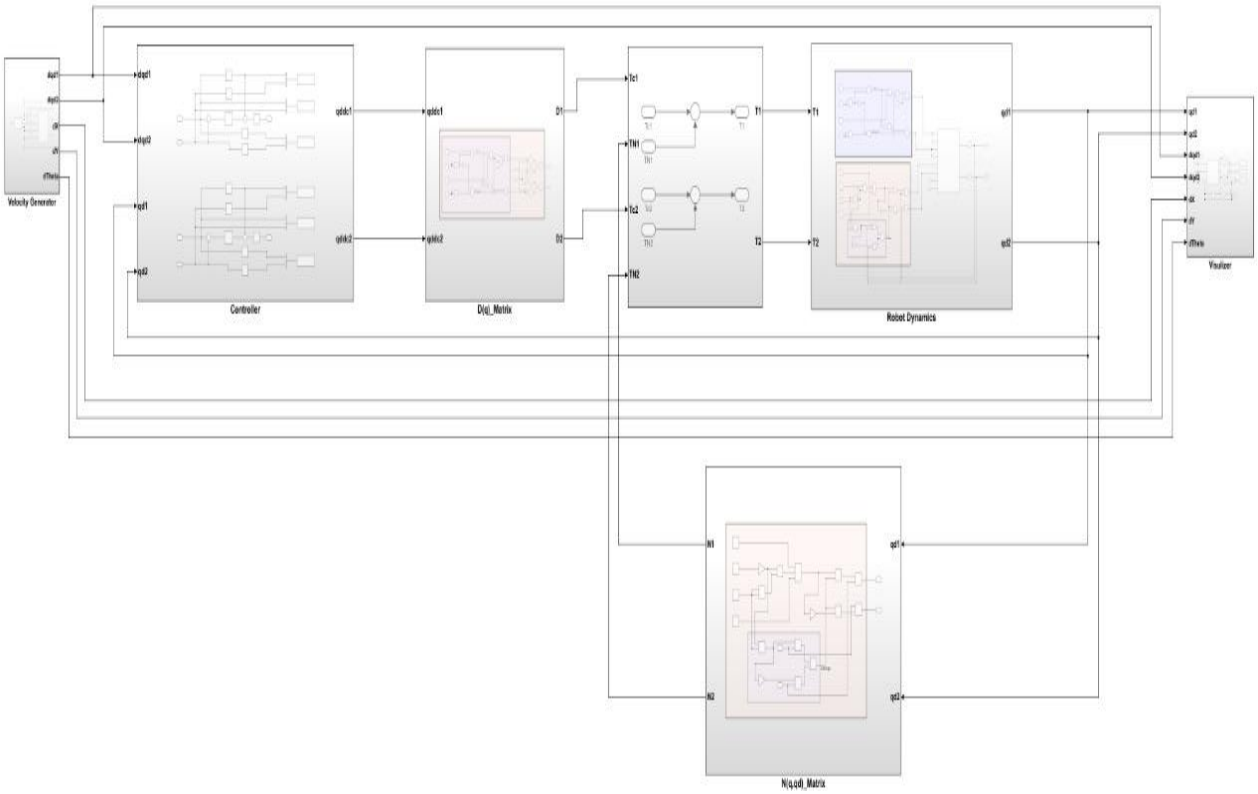


FIGURE 46 CTC SIMULINK

Detailed Simulink Block Diagram for Computed Torque Control in a Robot:

The second image shows a Simulink block diagram containing several functional blocks. The diagram appears complex but shows different stages for controlling the robot. It can be divided into the following parts:

3.10.1 Velocity Generator:

- This part generates the signals related to the desired velocity or position.
- It may depend on a reference model to move the robot according to the desired path.

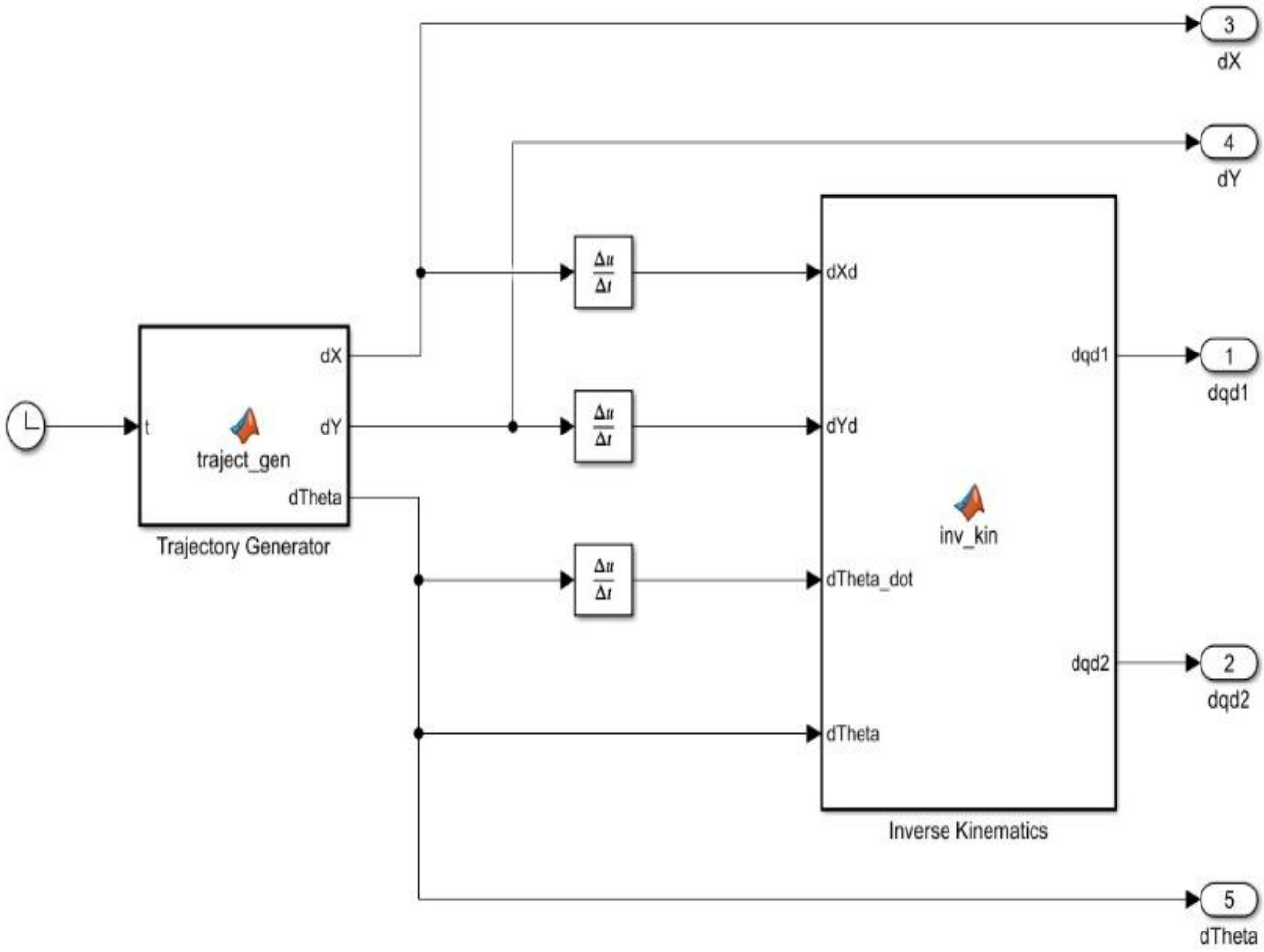


FIGURE 47 SUBSYSTEM OF VELOCITY GENERATOR

3.10.2 Controller:

This part contains a control unit (likely a PID controller) that takes the error between the actual and desired velocity or position and generates control signals.

These control signals are sent to the next blocks for execution in the robot's dynamics.

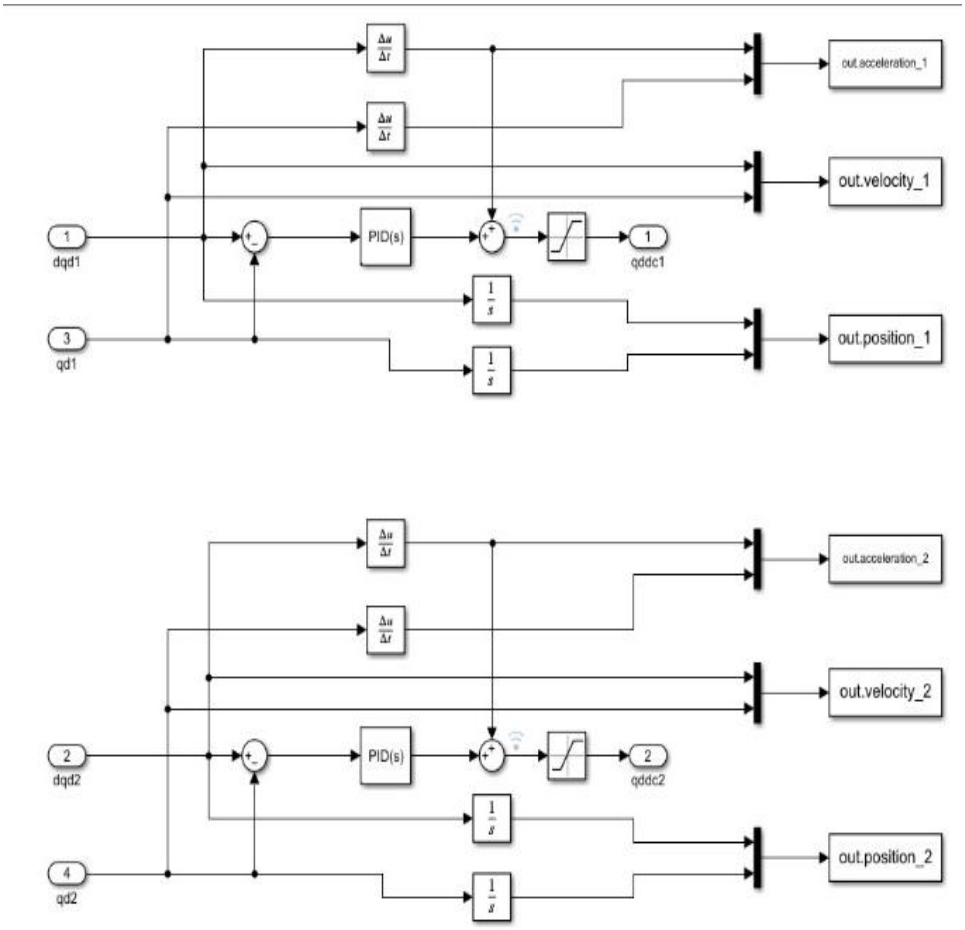


FIGURE 48 SUBSYSTEM OF PID CONTROLLER

3.10.3 $D(q)$ Block:

- This block represents the mass matrix $M(q)$ or the force matrix used to compute the required torque based on the robot's dynamics.
- It controls the relationship between the robot’s acceleration and the required torque.

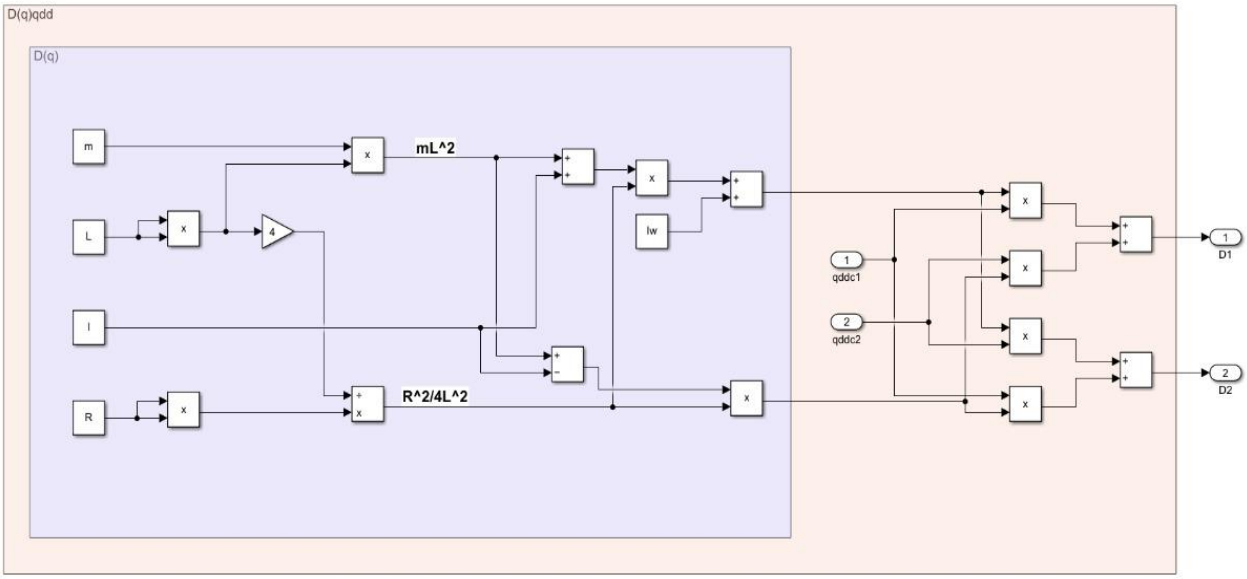


FIGURE 49 SUBSYSTEM OF INERTIA MATRIX

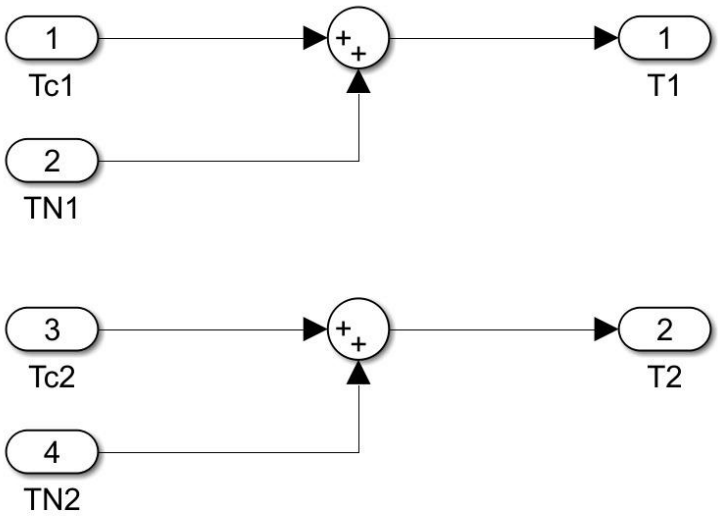


FIGURE 50 SUBSYSTEM OF SUMMING POINT

3.10.4 Robot Dynamics:

- This block represents the full dynamic model of the robot (the kinematic equations governing the robot's actual movement).
- It takes the torque output from the previous blocks and converts it into the actual motion of the robot.

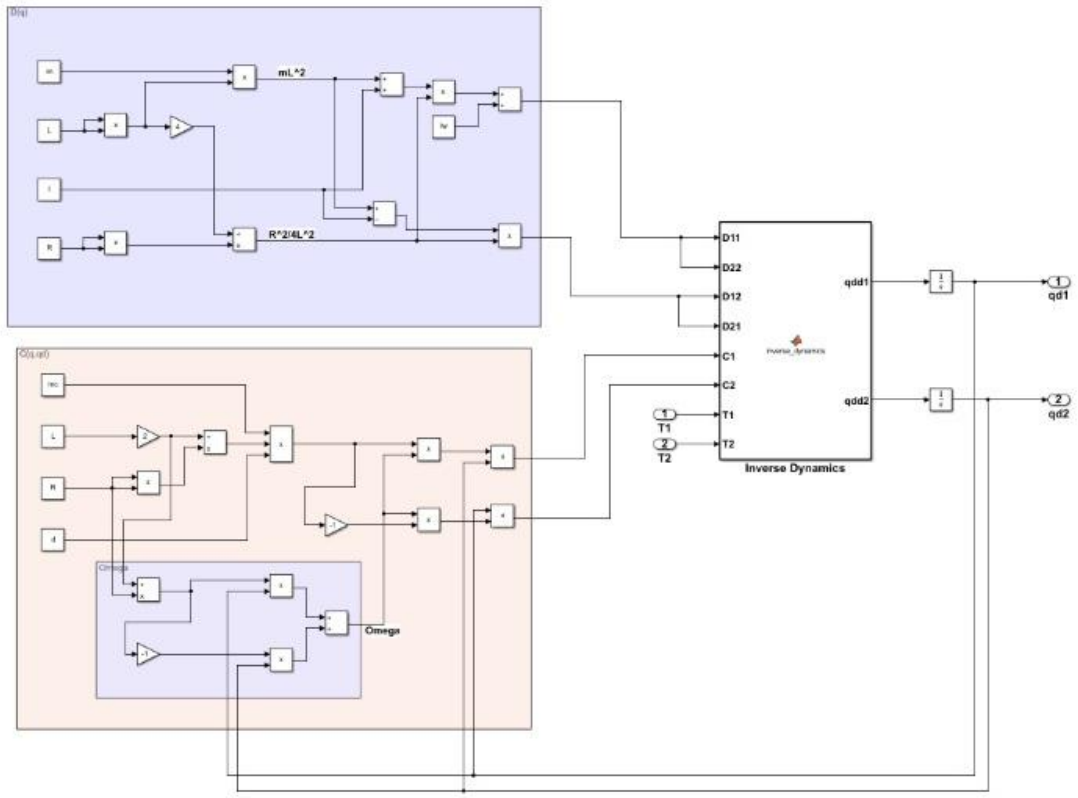


FIGURE 51 SUBSYSTEM OF ROBOT DYNAMICS

3.10.5 $N(q, q_d)$ Block:

- This block represents the nonlinear forces (like Coriolis forces) that affect the robot's motion due to velocity.
- Its function is to compensate for nonlinear factors to maintain the desired motion.

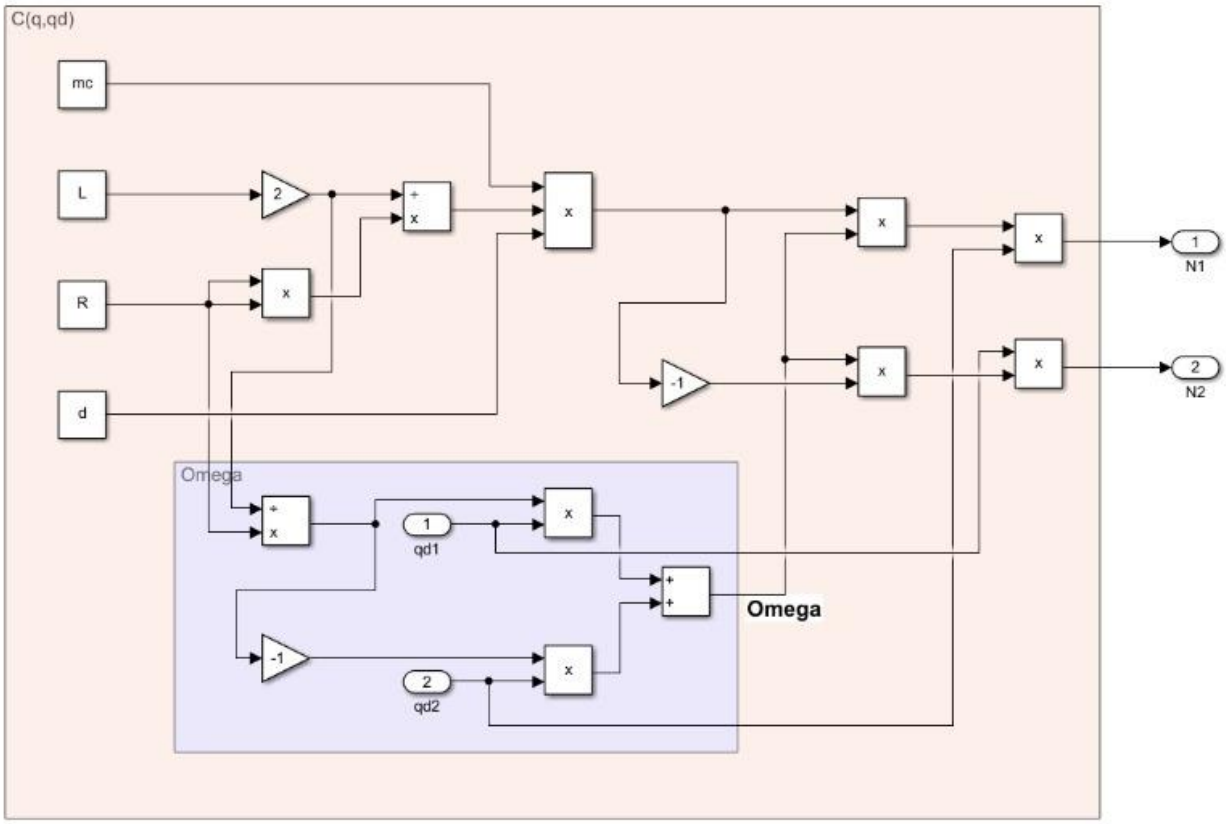


FIGURE 52 SUBSYSTEM OF CORIOLIS MATRIX

3.10.6 Visualizer:

In the end, the visualizer block shows the actual path and motion of the robot based on the computed values.

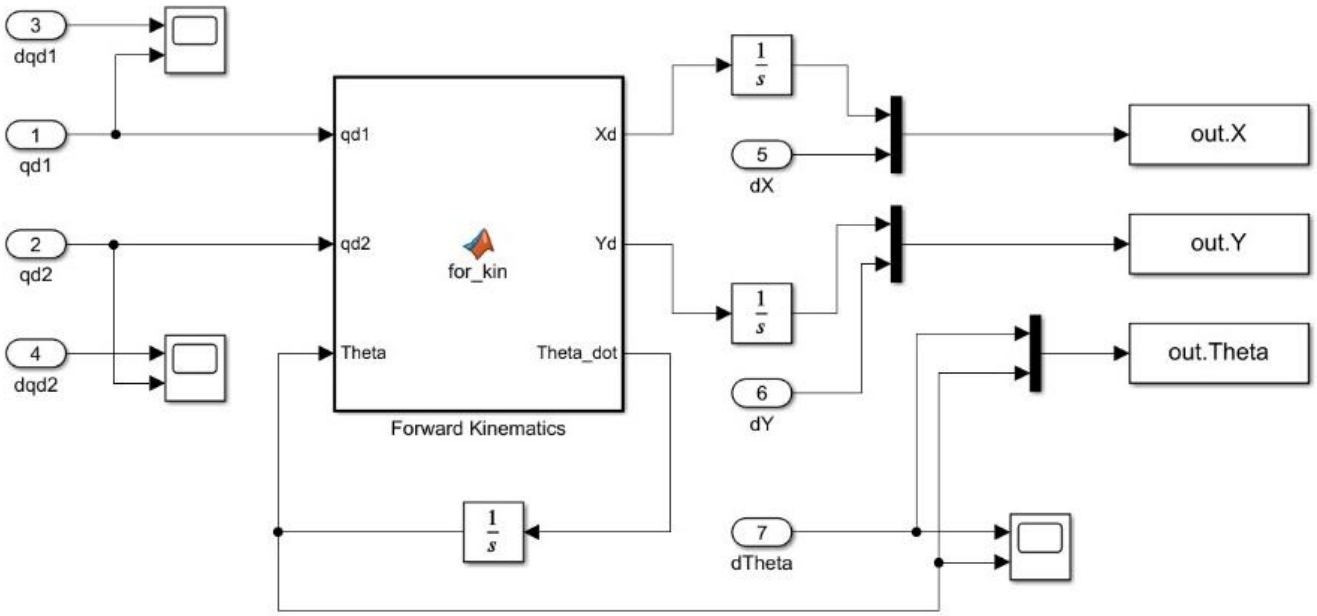


FIGURE 53 SUBSYSTEM OF VISUALIZER

The result of the simulation shows that we started from (0,0) and ended to the desired positions:

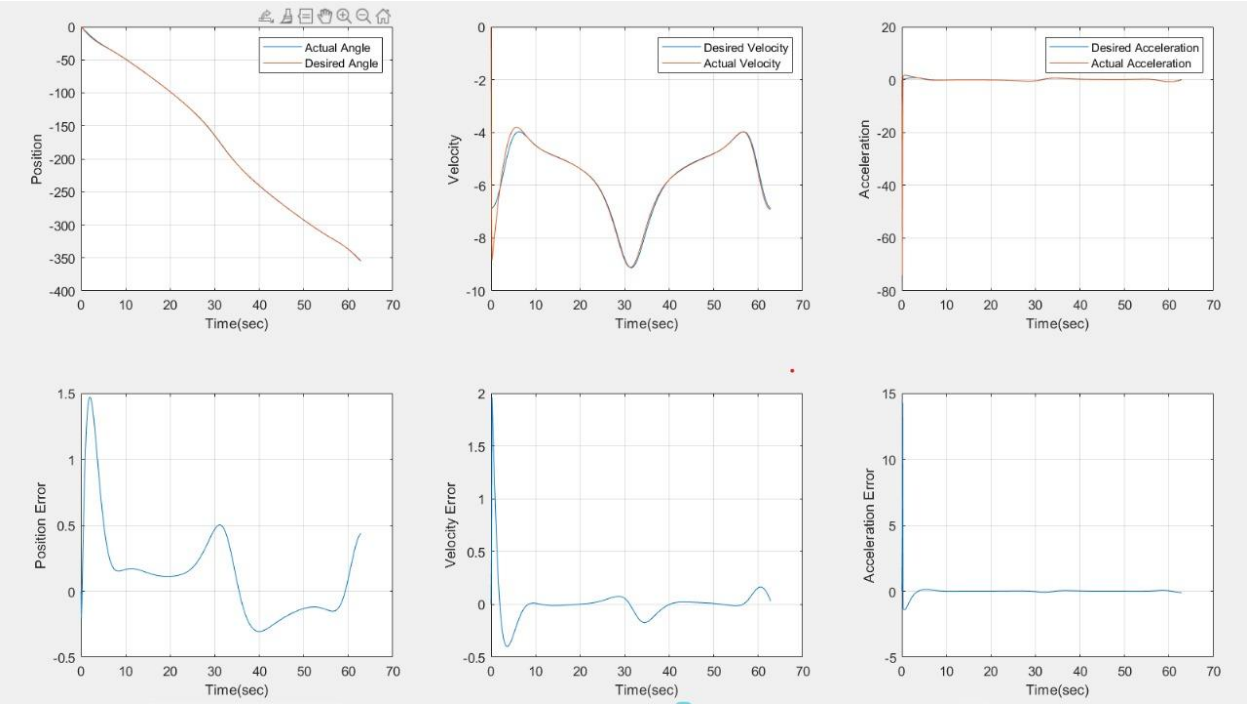


FIGURE 54 RIGHT WHEEL

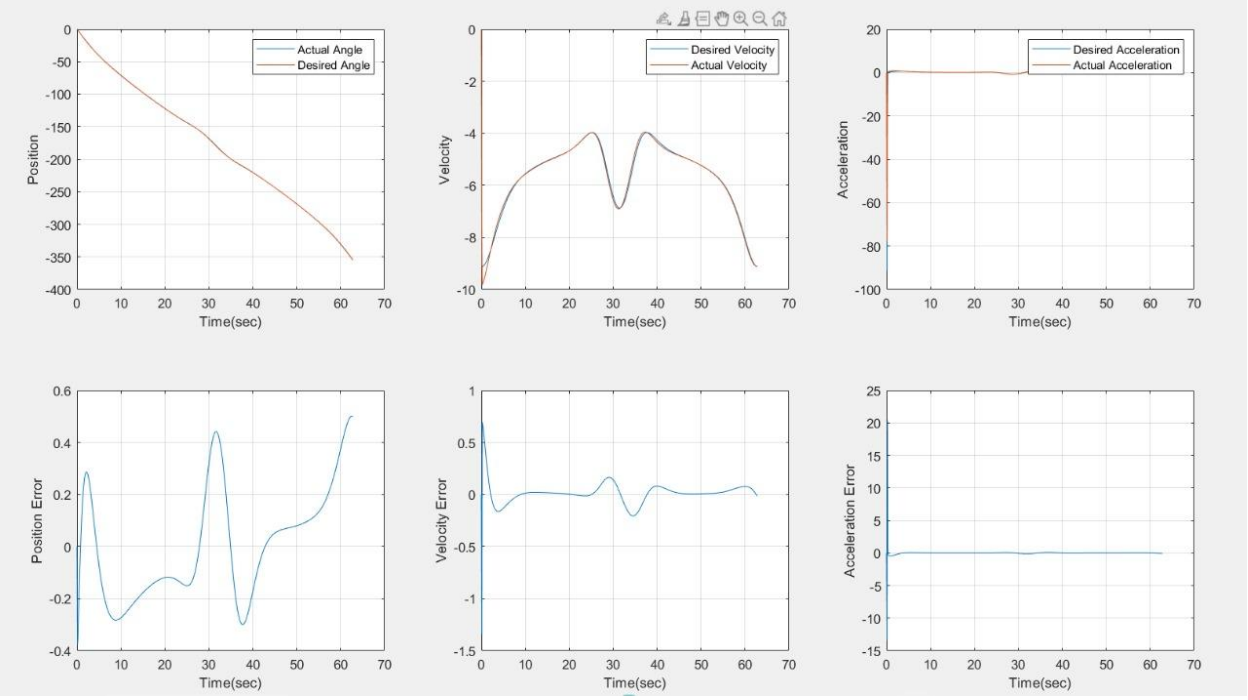


FIGURE 55 LEFT WHEEL

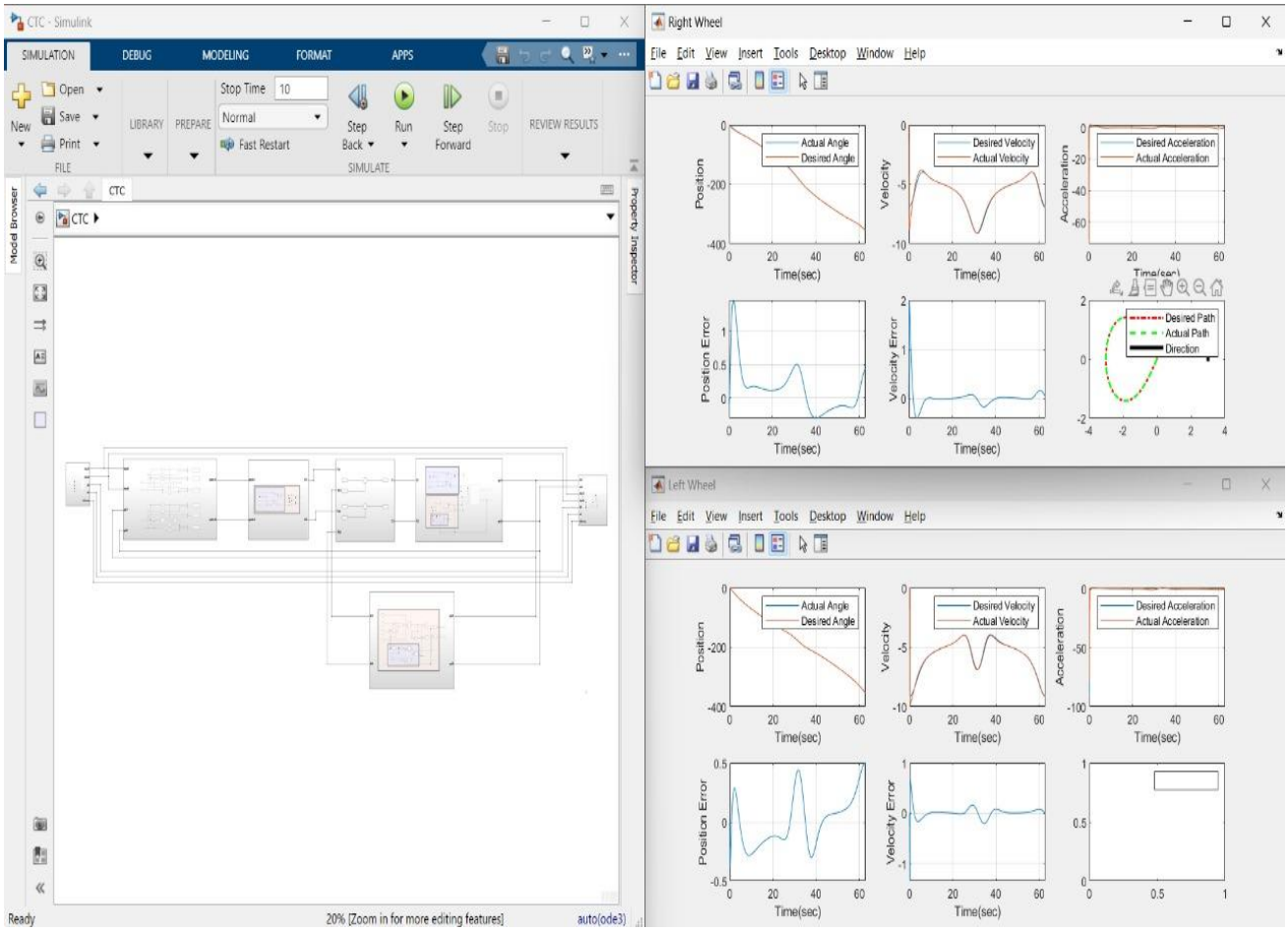
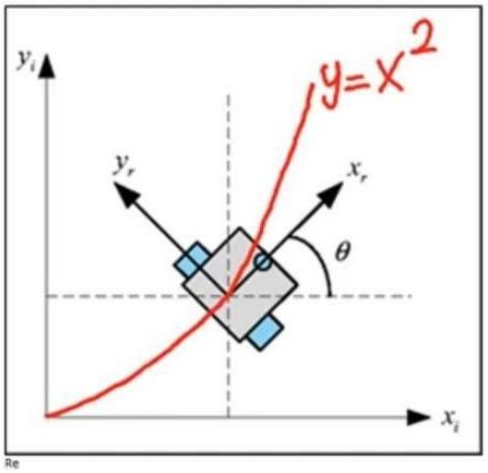


FIGURE 56 SIMULINK OF CTC AND RIGHT & LEFT ROBOT WHEEL TRAJECTORY IN XY PLANE

Control of the tracked wheel mobile robot for trajectory traversal.

Here we will discuss the control to these non-linear governing equations, so that the robot can follow the desired trajectory at a desired speed.



The equations of motion for the tracked wheel mobile robot from the previous section was

$$M\ddot{q} + B + C^T\lambda = \tau \qquad where\ \lambda = -(CM^{-1}C^T)^{-1}\left(CM^{-1}(\tau - B) + \dot{C}\dot{q}\right)$$

Where in the equation above:

$M\ddot{q}$ is the term involving acceleration, B is the term involves if you have centripetal or Coriolis accelerations, τ is the input of the system and $C^T\lambda$ is the term that results from the constraints of the robot.

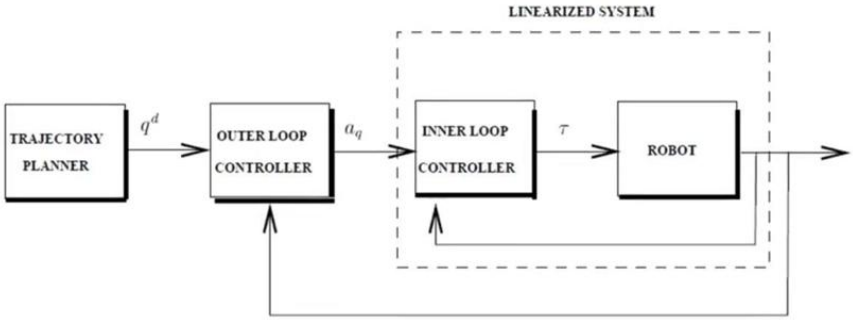
λ is the fractional force of each wheel.

our goal here is to determine how much torque is needed from the motors to follow a desired trajectory while maintaining a constant speed.

To achieved that we will use one of the best for motion control which is, Multivariate Centralized Robot Control with Feedback Linearization.

In the controller, we have two layers of control (inner loop and outer loop).

Where the inner loop creates the torque based on the PID controller, and the outer loop control will generate the desired acceleration.



In the equation of motion, we have for the robot, the left-hand side has a term that's a function of the right-hand term side (everything should be independent).

It's apparent in the equation of λ, which is a left-hand side term, depends on the right-hand side τ.

So, to use this strategy of control, we have to rewrite the equation of motion. We will combine them into one equation where the left-hand side has no τ in it.

$$M\ddot{q} + B - C^T(CM^{-1}C^T)^{-1}(CM^{-1}(\tau - B) + C\dot{q}) = \tau$$

So we will move all τ to one side of the equation. For that:

$$M\ddot{q} + B - C^T(CM^{-1}C^T)^{-1}C\dot{q} + C^T(CM^{-1}C^T)^{-1}CM^{-1}B = \tau(C^T(CM^{-1}C^T)^{-1}CM^{-1} + I)$$

Now we can apply the inner and outer loop control on the equation.

Inner Control Loop:

If we look at the feedback linearization, what we need to add should get away all this term:

$$B - C^T(CM^{-1}C^T)^{-1}C\dot{q} + C^T(CM^{-1}C^T)^{-1}CM^{-1}B$$

In the previous equation. This term should be included in the control law that we choose for τ in the inner control loop. But not just that, we also need to find the inverse of the term: $(C^T(CM^{-1}C^T)^{-1}CM^{-1} + I)$ on the right-hand side. So, that the right-hand side term and the inverse cancel each other and the left-hand side gets removed by its negative mirror and we are only down to the terms of the Maq which we need.

$$\tau = (C^T(CM^{-1}C^T)^{-1}CM^{-1} + I)^{-1}(Maq + B - C^T(CM^{-1}C^T)^{-1}C\dot{q} + C^T(CM^{-1}C^T)^{-1}CM^{-1}B)$$ (inner control equation)

Substitute the above equations and multiplying both side by M^{-1} :

$$\ddot{q} = aq$$

Outer Control Loop:

Now, we apply the outer control loop. Where we should acquire the value aq

So here we choose a combination of PD control and a feedforward.

$$aq = \ddot{q}_d + K_p q_e + K_d \dot{q}_e \qquad \text{(outer control equation)}$$

Where $\ddot{q}_d$ is the feedforward term (because this term doesn't depend on the feedback from the system, as it's a predefined signal).

From last two equations:

$$\ddot{q}_e + K_p q_e + K_d \dot{q}_e = 0$$

4.3 System Identification

In engineering, a transfer function (also known as system function or network function of a system, sub-system, or component is a mathematical function that theoretically models the system's output for each possible input. They are widely used in electronics and control systems.

The dimensions and units of the transfer function model the output response of the device for a range of possible inputs.

The transfer function of the motor can be obtained by system identification through MATLAB in order to use it in for control design.

System identification is a methodology for building mathematical models of dynamic systems using measurements of the input and output signals of the system.

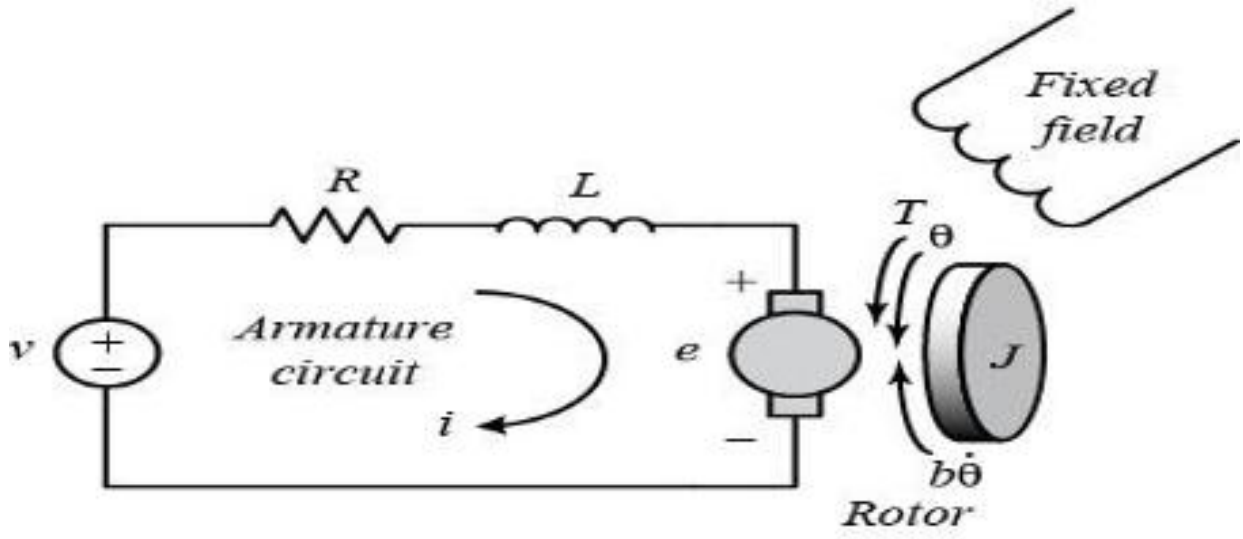


FIGURE 57 DC MOTOR CIRCUIT WITH UNKNOWN PARAMETERS (R, L, J, B, K)

The process of system identification requires that you measure the input and output signals from your system in time or frequency domain.

As mentioned before, the system identification method needs outputs and inputs of the motor to deduce its transfer function, so we will be using our readings from the encoder as the output of the motor and readings from the current sensor as the input of the motor then record them in an excel sheet (.csv) and place each of them in an array on the MATLAB workspace.

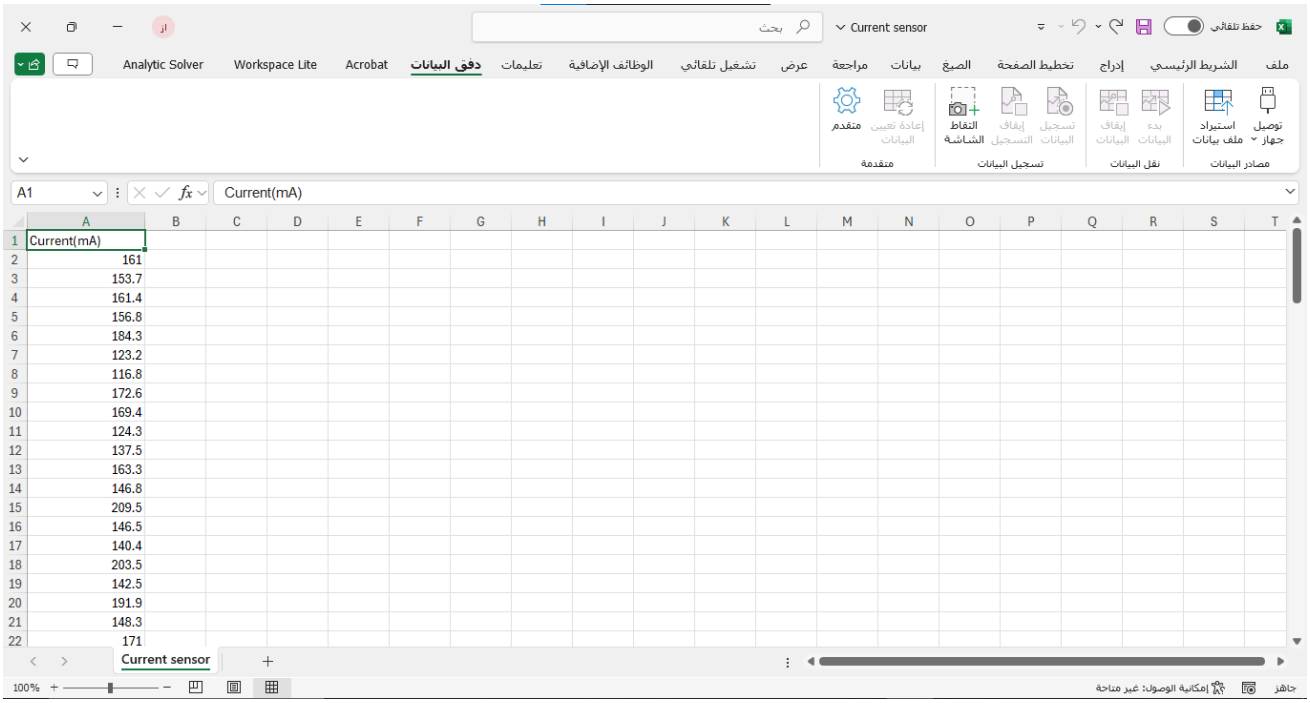


FIGURE 58 READINGS OF THE CURRENT SENSOR (MA) IN EXCEL

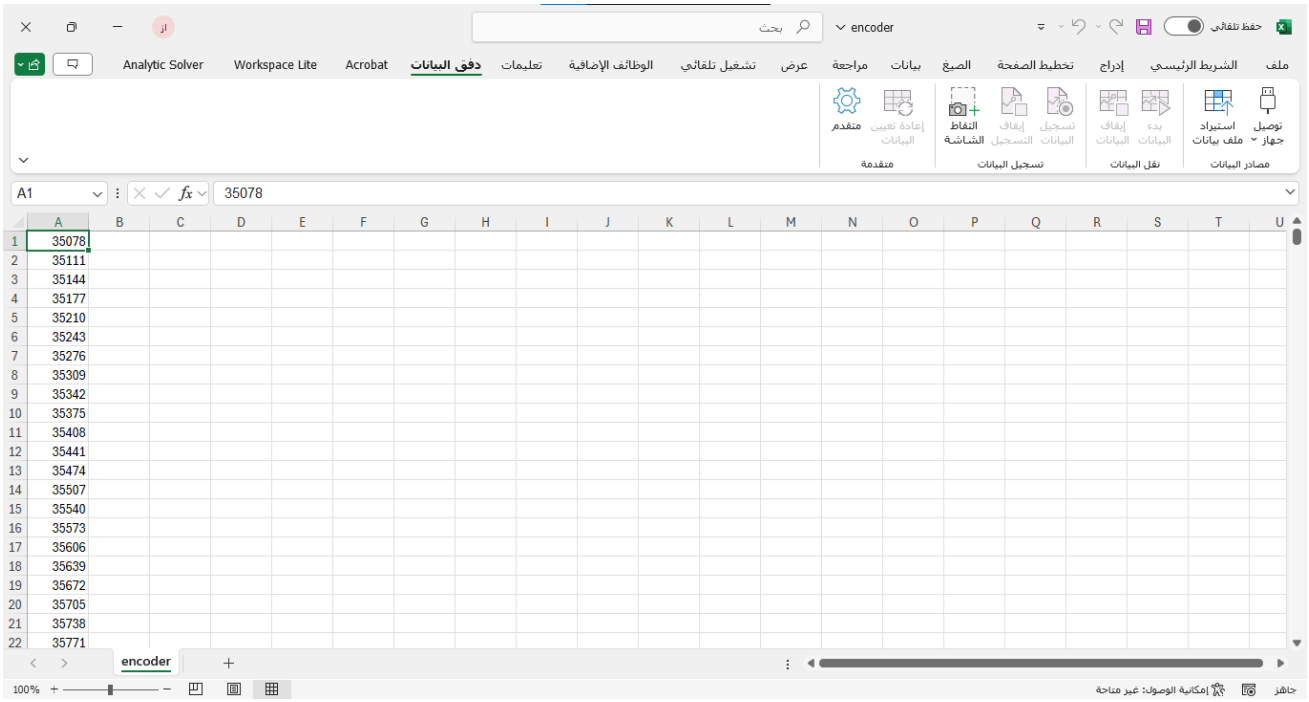


FIGURE 59 READINGS OF THE ENCODER IN EXCEL

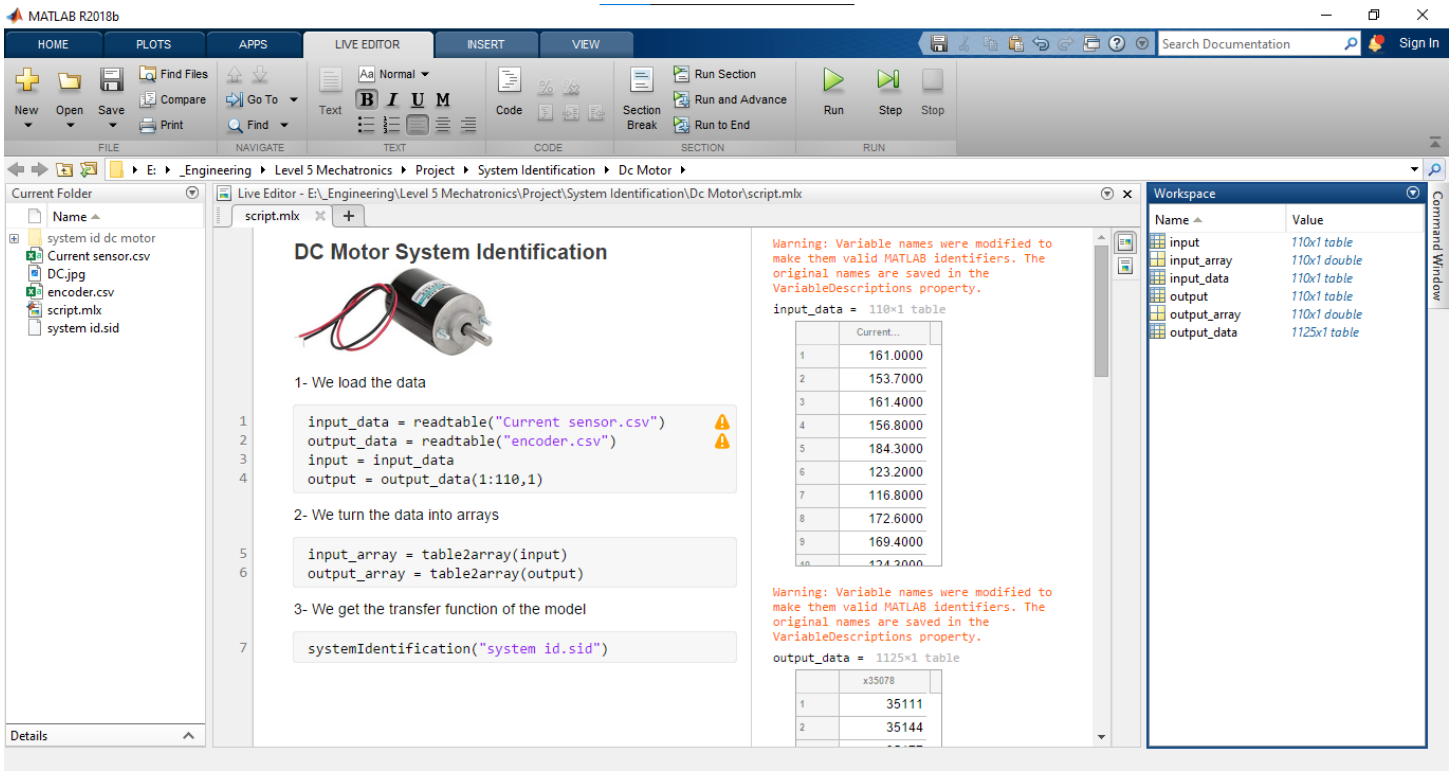


FIGURE 60 ASSIGNING INPUT AND OUTPUTS ON MATLAB

By using the command system identification, the following window appears, which need to be filled with some data which is essential to predict the transfer function which are:

1. Importing our data (input and output).
2. Inserting the number of poles of zeroes of predicted transfer function (can be changed until the estimation fit percentage is high enough).

After entering the required data, MATLAB starts using the system identification method to determine the transfer function and represent the estimation percent and the transfer function can be exported to the MATLAB workspace for further control and applications.

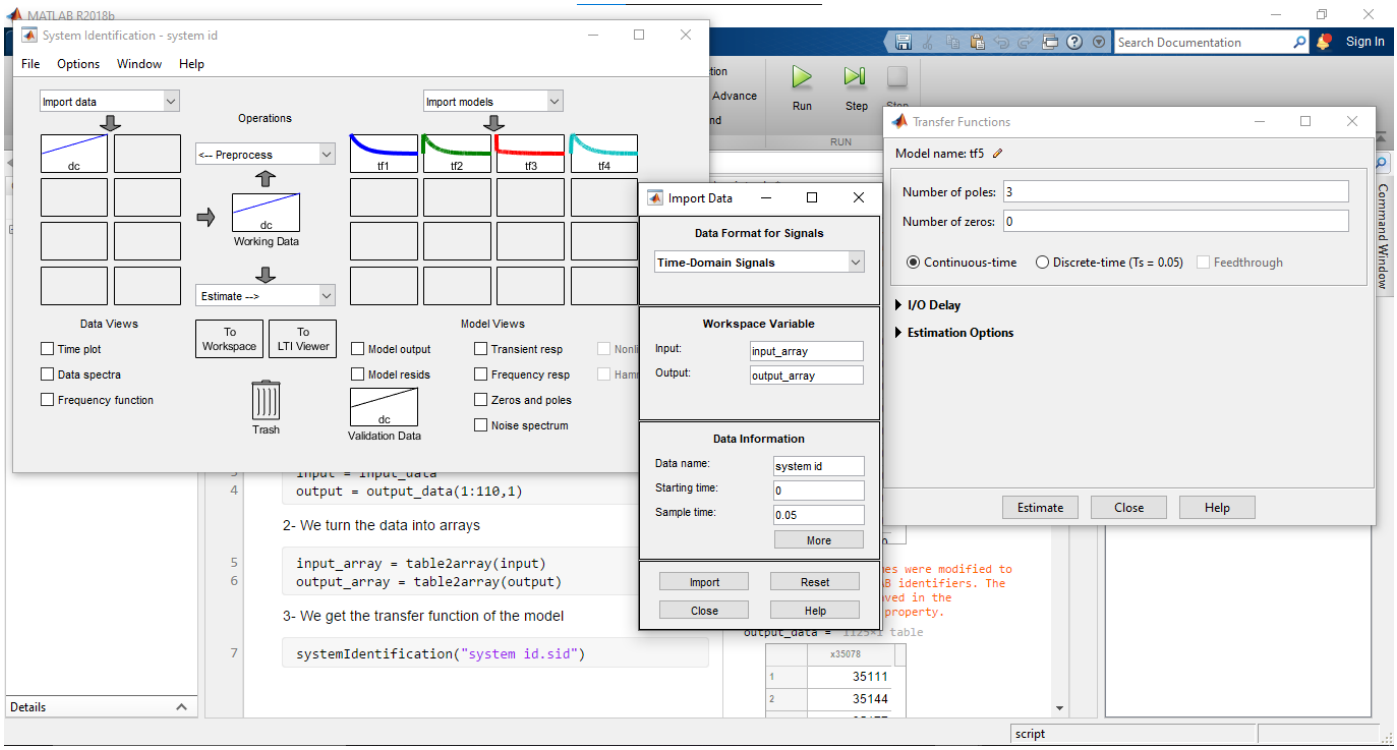


FIGURE 61 SYSTEM IDENTIFICATION WINDOW

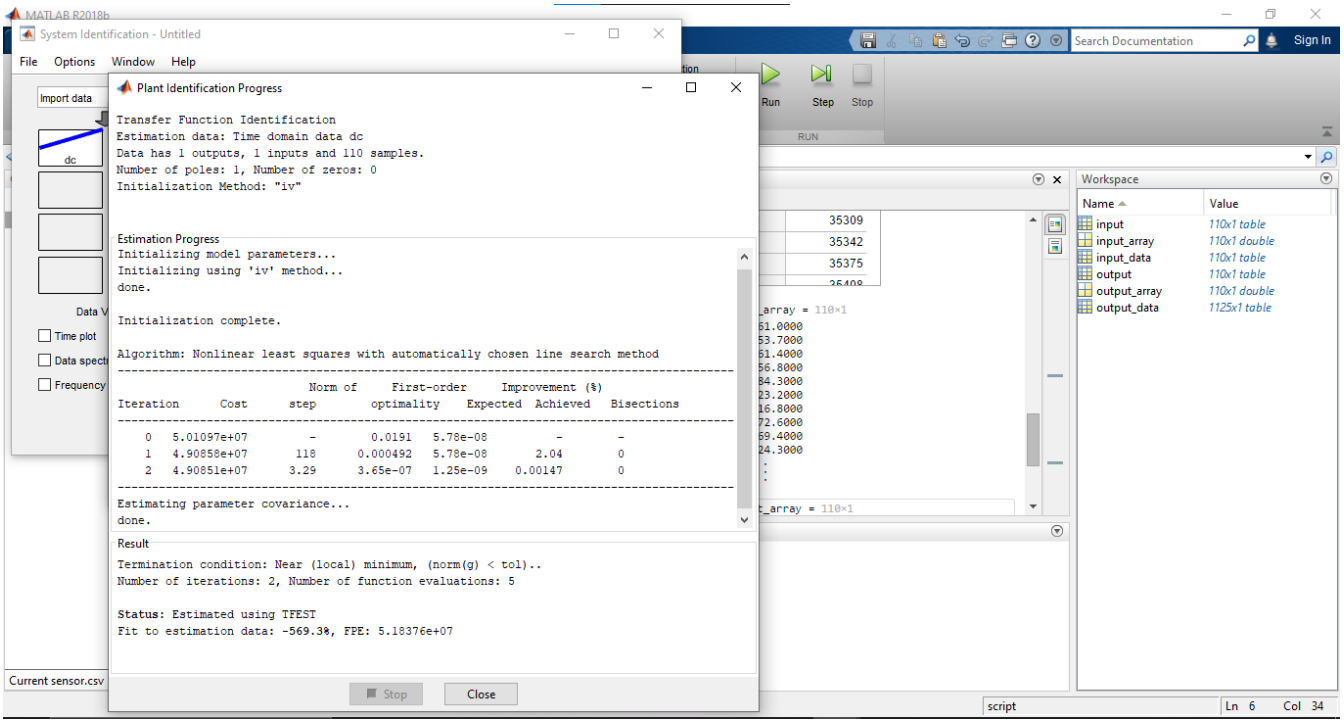


FIGURE 62 1ST ITERATION (1 POLE, NO ZEROS)

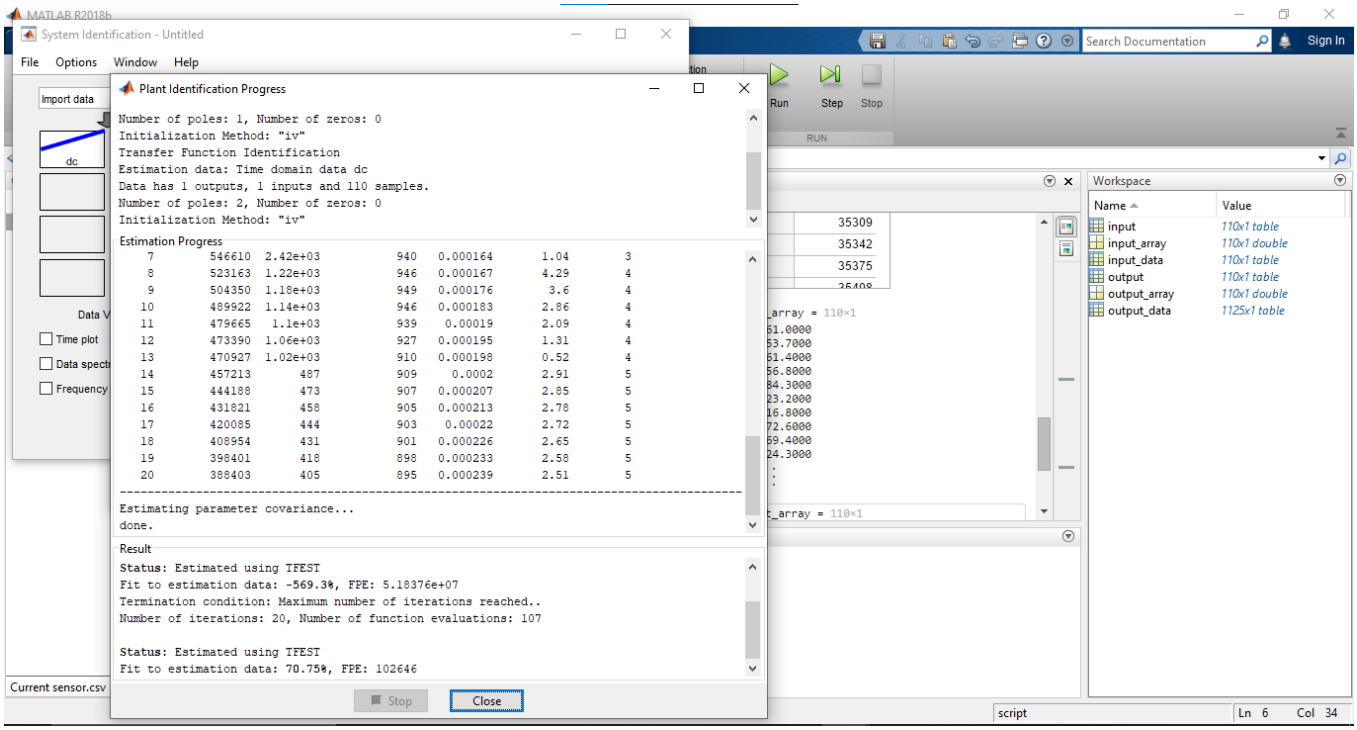


FIGURE 63 2ND ITERATION (2 POLES, NO ZEROS)

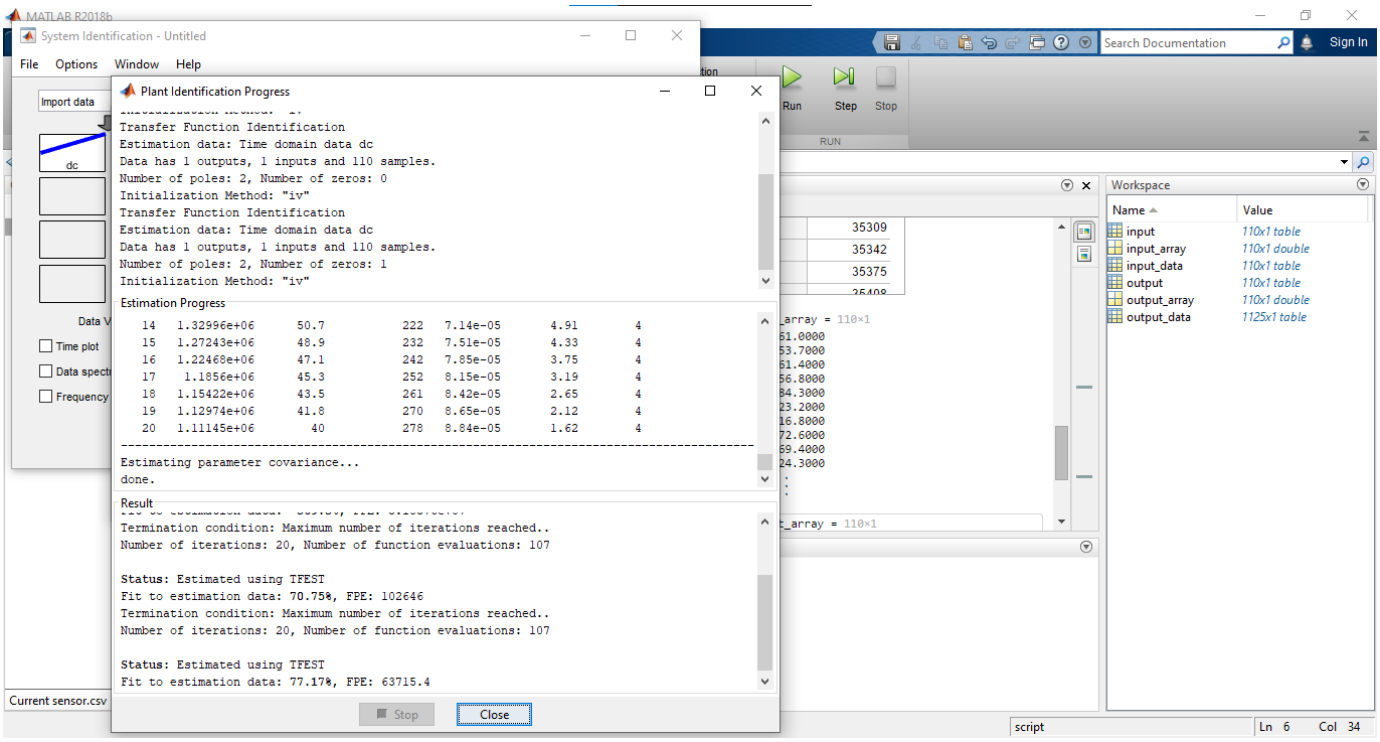


FIGURE 64 3RD ITERATION (2 POLES, 1 ZERO)

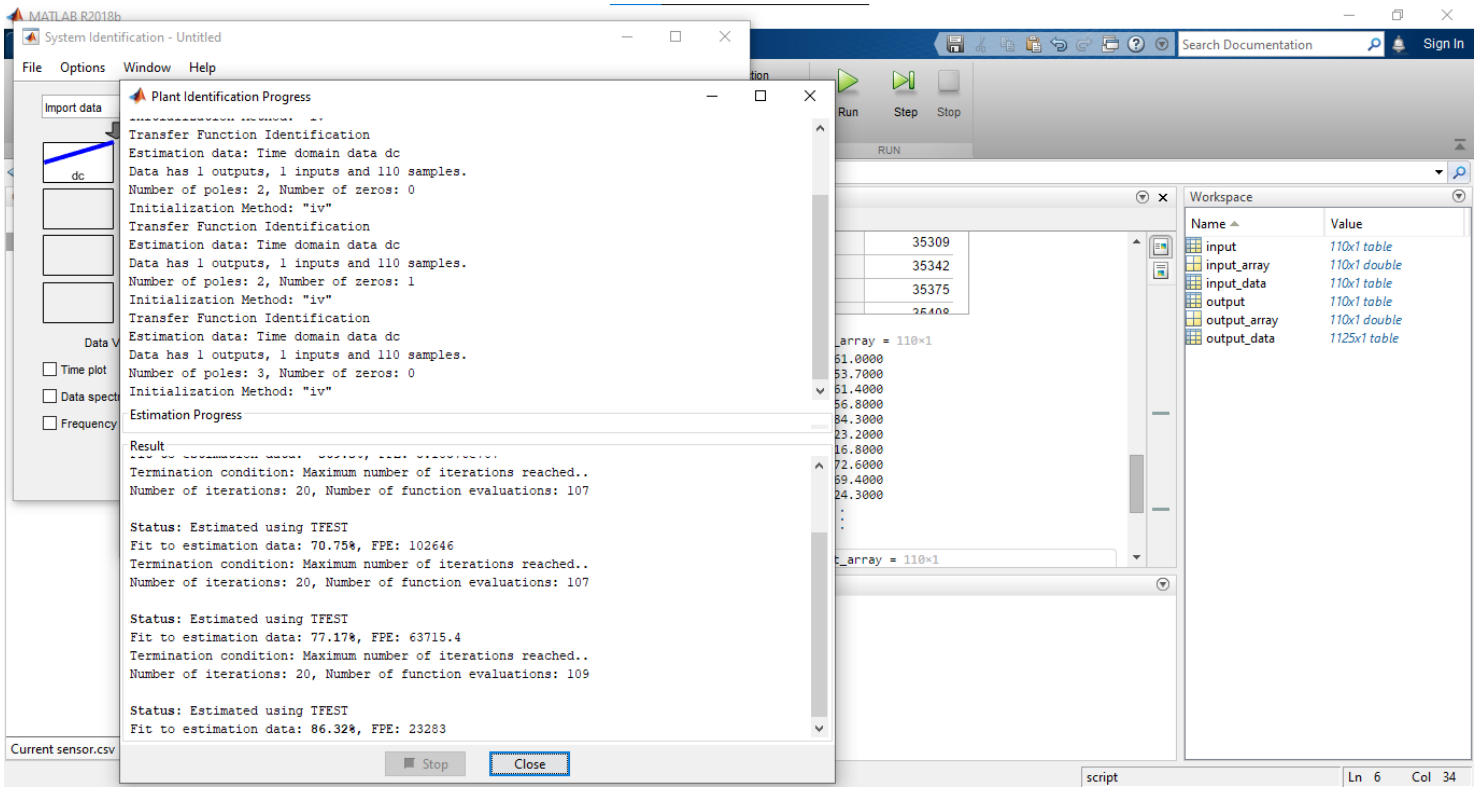


FIGURE 65 4TH ITERATION (3 POLES, NO ZEROS)

```
  From input "u1" to output "y1":
              1.649e04
  ---------------------------------
  s^3 + 36.07 s^2 + 226.7 s + 70.89
Name: tf4
Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 0
   Number of free coefficients: 4
```

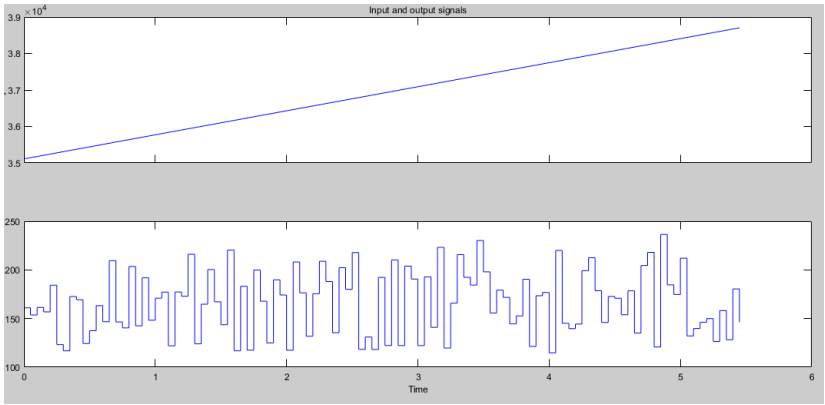
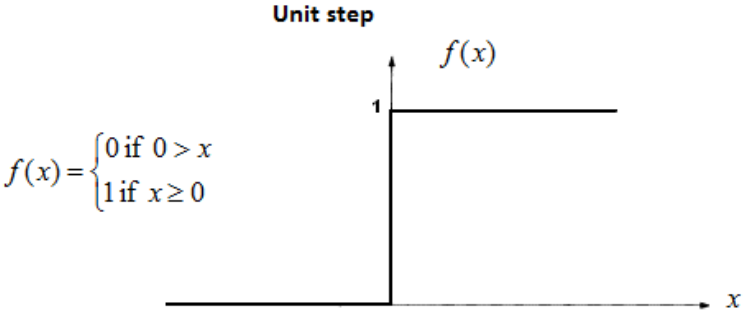


FIGURE 66 INPUT & OUTPUT SIGNALS



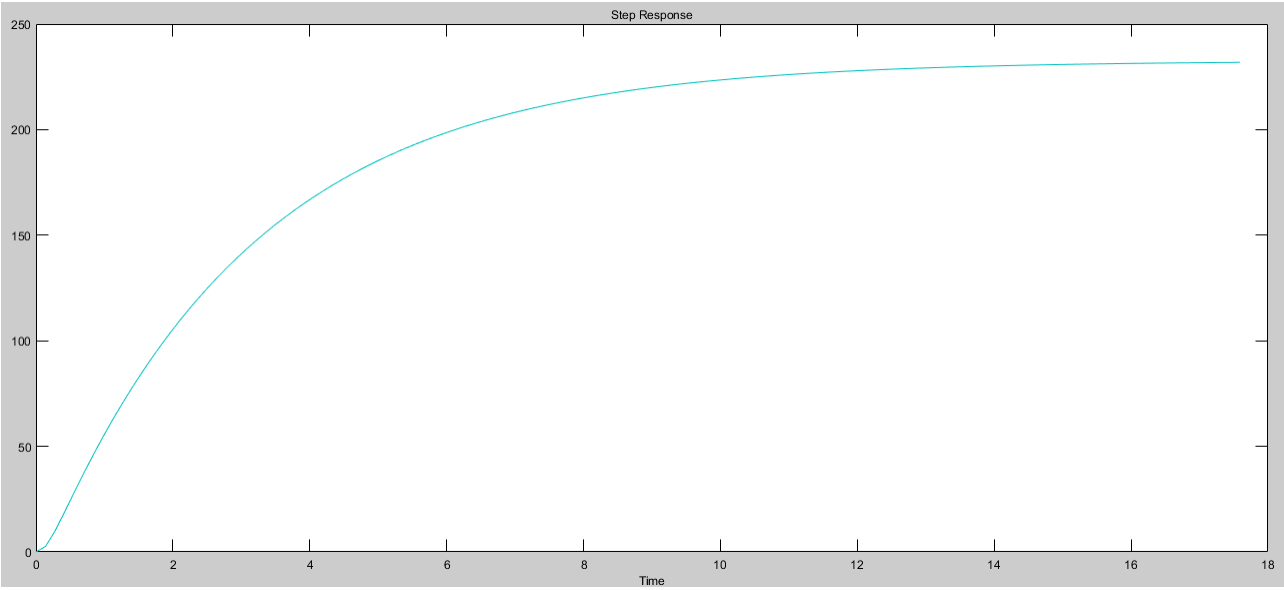


FIGURE 67 STEP RESPONSE

Step Response of The Model

The step response of the model showing how our system reacts over time when a step input (a sudden change from zero to a constant value) is applied. It shows the system's stability and transient behavior.

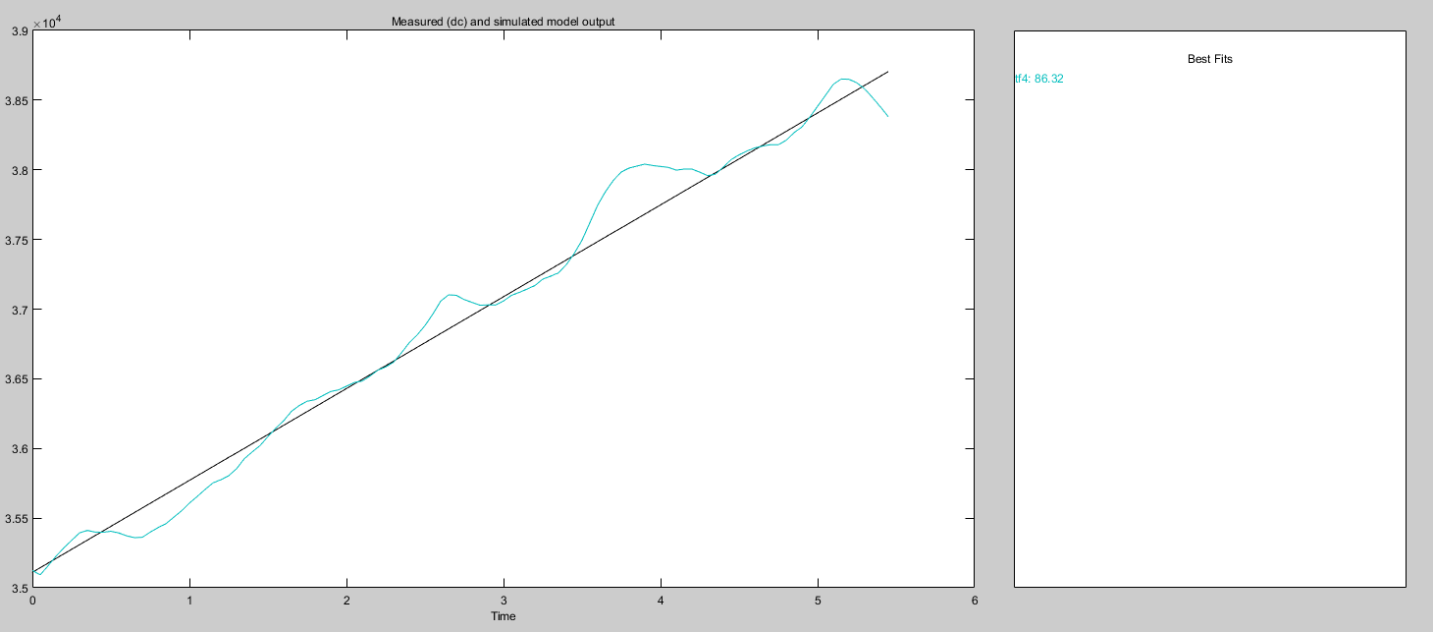


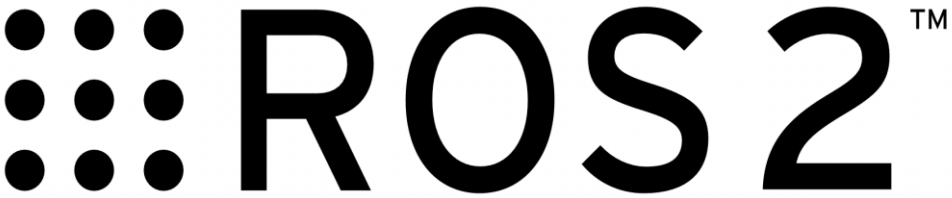
FIGURE 68 MODEL OUTPUT

The Model's Output

The output of the system identification model in MATLAB (the predicted response of the system to the input), based on the identified model parameters. It reflects how well the model captures the system dynamics.

Chapter 4
ROS AND URDF

Chapter 4 ROS2



What is ROS2?

ROS2, or Robot Operating System 2, is a flexible and open-source framework for developing robotic systems.

It provides a collection of software libraries and tools that help with various aspects of robotics development, such as communication, hardware abstraction, and visualization.

Why ROS2?

ROS2 builds upon the success of its predecessor, ROS1, and addresses some of its limitations.

It offers improved performance, scalability, and reliability, making it suitable for a wider range of applications, from small robots to large-scale systems.

ROS2 also emphasizes modularity, allowing developers to build and integrate components more easily.

Key Features of ROS2:

1. **Communication:** ROS2 enables communication between different components of a robotic system, such as sensors, actuators, and controllers, through a publish-subscribe messaging model.
2. **Middleware:** It provides a middleware layer that abstracts the underlying communication infrastructure, allowing developers to write robot applications independently of the specific hardware or network protocols.
3. **Tools and Ecosystem:** ROS2 offers a rich set of tools, libraries, and packages that facilitate robot development, simulation, visualization, and testing.

4.1 ROS2 Workspaces:

In ROS2 (Robot Operating System 2), a workspace is a directory structure used to organize and manage packages, code, and dependencies for a robotics project. It is an essential part of the ROS 2 development workflow and helps maintain a clear and consistent structure for your projects.

4.1.1 Key Features of a ROS 2 Workspace:

1. **Customization:** You can create a workspace tailored to your project's needs, adding specific packages or dependencies.
2. **Isolation:** Different workspaces can coexist on the same system, making it easy to separate and test multiple projects.
3. **Dependency Management:** Workspaces manage dependencies for building and running ROS 2 packages.
4. **Modularity:** Each package in a workspace can perform specific functions, and multiple packages can interact to form a complete robotic system.

4.1.2 Structure of a ROS 2 Workspace

A typical ROS 2 workspace contains the following directories:

1. **src:** This is where the source code for packages is stored. It is the main directory you work with when developing a new package.
2. **Build:** Created after building the workspace. Contains intermediate build files for packages.
3. **Install:** Holds the installed package binaries and other necessary files after building the workspace.
4. **Log:** Stores log files generated during builds or running nodes.

4.1.3 Steps to Create and Use a ROS 2 Workspace

1. Create a Workspace:
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/
Here, ros2_ws is the name of the workspace. Replace it with a name that suits your project.
2. Add Packages:
Clone or create packages in the src/ directory of the workspace.
3. Build the Workspace:
Use colcon, the build tool for ROS 2:
colcon build
4. Source the Workspace:
To use the built packages, source the workspace:
source install/setup.bash
5. Run Nodes or Use the Workspace:
After sourcing, you can run ROS 2 nodes or execute commands from your workspace.

- **Example Use Case**

Imagine you're working on a robot with navigation, perception, and manipulation modules. You could organize your workspace as follows:

```
ros2_ws/
└── src/
    ├── navigation_package/
    ├── perception_package/
    └── manipulation_package/
```

You can then build and test each module individually or together in the same workspace.

4.1.4 Advantages of ROS 2 Workspaces

1. Simplifies package management and builds.
2. Encourages modular design, making code reusable.
3. Makes it easier to handle large projects with multiple teams.

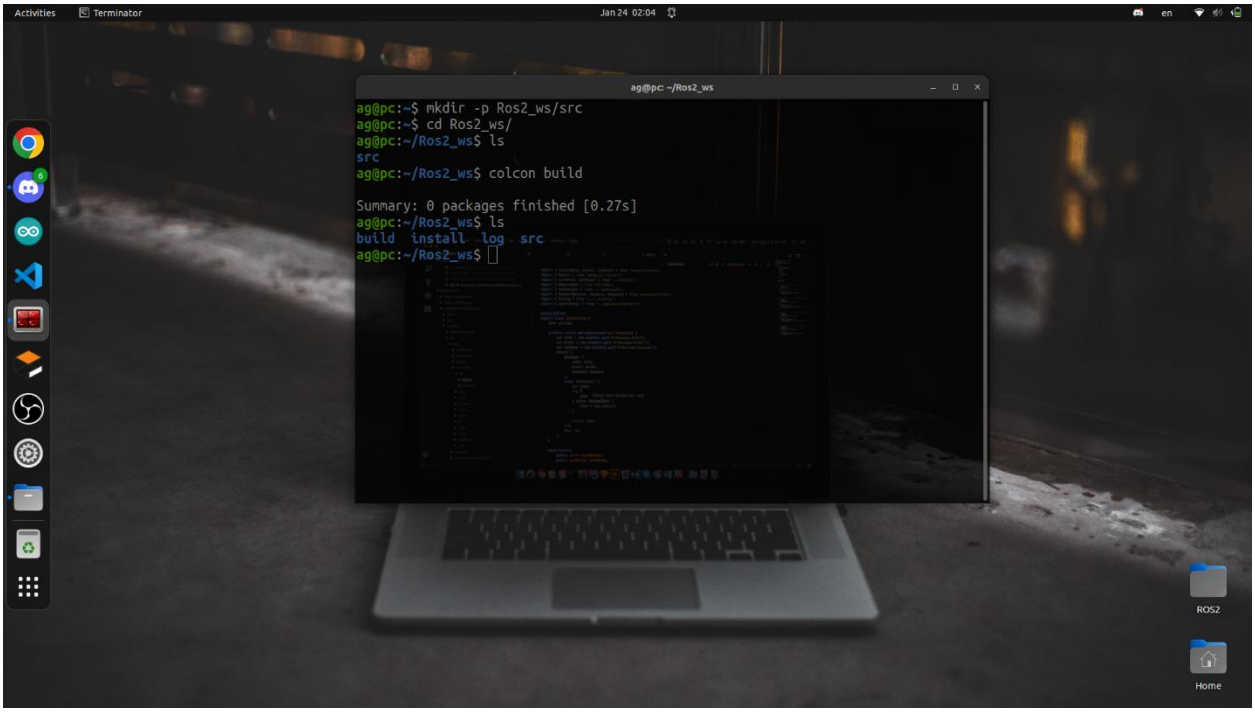


FIGURE 69 CREATE A NEW WORKSPACE

4.2 ROS2 Packages:

In ROS 2 (Robot Operating System 2), a package is the smallest unit of software organization. It contains all the files needed to create a piece of functionality, such as nodes, libraries, scripts, configuration files, and more. Packages are the building blocks of a ROS 2 system and are designed to promote modularity and reusability.

4.2.1 Key Features of a ROS 2 Package:

1. **Self-Contained:** Each package is self-contained, making it easy to share and reuse.
2. **Organized Structure:** A standard package structure ensures consistency across ROS projects.
3. **Dependencies:** A package can declare its dependencies, which are automatically handled during building.
4. **Modular:** Different packages can work together to form a complete robotic system.

4.2.2 Typical Structure of a ROS 2 Package:

A ROS 2 package follows a standard directory layout:

```
package_name/
├── CMakeLists.txt        # Build configuration using CMake
├── package.xml           # Metadata about the package (name, version, dependencies)
├── src/                  # Source code (e.g., .cpp files for nodes)
├── include/              # Header files (for libraries or shared code)
├── launch/               # Launch files to start nodes or systems
├── config/               # Configuration files (e.g., YAML)
├── msg/                  # Custom message definitions (if any)
├── srv/                  # Custom service definitions (if any)
└── scripts/              # Executable Python scripts or other scripts
```

4.2.3 Key Files in a ROS 2 Package:

1. **package.xml:**
   Contains metadata about the package (e.g., name, version, description, dependencies).
2. **CMakeLists.txt:**
   Specifies how the package should be built using CMake.

Creating a New ROS 2 Package:

To create a new package, use the ros2 pkg create command:
~ ros2 pkg create <package_name> --build-type <build_tool> --dependencies <dependencies>

- **Example**:
  ~ ros2 pkg create my_robot --build-type ament_cmake --dependencies rclcpp std_msgs
  This creates a package named my_robot with:
  - **Build tool:** ament_cmake or ament_python
  - **Dependencies:** rclcpp or rclpy and std_msgs

Using a ROS 2 Package:

1. **Build the Package:**
   Add the package to your workspace and build it using colcon:
   ~ colcon build
2. **Source the Workspace:**
   Source the workspace to use the package:
   ~ source install/setup.bash
3. **Run Nodes in the Package:**
   If your package has nodes, you can run them using 'ros2 run':
   ~ ros2 run <package_name> <node_name>

4.2.4 Benefits of ROS 2 Packages:

1. **Modular Design**: Breaks functionality into smaller, reusable components.
2. **Ease of Collaboration:** Allows teams to work on different packages simultaneously.
3. **Dependency Management:** Handles library dependencies efficiently.

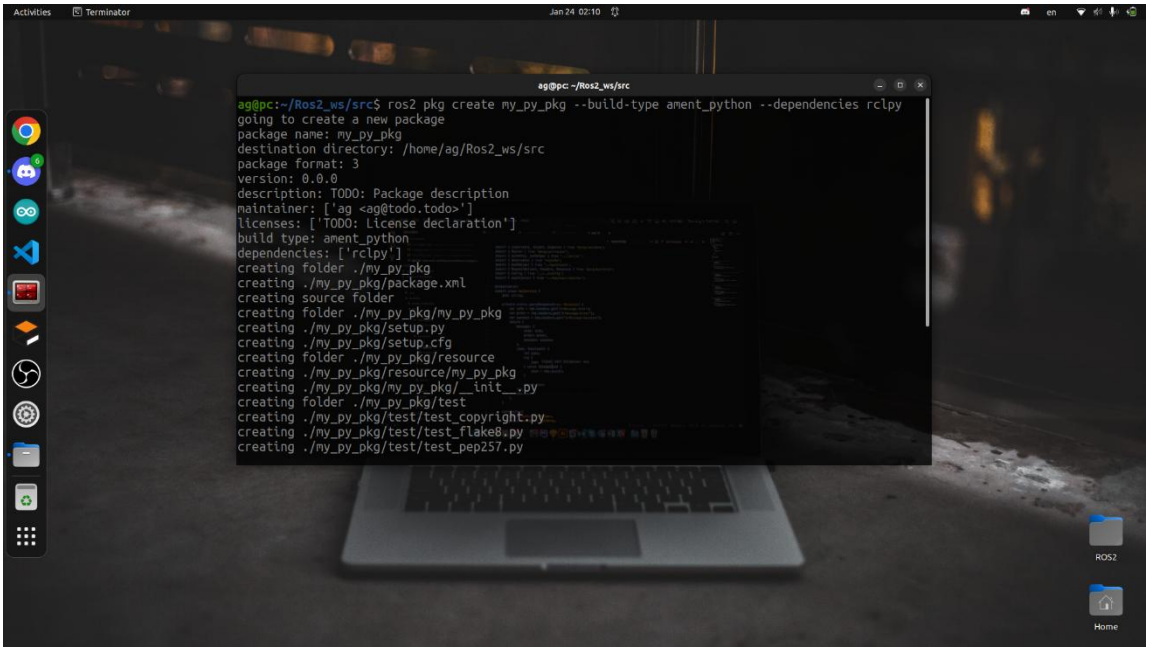


FIGURE 70 CREATE NEW PYTHON PACKAGE

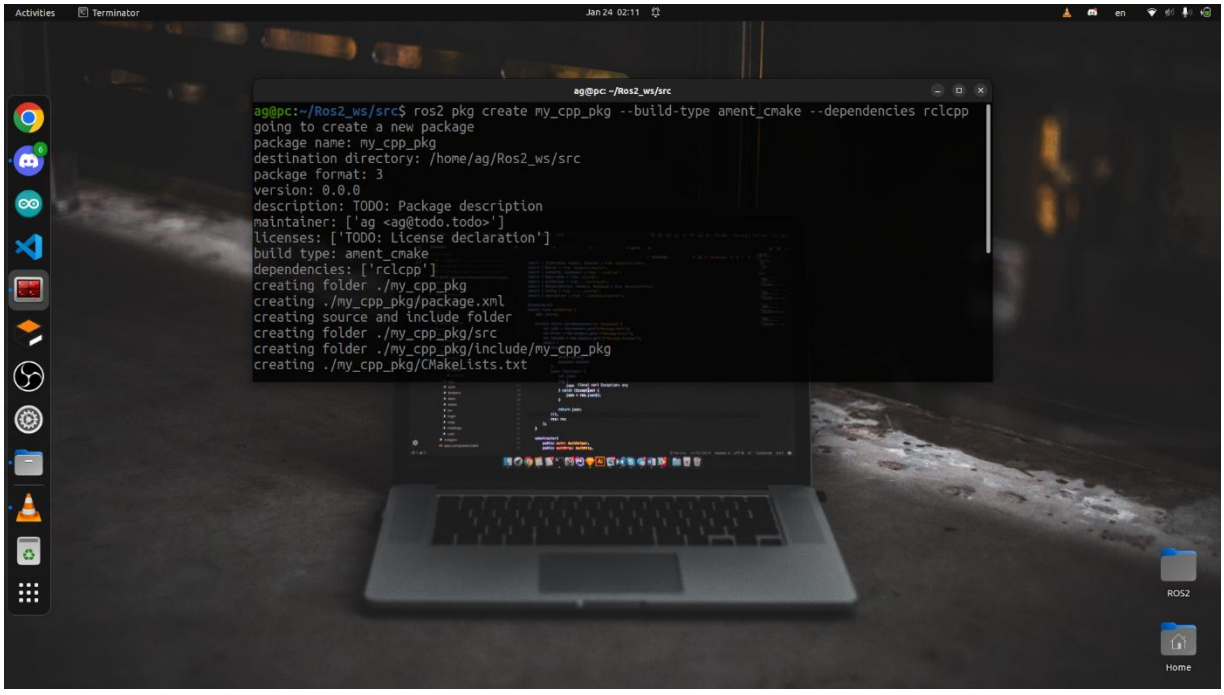


FIGURE 71 CREATE NEW CPP PACKAGE

4.3 ROS2 Nodes:

In ROS 2, a node is a fundamental unit of computation. It represents a process that performs specific tasks, such as controlling a sensor, processing data, or commanding a robot. Nodes are the building blocks of a ROS 2 system, and multiple nodes can communicate with each other to achieve complex behavior.

4.3.1 Key Features of ROS 2 Nodes:

1. **Decentralized Communication:** Nodes communicate using a publish/subscribe model, services, or actions, without needing to know each other's existence directly.
2. **Isolation:** Each node runs in its own process, ensuring robustness. If one node crashes, others are unaffected.
3. **Modularity:** Nodes are designed to perform specific tasks, promoting modular design and code reusability.

- **Creating and Using ROS 2 Nodes:**
  ROS 2 nodes can be written in various programming languages, such as Python or C++, using client libraries like ‘rclcpp’ (C++) or ‘rclpy’ (Python).

- **Running a ROS 2 Node:**
  1. **Build the Node:**
     For Python nodes, no compilation is needed. Just ensure the package is built with colcon.
     For C++ nodes, use colcon build to compile the code.
  2. **Source the Workspace:**
     Before running a node, source the workspace:
     ~ source install/setup.bash
  3. **Run the Node:**
     Use the ros2 run command:
     ~ ros2 run <package_name> <node_executable>

4.3.2 Node Communication Methods:

Nodes in ROS 2 interact using the following mechanisms:

1. **Topics:**
   For publish/subscribe communication.
   Example: A sensor node publishes data, and other nodes subscribe to process it.
2. **Services:**
   For request/response interactions.
   Example: A node requests the current robot pose from another node.
3. **Actions:**
   For long-running tasks with feedback.
   Example: A node commands a robot to navigate to a point while receiving progress updates.

4.3.3 Benefits of ROS 2 Nodes:

1. **Scalability:** Nodes can be distributed across multiple computers in a network.
2. **Fault Isolation:** One node's failure won't bring down the entire system.
3. **Code Reusability:** Nodes encapsulate functionality, making them easy to reuse in other projects.

4.3.4 Types of ROS2 nodes:

1. Publisher Nodes

Purpose: Publish data to a specific topic.
Use Case: A node that collects data from a sensor and broadcasts it to other nodes.
Example: A node publishing data from a LiDAR sensor to a topic `/scan`.

```
self.publisher_ = self.create_publisher(String, 'topic_name', 10)
self.publisher_.publish(String(data='Hello!'))
```

2. **Subscriber Nodes**
   Purpose: Subscribe to a topic to receive and process data.
   Use Case: A node that processes sensor data, such as obstacle detection.
   Example: A node subscribing to `/cmd_vel` to control a robot's movement.

```
self.subscription_=self.create_subscription(String,'topic_name',self.listener_callback,
10)
def listener_callback(self, msg):
self.get_logger().info(f'Received: {msg.data}')
```

3. **Service Nodes**
   Purpose: Offer a service that other nodes can request.
   Use Case: A node that provides information, such as returning the robot's current position.
   Example: A node providing a service to reset the robot's state.
   self.service = self.create_service(Empty, 'reset_robot', self.service_callback)
   def service_callback(self, request, response):
   self.get_logger().info('Resetting robot...')
   return response

4. **Client Nodes**
   Purpose: Request services from other nodes.
   Use Case: A node requesting a map from a mapping node.
   Example: A client node requesting the robot's current location.
   self.client = self.create_client(Empty, 'reset_robot')
   future = self.client.call_async(Empty.Request())

5. **Action Nodes**
   Purpose: Execute long-running tasks with progress feedback.
   Use Case: A node that handles robot navigation to a specific goal.
   Types:
   - Action Servers: Provide an action interface.
   - Action Clients: Request actions and receive feedback.

   Example: A robot navigation action that sends feedback on progress.
   self.server=ActionServer(self,Navigate, 'navigate_to_goal', self.execute_callback)

6. **Transform Nodes (TF Nodes)**
   Purpose: Manage and broadcast coordinate frame transformations.
   Use Case: A node that tracks the position of a sensor relative to the robot base.
   Example: Broadcasting transformations for a camera mounted on a robot arm.
   Common Tools: 'tf2_ros' for transformations.

7. **Parameter Nodes**
   Purpose: Provide dynamic configuration using parameters.
   Use Case: A node whose behavior (e.g., control gains) can be adjusted at runtime.
   Example: Dynamically changing the update rate of a sensor node.
   self.declare_parameter('update_rate', 10.0)
   rate = self.get_parameter('update_rate').value

8. **Lifecycle Nodes**
   Purpose: Enable managed lifecycle states for nodes (e.g., startup, running, shutdown).
   Use Case: A node that needs controlled initialization and cleanup, such as hardware drivers.
   Lifecycle States:
   `unconfigured` , `inactive` , `active` , `finalized`
   Example: A robot arm driver node that initializes hardware only when transitioning to the `active` state.

9. **Driver Nodes**
   Purpose: Interface with hardware devices, such as sensors and actuators.
   Use Case: A node that reads data from a LiDAR or controls a motor.
   Example: A node managing a camera driver.

10. **Middleware Bridge Nodes**
    Purpose: Act as a bridge between different communication protocols or middleware.
    Use Case: Connecting ROS 2 systems with non-ROS systems or ROS 1 systems.
    Example: A node bridging ROS 1 and ROS 2 using the `ros1_bridge` package.

11. **Monitoring Nodes**
    Purpose: Monitor system health or log data for debugging.
    Use Case: A node that logs CPU usage or checks the status of other nodes.
    Example: A diagnostics node reporting system performance metrics.

12. **Simulation Nodes**
    Purpose: Simulate sensors, actuators, or entire robot behavior.
    Use Case: A node simulating LiDAR data in Gazebo or RViz.
    Example: A node publishing simulated odometry data.

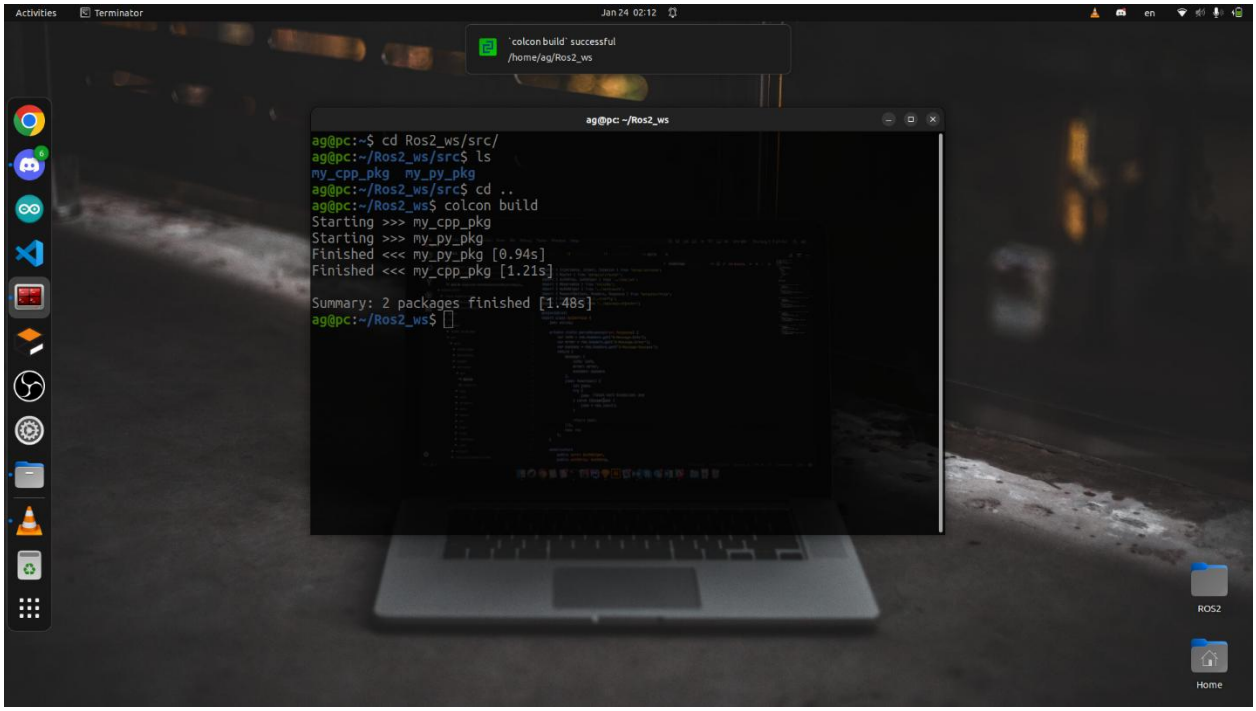


FIGURE 72 BUILDING ALL PACKAGES IN WORKSPACE

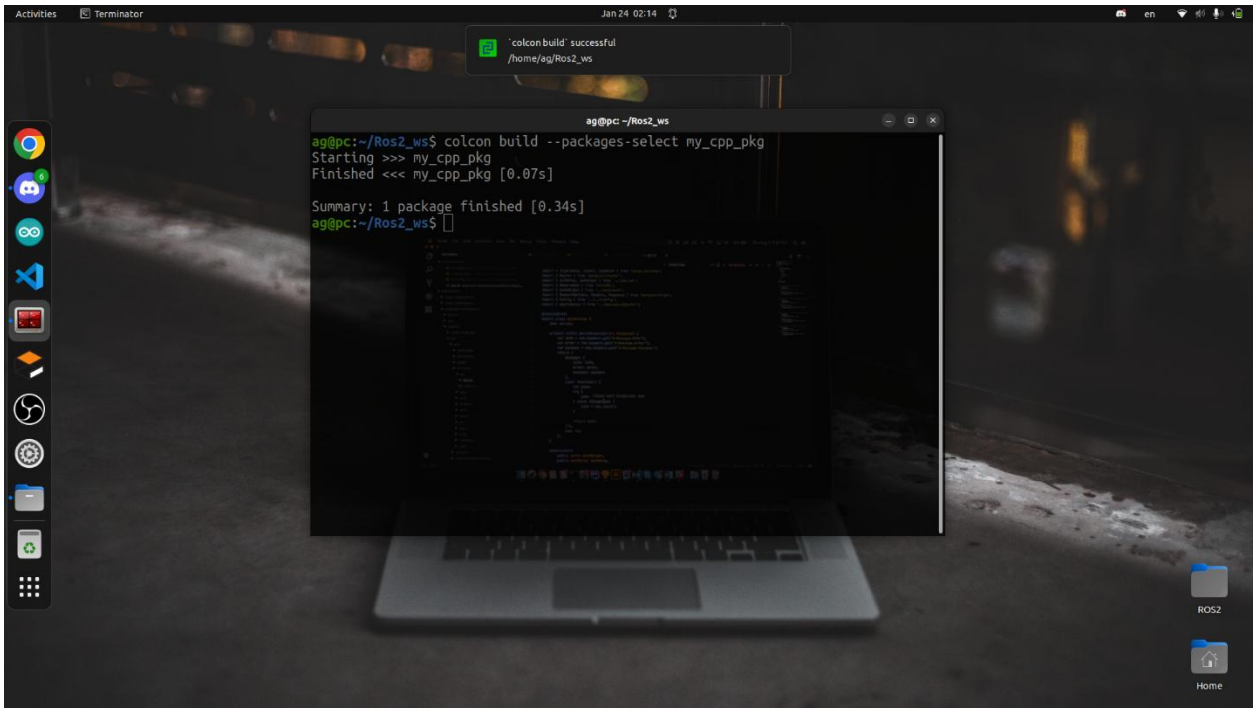


FIGURE 73 SELECTING A SPECIFIC PACKAGE TO BUILD

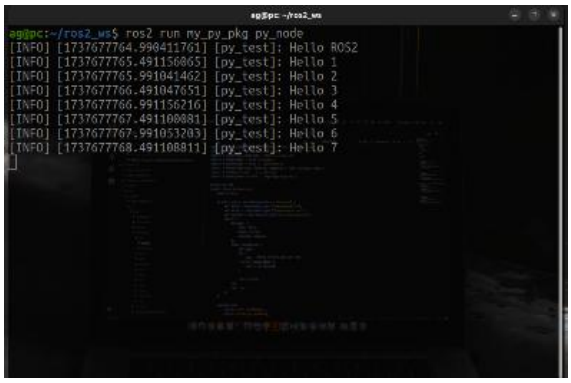


FIGURE 74 RUN A PYTHON NODE

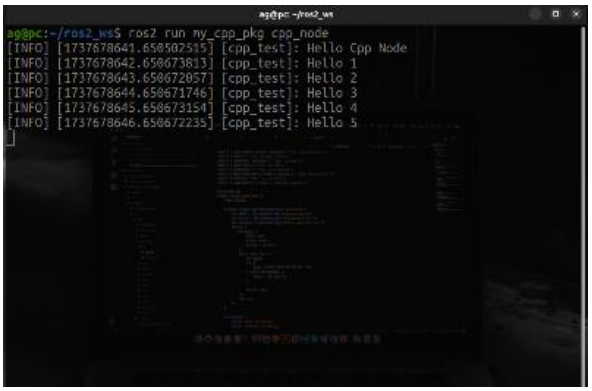


FIGURE 75 RUN A CPP NODE

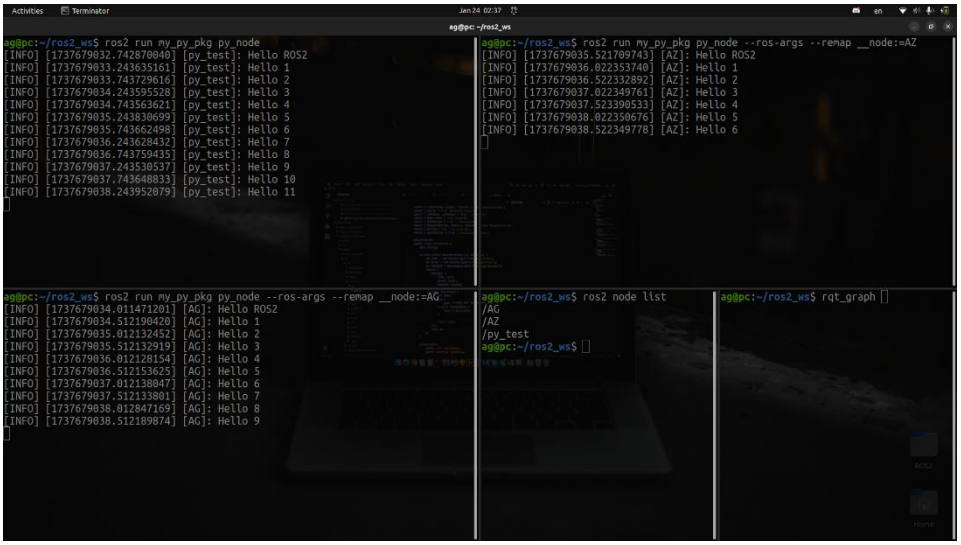


FIGURE 76 RUN A PYTHON AND CPP NODES

4.4 Command-line arguments:

1. **Rename a Node Using Command-Line Arguments**
   When launching a ROS 2 node, you can rename it using the '--ros-args' option with the `--remap` flag. The syntax is:
   ~ ros2 run <package_name> <node_executable> --ros-args –remap __node:=<new_node_name>

2. **Verify the Node Name**
   After renaming the node, you can check its new name using:
   ~ ros2 node list

3. **Renaming Nodes in Launch Files**
   If you are using a launch file, you can specify the new name using the `node_name` parameter or through remapping rules.
   ~ ros2 launch <your_package> rename_node_launch.py

4. **Runtime Considerations**
   - **Name Collisions:** If a node with the same name already exists in the system, the newly renamed node will fail to start. Ensure unique names for each node.
   - **Parameters:** If your node reads parameters from the parameter server, ensure the renamed node has the correct parameter namespace.

5. **Use Cases**
   - Debugging multiple instances of a node by assigning unique names at runtime.
   - Running the same node in different roles (e.g., multiple robots).
   - Dynamically assigning node names in multi-robot systems.

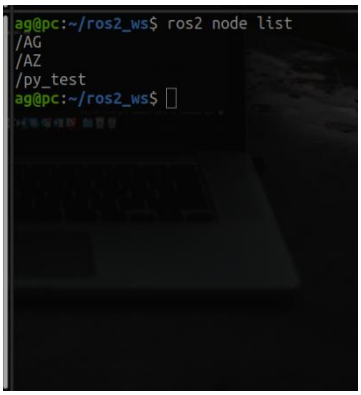


FIGURE 77 NODE LIST COMMAND

4.5 RQT

Rqt is a Qt-based graphical user interface (GUI) framework for ROS that provides a collection of plugins to visualize and interact with nodes, topics, services, parameters, and more.

4.5.1 Features of Rqt

- **Modular:** Supports various plugins that can be added based on requirements.
- **GUI-Based Debugging:** Provides a graphical way to debug and inspect ROS nodes and their interactions.
- **Extensible:** New plugins can be developed to extend its functionality.

4.5.2 Commonly Used Rqt Plugins

- **rqt_console:** Displays log messages and filters them by severity or node.
- **rqt_graph:** Visualizes the communication graph of nodes and topics.
- **rqt_plot:** Plots numeric data from topics in real-time.
- **rqt_logger_level:** Adjusts the logging levels of nodes at runtime.
- **rqt_reconfigure:** Dynamically reconfigures node parameters.
- **rqt_tf_tree:** Displays the TF (transform) tree for debugging transformations.

4.5.3 How to Launch Rqt

- To start Rqt with the default interface:

  ~ rqt
- To launch Rqt with a specific plugin:

  ~ rqt --standalone <plugin_name>

4.6 rqt_graph

rqt_graph is a plugin within Rqt that visualizes the communication graph of ROS nodes and topics. It provides an intuitive way to understand how nodes are interacting with each other through topics, services, and actions.

4.6.1 Features of rqt_graph

- **Node Visualization:** Displays nodes in the system and their connections.
- **Topic Visualization:** Shows the topics being published or subscribed to.
- **Dynamic Updates:** Reflects real-time changes in the communication graph.
- **Color-Coded Display:** Differentiates between publishers, subscribers, and topics.

4.6.2 How to Launch rqt_graph

You can launch rqt_graph as part of Rqt or directly as a standalone tool:

```
~ rqt_graph
```

Example Output

- **Nodes:** Represented as boxes with their names.
- **Topics:** Represented as circles connecting nodes.
- **Arrows:** Indicate the direction of data flow (publisher to subscriber).

4.6.3 Use Cases for Rqt and rqt_graph

- **System Debugging:**
  - Use rqt_console to monitor logs for errors or warnings.
  - Use rqt_graph to visualize communication and identify missing connections.
- **Performance Monitoring:**
  Use rqt_plot to monitor sensor data or control commands.
- **Dynamic Parameter Adjustment:**
  Use rqt_reconfigure to tune parameters in real-time.
- **Transform Debugging:**
  Use rqt_tf_tree to debug the TF hierarchy and transformations.

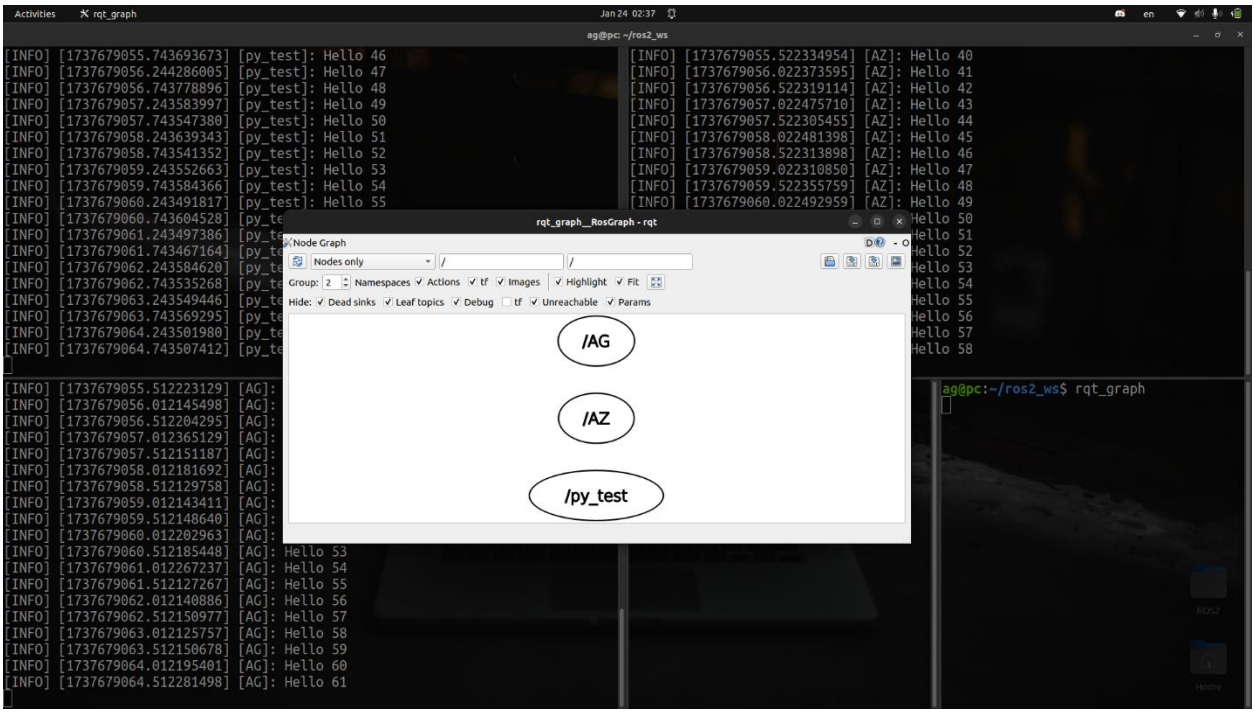


FIGURE 78 RQT GRAPH FOR THE CURRENT NODES

4.7 ROS2 Topics

A ROS 2 topic is a communication channel used for publish/subscribe messaging between nodes. It is a core concept in ROS 2, enabling nodes to exchange data asynchronously.

4.7.1 Key Features of Topics:

- **Publisher-Subscriber Model:** Nodes publish messages to a topic, and other nodes subscribe to the topic to receive those messages.
- **Asynchronous Communication:** Publishers and subscribers do not need to know each other or exist simultaneously.
- **Message Types:** Topics use specific message types (e.g., `std_msgs/msg/String`, `sensor_msgs/msg/Image`) to structure data.
- **Anonymous Communication:** Nodes can interact without knowing the identity of other nodes.

Example:

- **Publisher:** A node publishing sensor data (e.g., `/camera/image_raw`).
- **Subscriber:** A node processing the sensor data (e.g., image analysis).

4.7.2 Debug ROS 2 Topics with Command-Line Tools

1. **List Active Topics**
   To see all currently active topics:
   ~ ros2 topic list
2. **Echo Topic Data**
   To view real-time data being published on a topic:
   ~ ros2 topic echo <topic_name>
3. **Get Topic Type**
   To find out the type of a topic:
   ~ ros2 topic type <topic_name>
4. **Get Topic Info**
   To get detailed information about a topic:
   ~ ros2 topic info <topic_name>

5. **Publish Messages to a Topic**
   To manually publish data to a topic:
   ~ ros2 topic pub <topic_name> <message_type> "{data}"
6. **Monitor Topic Bandwidth**
   To measure the bandwidth usage of a topic:
   ~ ros2 topic bw <topic_name>

4.7.3 Remap a Topic at Runtime

Topic remapping in ROS 2 allows you to dynamically redirect one topic to another at runtime. This is particularly useful for modular systems or when reusing nodes in different applications.

- **Remap Topics via Command Line**

  Use the `--ros-args --remap` option:
  ~ ros2 run <package_name> <node_executable> --ros-args --remap <original_topic_name>:=<new_topic_name>

- **Remap Topics in a Launch File**

  You can also specify remapping rules in launch files.

4.8 ROS2 Publishers

In ROS 2, a Publisher is a component that allows nodes to send messages to topics. A node that publishes information to a topic is called a publisher. Publishers are part of the publish-subscribe communication model, where publishers send messages, and subscribers receive them.

4.8.1 Key Features of ROS 2 Publishers:

1. **Message Sending:** Publishers send data to topics, which can be subscribed to by other nodes.
2. **Topic:** A publisher is always linked to a specific topic, and each topic can have multiple publishers or subscribers.
3. **Types of Messages:** The type of message published (e.g., `std_msgs/msg/String`, `sensor_msgs/msg/Image`) must be predefined and consistent across publishers and subscribers.
4. **Publisher Lifetime:** The publisher exists as long as the node is running and can be used to continuously send messages.

4.8.2 Debug Publisher with ROS 2 Tools

To debug a publisher and inspect the messages it sends, you can use the following ROS2 tools:

1. **View Published Topics:**

To list all the available topics:

~ ros2 topic list

2. **Inspect Messages on a Topic:**

To see the messages being published on a specific topic (e.g., `/chatter`):

~ ros2 topic echo /chatter

This command will print the data being sent by the publisher on the `chatter` topic.

3. **Check the Type of a Topic:**

To check the message type being published to a specific topic:

~ ros2 topic type /chatter

4. **Publish a Message to a Topic (Test Publisher):**

To manually publish a message to a topic for testing purposes:

~ ros2 topic pub /chatter std_msgs/msg/String "data: 'Hello, ROS 2!'"

5. **Check Publisher Information:**

To check which nodes are publishing to a specific topic:

~ ros2 topic info /chatter

4.8.3 Remap a Publisher at Runtime

In ROS2, you can remap topics at runtime, which allows you to change the topic name that a publisher (or subscriber) is sending (or receiving) messages on. This can be useful in many scenarios, such as running multiple nodes that publish to similar topics or when adjusting system configurations without modifying the code.

Remapping Topics in ROS2:

You can use the `ros2 topic` command to remap topics at runtime or you can specify topic remappings in your launch files or when launching nodes.

Remap from Command Line:

You can remap a topic when launching the node directly from the command line using the `ros2 run` command with remapping arguments:
~ ros2 run my_package minimal_publisher --ros-args -r /chatter:=/new_chatter

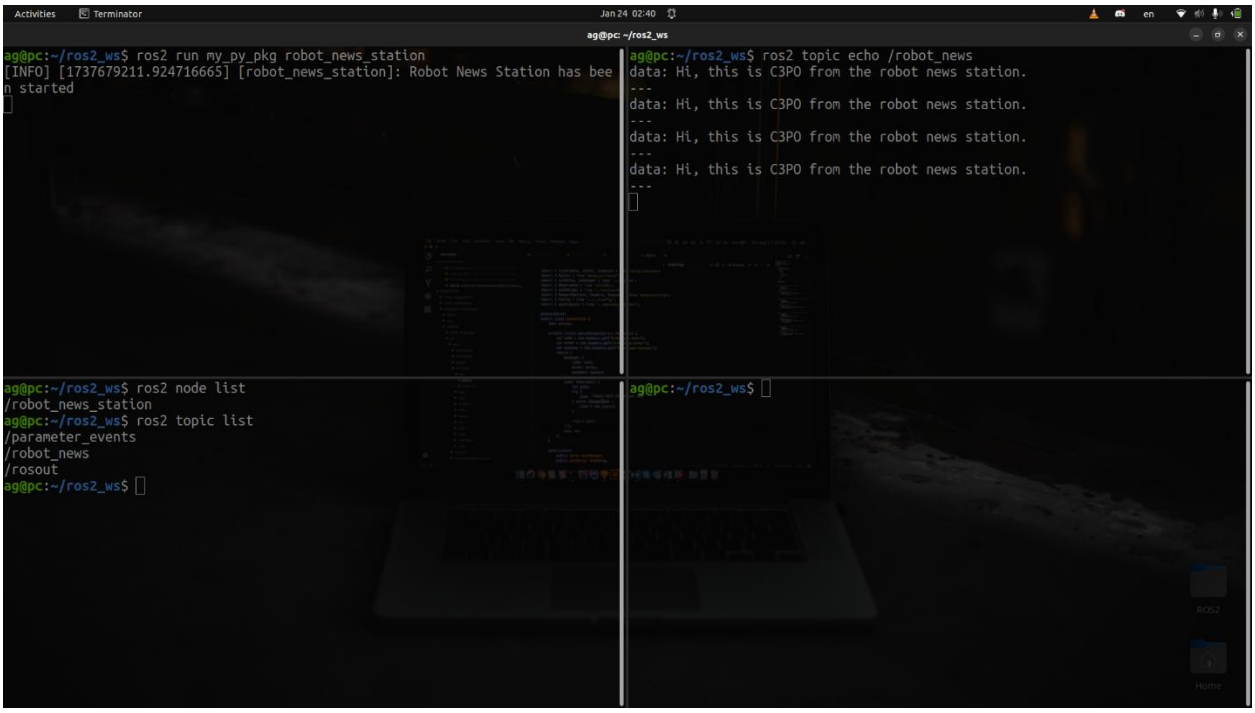


FIGURE 79 RUN A PUBLISHER VIA ROS2 TOPIC ECHO

4.9 ROS2 Subscribers

In ROS 2, a Subscriber is a component that receives messages from a topic. A node that subscribes to a topic and processes incoming messages is called a subscriber. Subscribers follow the publish-subscribe communication pattern, where publishers send messages to topics, and subscribers receive those messages.

4.9.1 Key Features of ROS 2 Subscribers:

1. **Receiving Messages:** Subscribers are used to receive data from topics that other nodes (publishers) are sending.
2. **Callback Functions:** Subscribers are typically associated with a callback function that is executed when a new message is received.
3. **Topic:** A subscriber listens to a specific topic, and it must subscribe to a topic that matches the message type being published by the publisher.
4. **Asynchronous:** The ROS 2 subscriber system is designed to be asynchronous, meaning subscribers are notified when a new message is available, without blocking the execution of the node.

4.9.2 Debug Subscriber with ROS 2 Tools

To debug a subscriber and inspect the messages it receives, you can use the following ROS 2 tools:

1. **View Subscribed Topics:**
   To list all available topics:
   ~ ros2 topic list
2. **Inspect the Messages Received by a Subscriber:**
   To see the messages being received by a specific subscriber (e.g., `/chatter`):
   ~ ros2 topic echo /chatter
   This command will display the messages being published on the `chatter` topic, and your subscriber will process these messages.
3. **Check the Type of a Topic:**
   To check the message type being published to a specific topic:
   ~ ros2 topic type /chatter
4. **Check Subscriber Information:**
   To check which nodes are subscribing to a specific topic:
   ~ ros2 topic info /chatter
5. **View Subscribed Data in Real-Time:**
   You can use the `ros2 topic echo` command to view incoming data on the subscriber's side in real-time. It helps you monitor what the subscriber is receiving from the publisher.

4.9.3 Remap a Subscriber at Runtime

In ROS 2, you can remap topics at runtime, which allows you to change the topic name that a subscriber is listening to. This is useful when you want to dynamically change the topic a node subscribes to without changing the code.

Remapping Topics in ROS 2:

You can use remapping to change the topic a subscriber listens to, either by modifying the launch file or by specifying remaps in the command line when running the node.

Remap from Command Line:

You can also remap topics directly when running the node from the command line:

~ ros2 run my_package minimal_subscriber --ros-args -r /chatter:=/new_chatter

This will remap the `chatter` topic to `new_chatter` at runtime for the subscriber node.

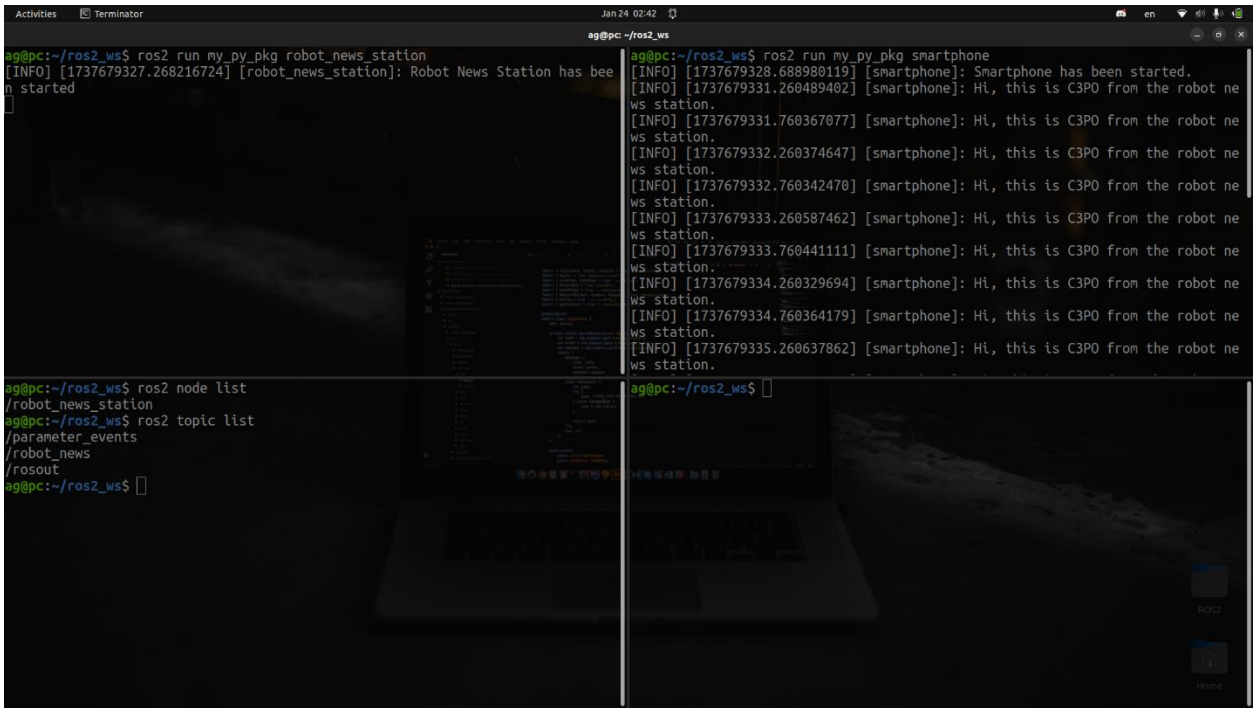


FIGURE 80 RUN A SUBSCRIBER

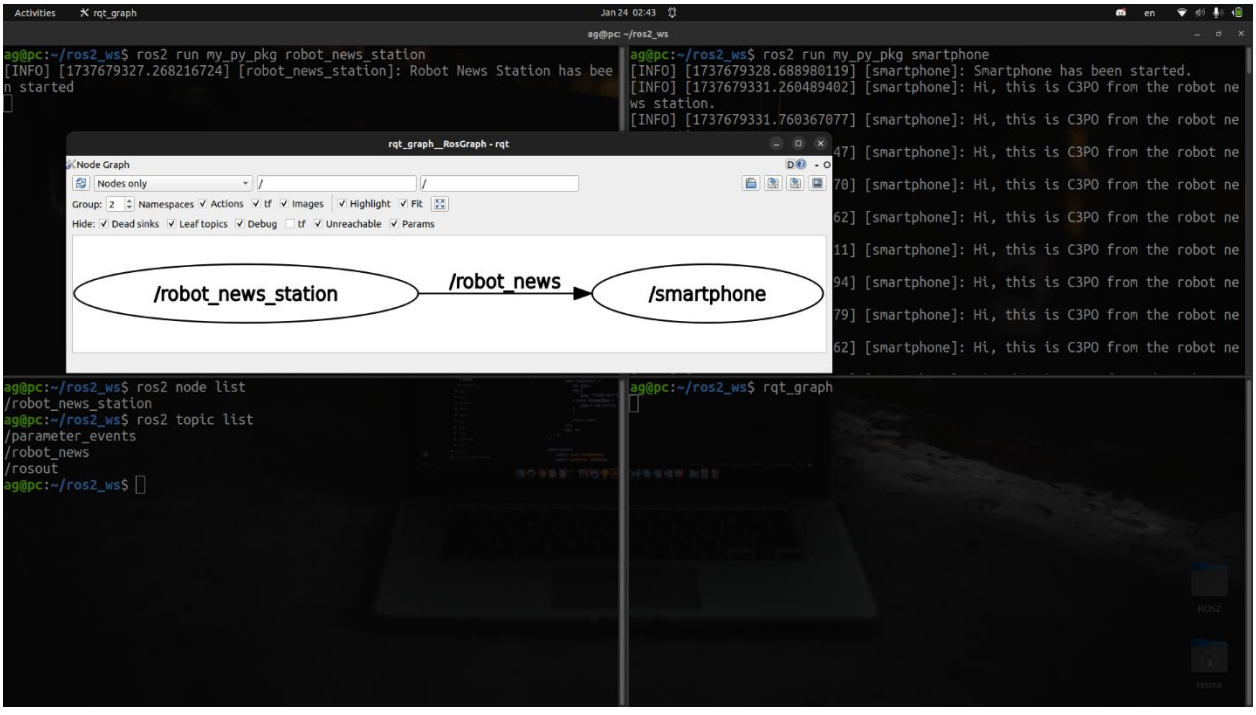


FIGURE 81 RQT GRAPH FOR THE CURRENT TOPICS

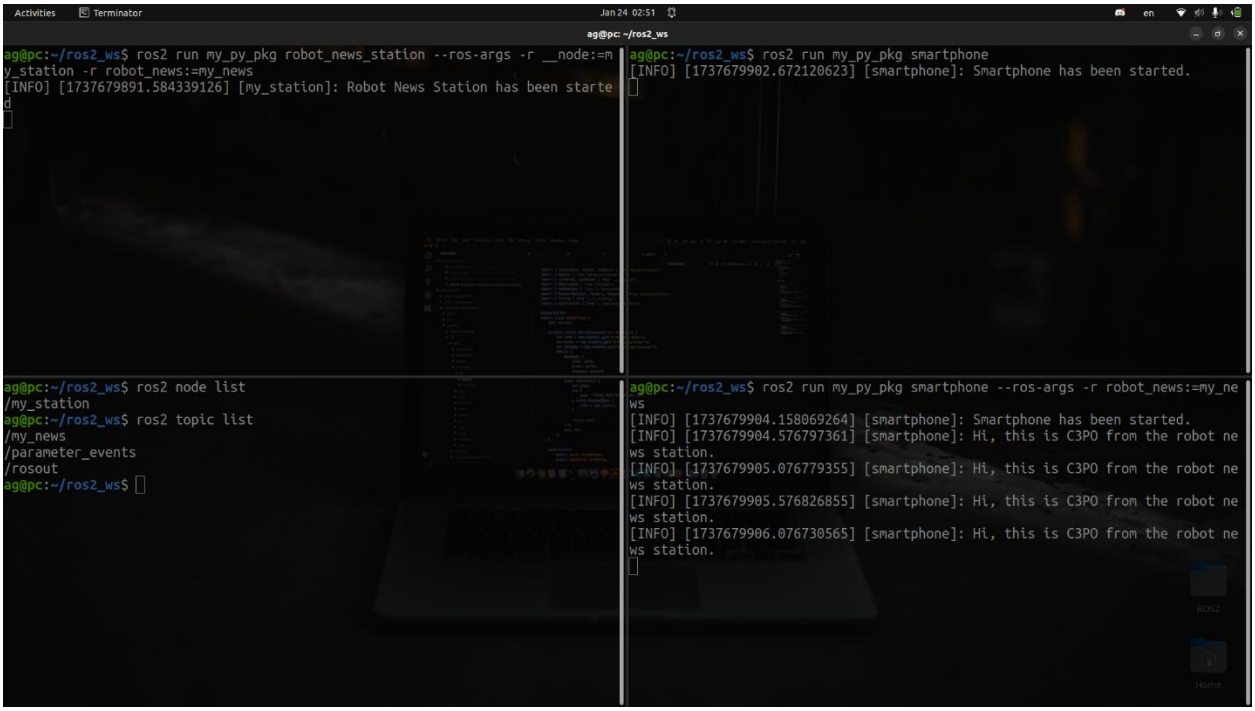


FIGURE 82 REMAPPING THE TOPICS

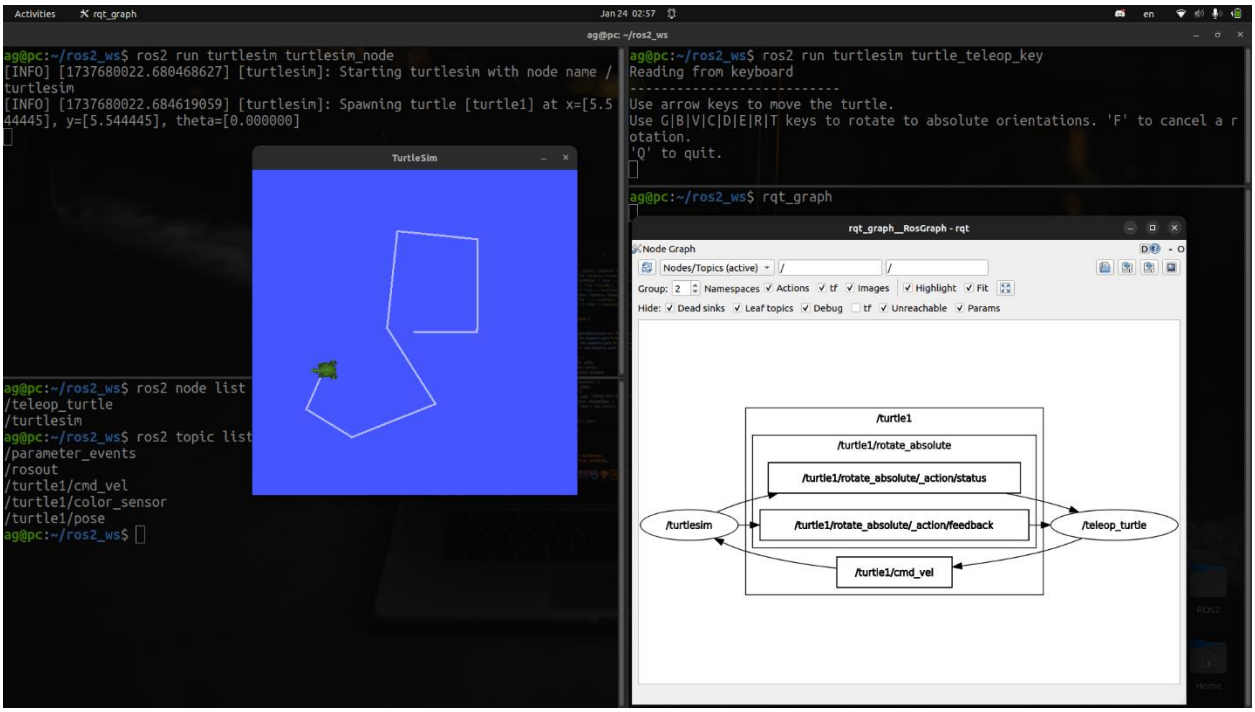


FIGURE 83 RUN TURTLESIM NODE AND DISPLAYING RQT GRAPH

4.10 ROS2 Services

A ROS 2 Service is a request/response communication mechanism between nodes. Unlike topics, which enable asynchronous communication, services allow a node to send a request and wait for a response from another node.

4.10.1 Key Features of Services:

1. **Synchronous Communication:** A client sends a request, and the server processes it and sends back a response.
2. **Service Type:** Each service is associated with a specific type that defines the structure of the request and response.
3. **Use Cases:** Used for operations that require a direct response, like triggering an action, querying data, or changing a system's state.

4.10.2 Components of a ROS 2 Service:

Service Server: Provides the service by implementing the logic to handle requests.

Service Client: Requests the service and waits for the response.

Example Service:

- **Service Name:** `/add_two_ints`
- **Service Type:** `example_interfaces/srv/AddTwoInts`
- **Request Two:** integers to be added.
- **Response:** The sum of the two integers.

Command to List Services

~ ros2 service list

4.10.3 Debug Services with ROS 2 Tools

1. **List Available Services**
   To see all currently active services:
   ~ ros2 service list
2. **Get Service Type**
   To determine the type of a specific service:
   ~ ros2 service type <service_name>
   Example:
   ~ ros2 service type /add_two_ints
3. **Show Service Description**
   To display the request and response structure of a service type:
   ~ ros2 interface show <service_type>
   Example:
   ~ ros2 interface show example_interfaces/srv/AddTwoInts
4. **Call a Service**
   To manually call a service and send a request:
   ~ ros2 service call <service_name> <service_type> "{<request_fields>}"
   Example:
   ~ ros2 service call /add_two_ints example_interfaces/srv/AddTwoInts "{a: 2, b: 3}"
5. **Check Service Detail**
   To get detailed information about a service:
   ~ ros2 service info <service_name>

4.10.4 Remap a Service at Runtime

- **Remap Services via Command Line**
  Use the `--ros-args --remap` option when launching a node to remap a service name.
  ~ ros2 run <package_name> <node_executable> --ros-args --remap <original_service_name>: =<new_service_name>
- **Remap Services in a Launch File**
  You can specify remapping rules in a launch file.

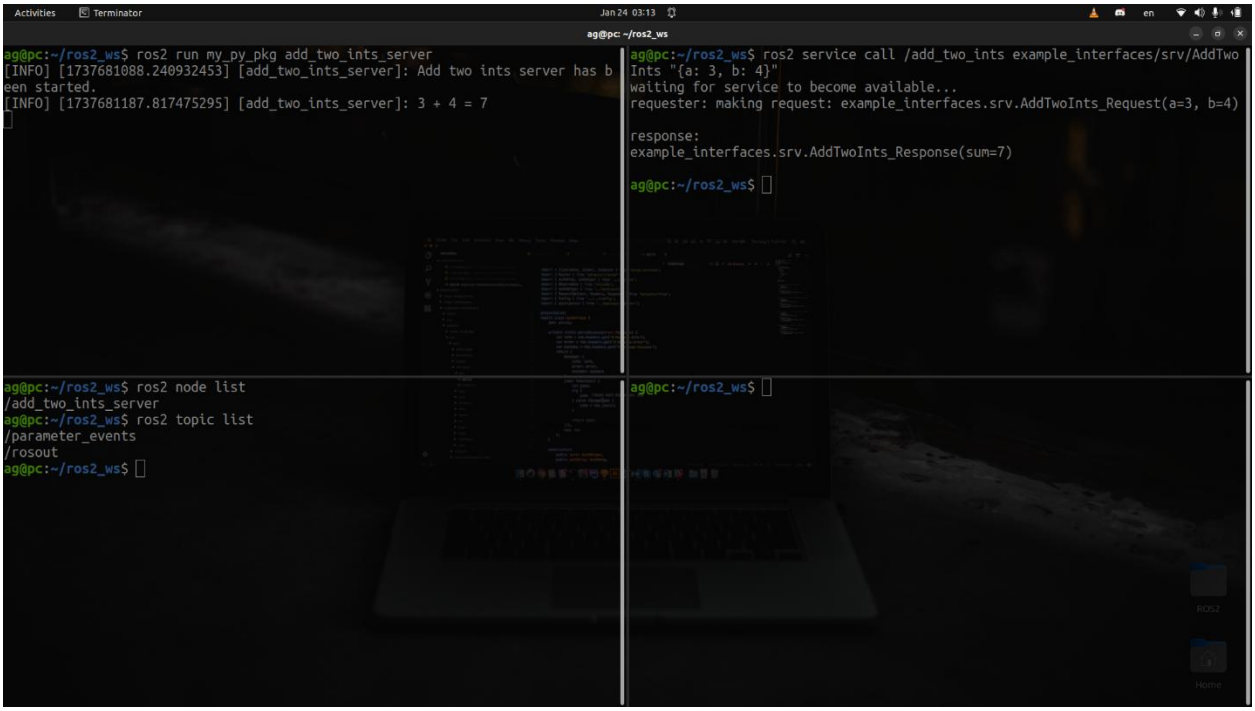


FIGURE 84 RUN A SERVER AND CALL A SERVICE

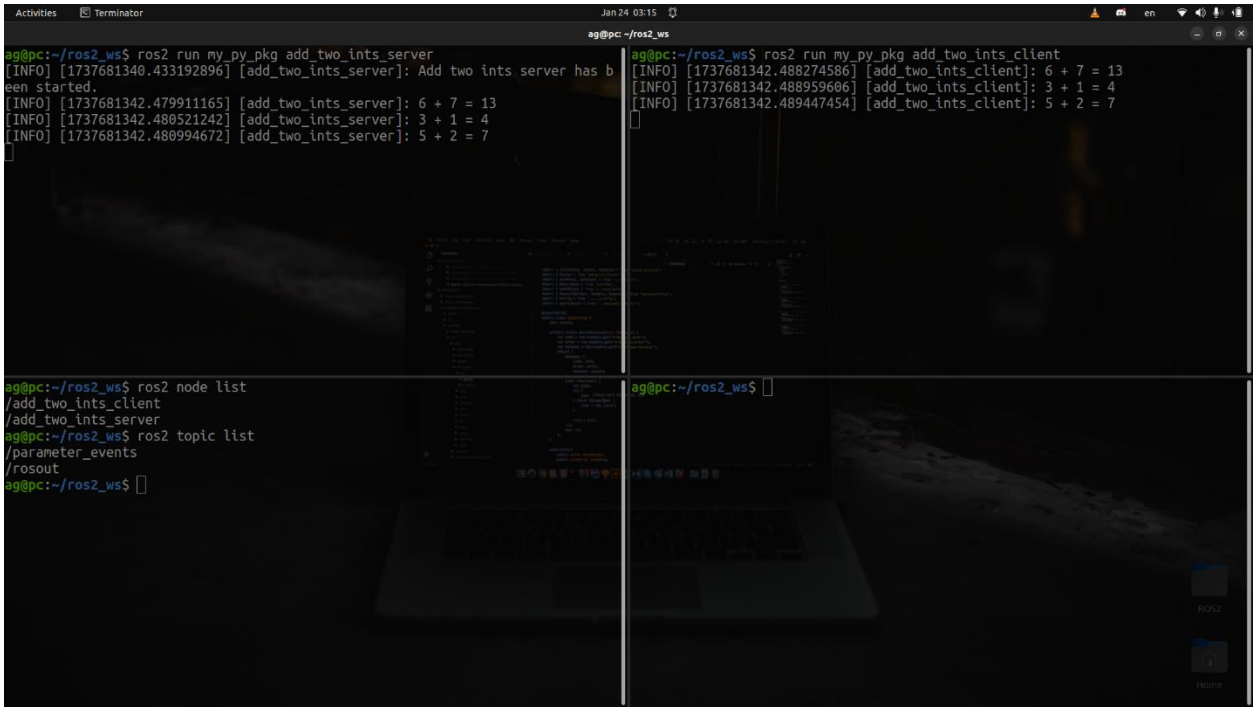


FIGURE 85 RUN A SERVICE AND A CLIENT

4.11 ROS2 Interfaces

In ROS 2, interfaces define the structure of data exchanged between nodes. They specify the format of messages, services, and actions, enabling consistent communication.

4.11.1 Types of ROS 2 Interfaces:

1. **Messages (`msg`):**
   Define the structure of data sent via topics.
   Examples: `std_msgs/msg/String`, `sensor_msgs/msg/Image`.
2. **Services (`srv`):**
   Define the request and response structure for services.
   Examples: `example_interfaces/srv/AddTwoInts`.
3. **Actions (`action`):**
   Define the structure for action goals, feedback, and result.

4.11.2 Debug Messages and Services with ROS 2 Tools

1. **List All Available Interfaces**
   To list all available message, service, and action types:
   ~ ros2 interface list
2. **Show Interface Details**
   To display the structure of a specific interface:
   ~ ros2 interface show <interface_type>
3. **Debugging Topics Using Messages**
   Use `ros2 topic echo` to view messages:
   ~ ros2 topic echo /topic_name
   Use `ros2 topic pub` to publish a message:
   ~ ros2 topic pub /topic_name <message_type> "{<field_name>: <value>}"
4. **Debugging Services**
   Call a service:
   ~ ros2 service call <service_name> <service_type> "{<request_fields>}"
5. **Find the Type of a Topic or Service**
   Get the type of a topic:
   ~ ros2 topic type /topic_name
   Get the type of a service:
   ~ ros2 service type /service_name
6. **Explore Interface Packages**
   To list all message, service, and action types in a specific package:
   ~ ros2 interface package <package_name>

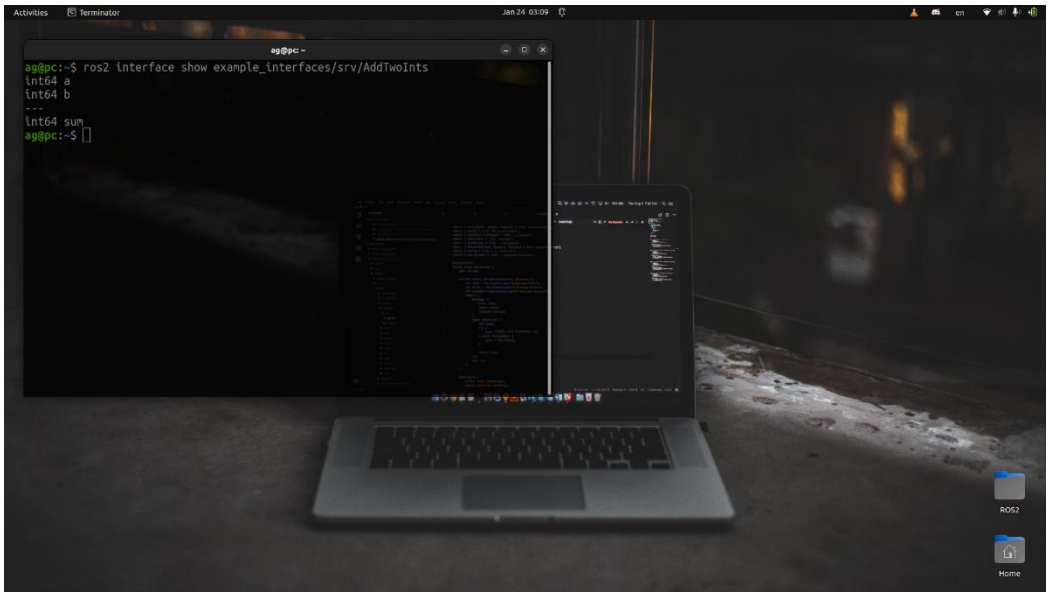


FIGURE 86 DISPLAYING INTERFACES

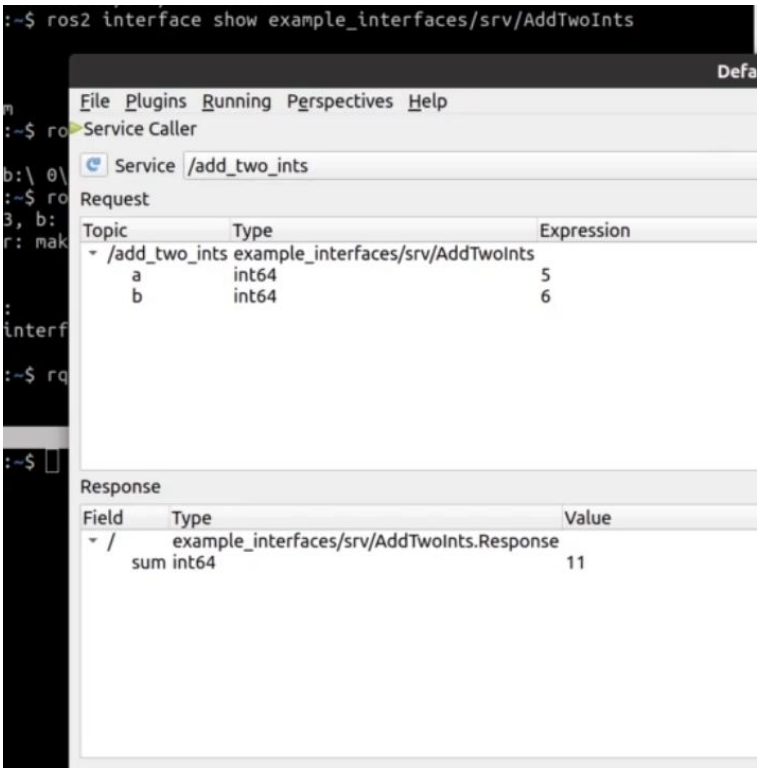


FIGURE 87 DISPLAYING INTERFACES IN RQT GRAPH

4.12 ROS2 Parameters

In ROS2, parameters are values that nodes can use to configure their behavior at runtime. Parameters are typically set when launching a node and can be updated dynamically while the node is running. They enable nodes to be more flexible and reusable, as the same node can behave differently based on the values of its parameters.

4.12.1 Key Features of Parameters:

1. **Dynamic Configuration:** Parameters allow the behavior of nodes to be changed without modifying the source code or restarting the node.
2. **Parameter Types:** Parameters can have various types, such as integers, floats, strings, Booleans, arrays, and even more complex types like lists or dictionaries.
3. **Default Values:** When declaring a parameter, a default value can be provided if no value is set externally.
4. **Access to Parameters:** Nodes can access and modify parameters using the ROS 2 API during runtime.

4.12.2 Common Use Cases for Parameters:

1. Tuning algorithm parameters (e.g., PID gains).
2. Setting configuration values such as operational limits.
3. Specifying settings like control frequencies, timeouts, or paths to files.

Command to List Parameters:

~ros2 param list

Command to Get Parameter Value:

~ros2 param get <node_name> <parameter_name>

Command to Set Parameter Value:

~ros2 param set <node_name> <parameter_name> <parameter_value>

4.12.3 Declare Your Parameters

To declare parameters in ROS 2, you typically do this in the node's code or in the launch file. The declaration ensures that the parameter has a default value and can be accessed by the node. Here is a common way to declare parameters in ROS 2:

- **Declaring Parameters in a Launch File**

  In ROS 2, you can also declare parameters in a launch file to set their values when launching the node. This allows for easy configuration without modifying the code.

4.13 ROS2 Launch File

In ROS 2, a launch file is a way to automate the process of starting multiple nodes and configuring their parameters. Launch files allow you to define a set of instructions that specify which nodes to run, their parameters, remapping's, and other configurations, all in a single place. They provide a more efficient and flexible way to start and manage your ROS 2 system, especially when working with multiple nodes and complex setups.

4.13.1 Key Features of a ROS2 Launch File:

1. **Node Configuration:** You can specify which nodes to run, their parameters, and configurations.
2. **Launching Multiple Nodes:** A launch file can manage the execution of multiple nodes at once, making it easier to coordinate the start-up of a system.
3. **Parameter Setting:** You can set parameters for nodes either from the launch file or through external configuration files.
4. **Remapping Topics and Services:** You can remap topics, services, and actions directly from the launch file.
5. **Lifecycle Management:** ROS 2 introduces lifecycle nodes, which can be controlled (e.g., transition states) using launch files.

4.13.2 Types of Launch Files in ROS2:

1. **Python Launch Files:** In ROS 2, the recommended method for writing launch files is using Python, rather than XML (which was used in ROS 1).
   Python Launch: Allows you to dynamically configure, manipulate, and run nodes based on conditions or parameters.
2. **XML Launch Files**

4.13.3 Launch File Components:

1. **Node:** Specifies which nodes to launch, their names, parameters, etc.
2. **Push Parameters:** Allows you to specify parameters for nodes.
3. **Remapping:** Enables topic, service, or action remapping.
4. **Actions:** Defines tasks such as setting up nodes or changing lifecycle states.

4.13.4 Basic Structure of a ROS 2 Launch File (Python)

- **'LaunchDescription':** This is the main container that holds the launch instructions.
- **'Node':** Defines the node to be launched with the specified package, executable, name, and parameters.
- **'parameters':** A dictionary or list of parameters that will be passed to the node.
- **'remappings':** List of topic or service remappings.

Launching the File

Once you have your launch file ready, you can run it using the `ros2 launch` command:

~ ros2 launch <package_name> <launch_file.py>

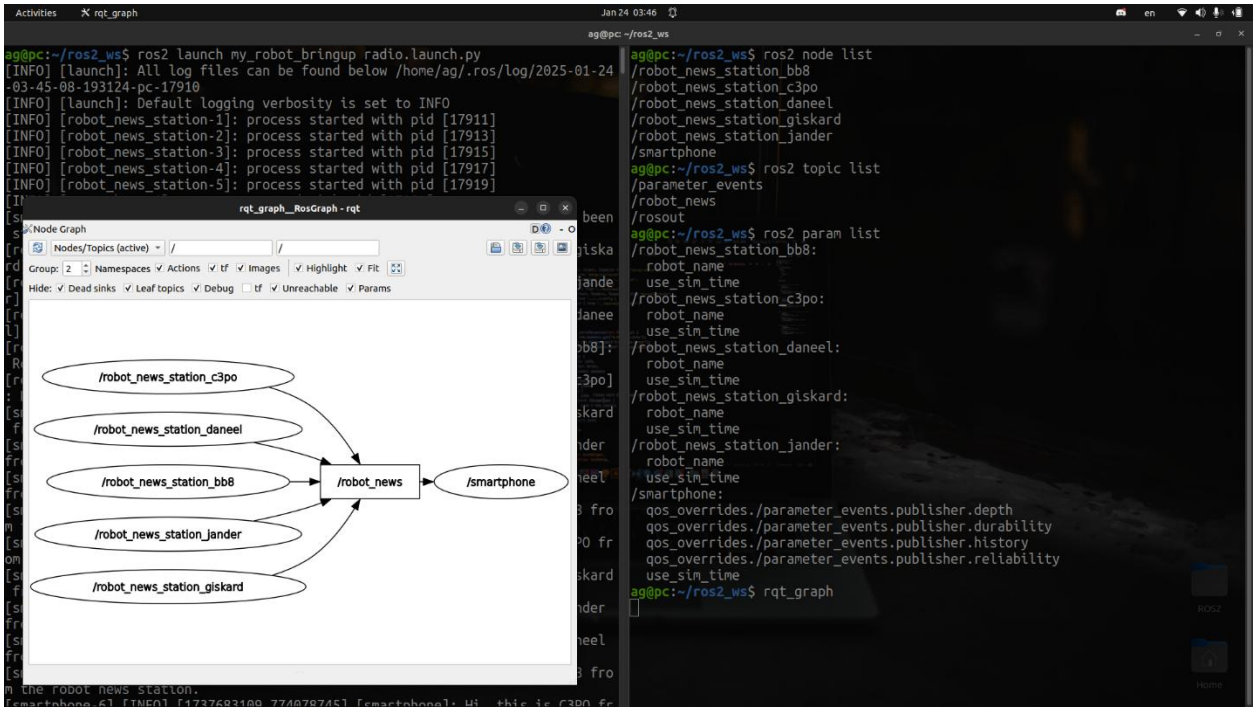
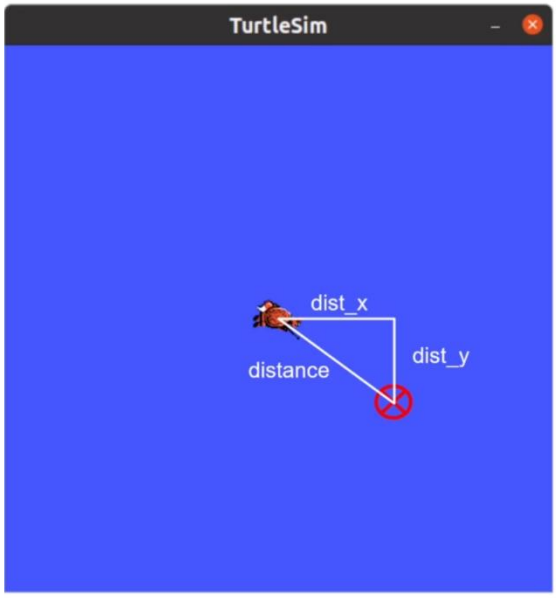
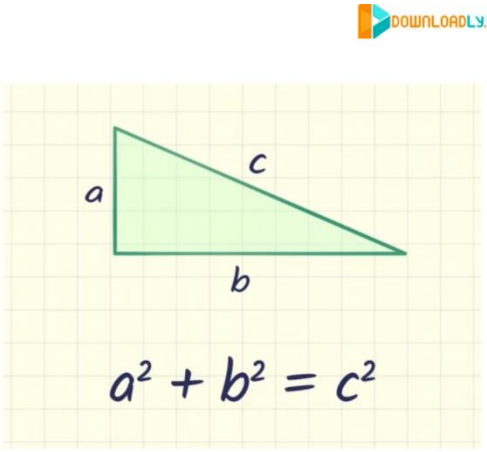
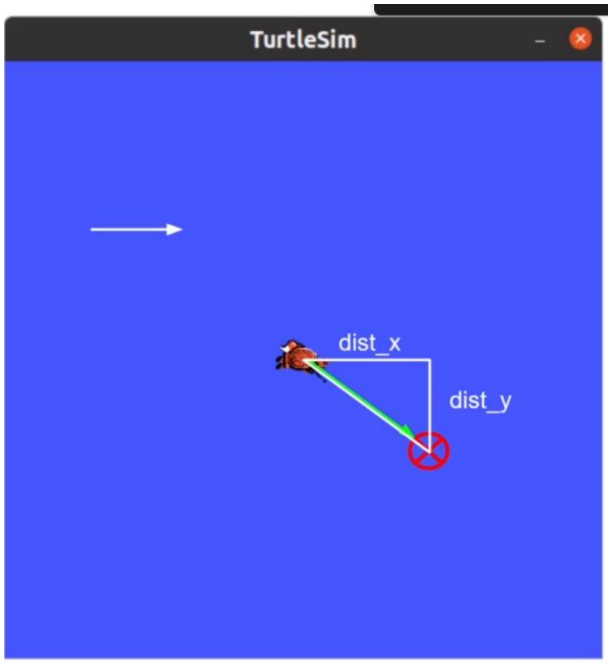


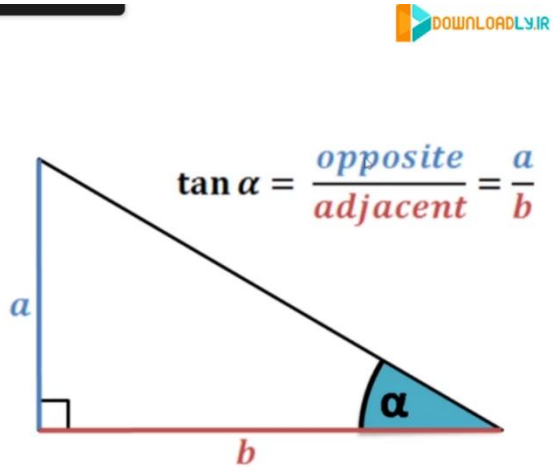
FIGURE 88 LAUNCHING A FILE AND DISPLAYING THE CURRENT NODES-TOPICS-PARAMETERS

Turtlesim “Catch Them All” project









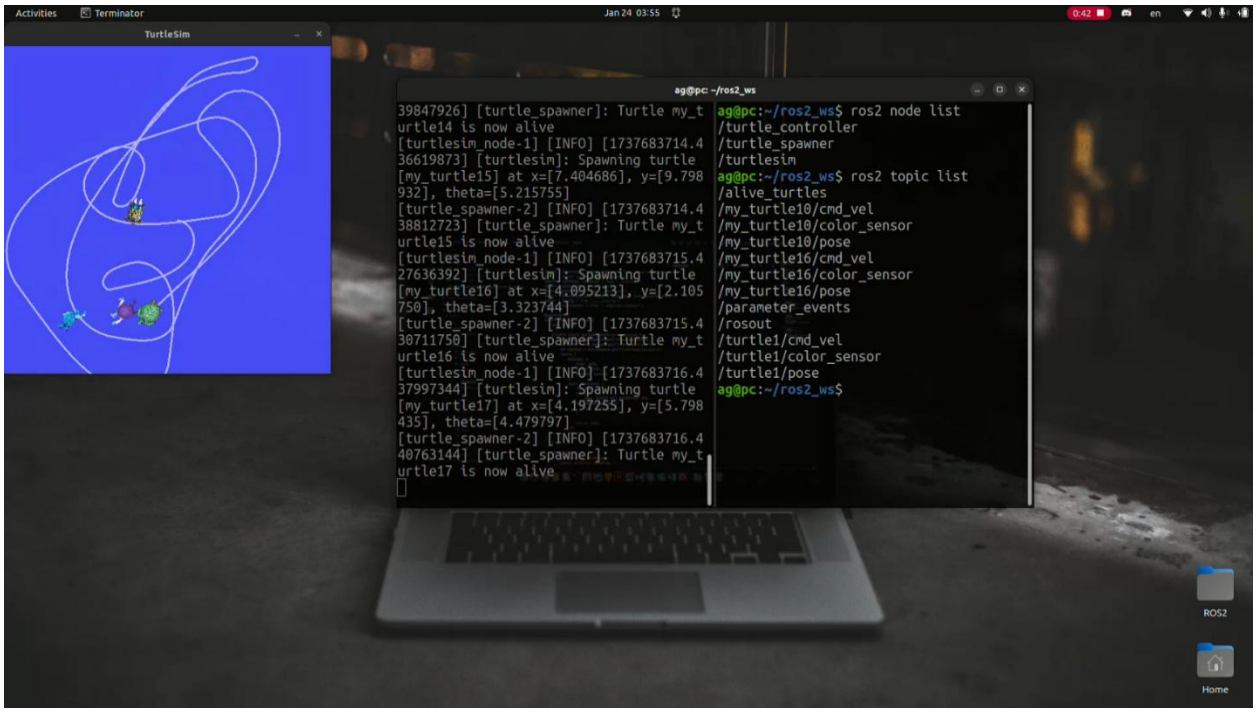


FIGURE 89

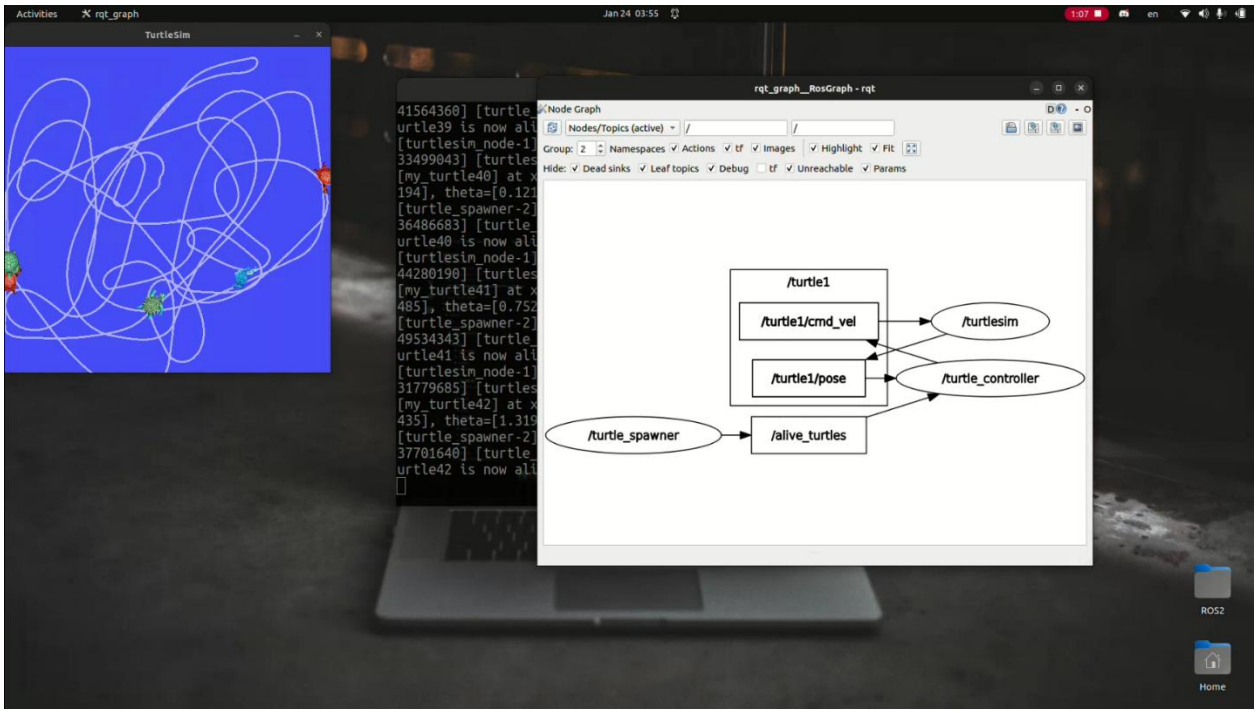
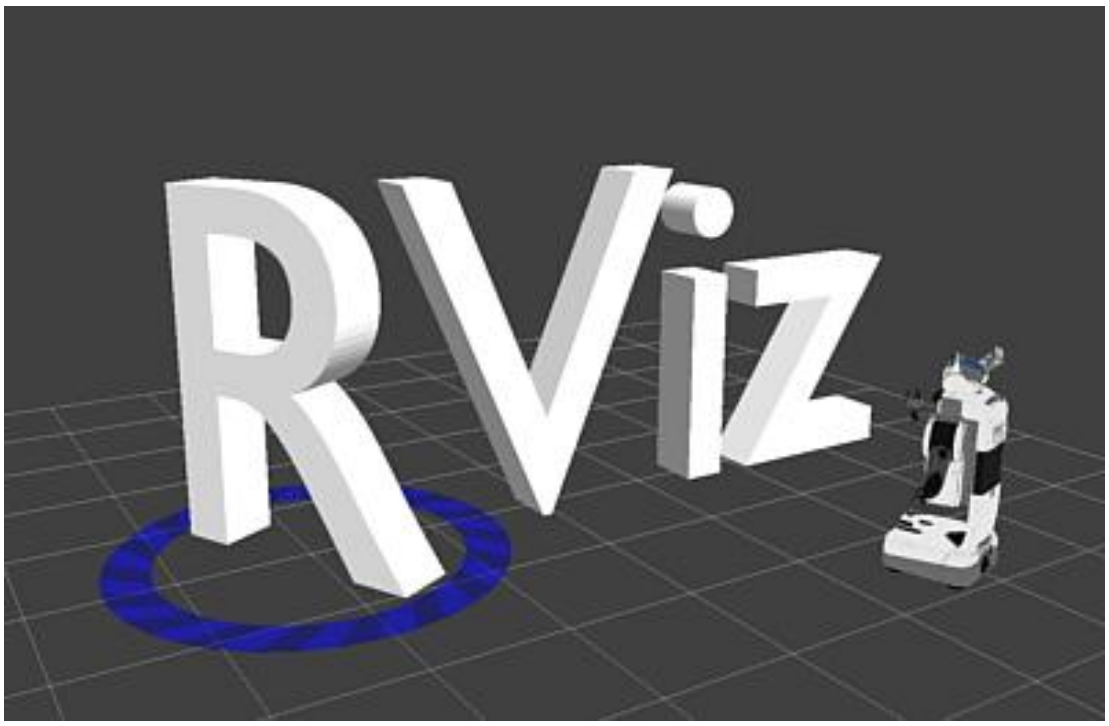


FIGURE 90

4.14 Rviz



RViz (short for "ROS Visualization") is a 3D visualization tool used in the Robot Operating System (ROS). It's designed to visualize and debug data generated by robots, such as sensor data, camera images, point clouds, maps, and robot states, in real time.

4.14.1 Supported Data Types:

RViz provides a graphical interface where you can display a variety of data types like:

- 3D models of robots
- Lidar and point cloud data
- Robot trajectories and paths
- Occupancy grid maps
- Sensor data like laser scans and camera feeds

4.14.2 Key Features of Rviz

RViz has several key features that make it an invaluable tool for visualizing and debugging robotics data. Here are some of the main features:

1. **3D Visualization**: RViz allows you to visualize the robot and its surroundings in 3D, giving you an intuitive understanding of the robot's environment, sensor data, and movements.
2. **Multiple Data Displays**: You can display various types of data simultaneously, such as:
   - Point Clouds (from LIDAR, depth sensors)
   - Laser Scan Data
   - Robot Models (including URDFs—Unified Robot Description Format)
   - Path Planning Data (like planned and executed trajectories)
   - Camera Feeds and other sensor data
   - Occupancy Grids (for mapping and localization)
3. **Interactive Markers**: RViz allows you to add and interact with markers, such as 3D shapes, that can be used to visualize robot tasks or states. You can move, resize, and interact with these markers in the RViz interface.
4. **Coordinate Frames**: It shows the coordinate frames (TF) of the robot and other objects in the environment, which is important for understanding the position and orientation of sensors, actuators, and other components.
5. **Real-time Updates**: RViz updates visualizations in real time as new data is published, providing live feedback during robot operation, testing, or simulation.
6. **Customizable Displays**: You can configure various displays (like point clouds, robot model, grid, etc.) according to your needs and toggle them on or off for a cleaner view.
7. **Plugins**: RViz supports plugins that extend its functionality, such as adding new types of displays, tools, or custom visualizations.
8. **Visualization of ROS Topics**: RViz can subscribe to ROS topics to visualize real-time data being published. You can visualize everything from sensor readings to robot status.
9. **Robot Motion Tracking**: RViz can show the robot's motion and trajectory over time, which is great for debugging and refining path planning algorithms.
10. **Integration with ROS Ecosystem**: RViz is tightly integrated with other ROS tools and libraries, making it a core part of the development and debugging workflow in ROS-based robotic systems.

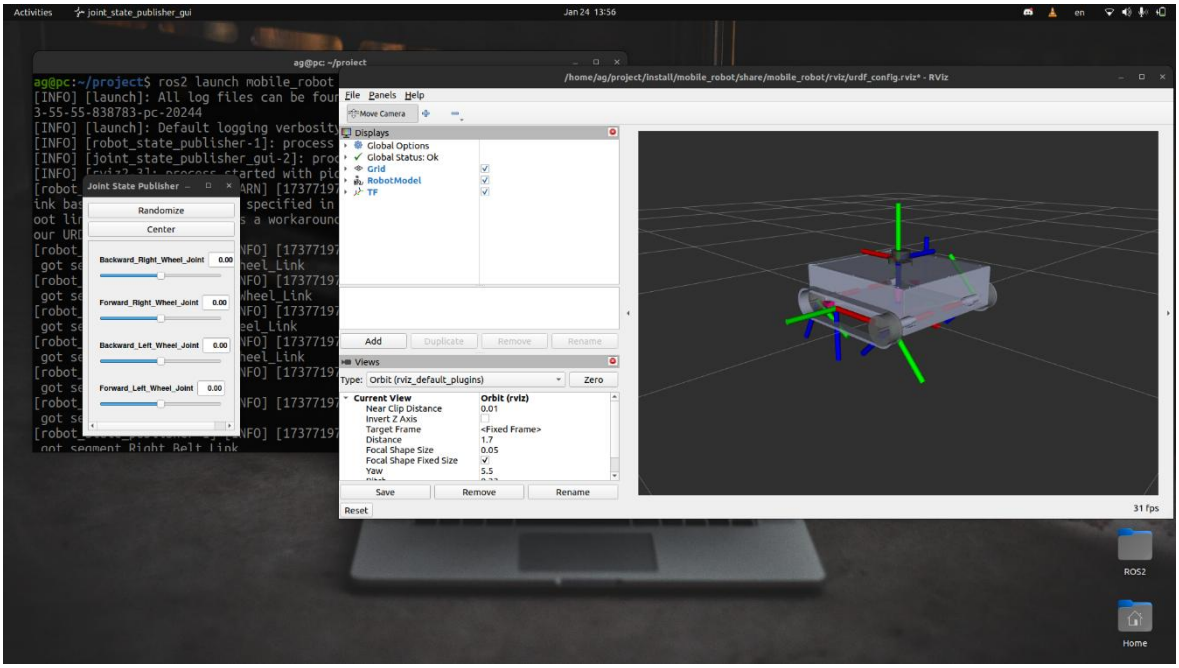


FIGURE 91 DISPLAYING THE PROJECT ON RVIZ

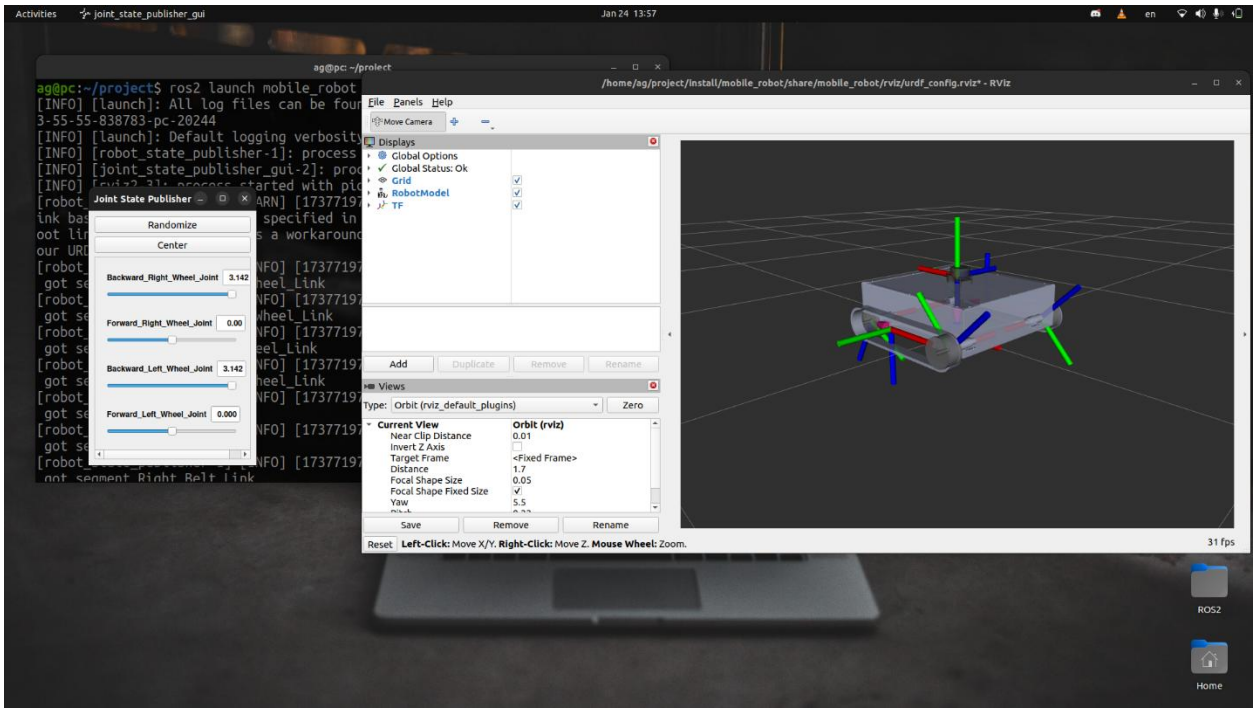


FIGURE 92 CHANGING THE JOINT STATE PUBLISHER VALUES

4.15 Gazebo

Gazebo is a powerful robotics simulation software that allows you to simulate robots in a 3D environment. It's often used in conjunction with the Robot Operating System (ROS) to test and develop robots in a virtual world before deploying them in the real world. Gazebo provides realistic physics, sensor simulation, and 3D graphics, making it a popular choice for robotics research, development, and testing.

4.15.1 Key features of Gazebo

1. **3D Simulation Environment**: Gazebo offers a 3D environment where you can place and interact with robots, objects, and sensors. You can simulate a variety of environments, including indoor and outdoor scenes with complex terrain, obstacles, and other elements.
2. **Physics Engine**: Gazebo supports multiple physics engines (e.g., ODE, Bullet, Simbody, DART) to simulate real-world physics like gravity, friction, and collision dynamics. This allows you to test robot behaviors under realistic conditions.
3. **Robot Modeling**: You can model robots using **URDF (Unified Robot Description Format)** or SDF (Simulation Description Format), which define the robot's joints, links, sensors, and other components. Gazebo simulates the robot's movement, actuators, and sensors based on these models.
4. **Sensor Simulation**: Gazebo can simulate a wide range of sensors commonly used in robotics, including cameras (RGB and depth), LIDAR, IMUs, GPS, and more. This helps in testing sensor-based algorithms without needing physical hardware.
5. **Integration with ROS**: Gazebo integrates tightly with ROS, which means you can use ROS nodes and topics to control and interact with the simulated robots. For example, you can send movement commands to the robot, read sensor data, and test ROS-based algorithms like SLAM (Simultaneous Localization and Mapping).
6. **Plugins and Extensions**: Gazebo supports plugins that allow users to extend its functionality. You can write custom plugins to control specific robot behaviors, integrate new sensors, or modify the physics engine. This flexibility makes it a powerful tool for custom simulations.
7. **Multirobot Simulation**: Gazebo supports simulating multiple robots in the same environment, making it great for testing systems involving multiple autonomous agents (like fleet management or robot collaboration).
8. **Real-time and Offline Simulation**: Gazebo supports both real-time simulations (where the virtual world runs at the same speed as the physical world) and offline simulations (where you can simulate events more quickly than real time).

9. **Visualization and Debugging**: Gazebo provides real-time visualizations of robot status, sensor data, and the environment, making it easier to debug and test algorithms, especially in complex scenarios.
10. **Community and Documentation**: Gazebo has an active community and extensive documentation, making it easier for developers to get started, find support, and use the tool effectively.

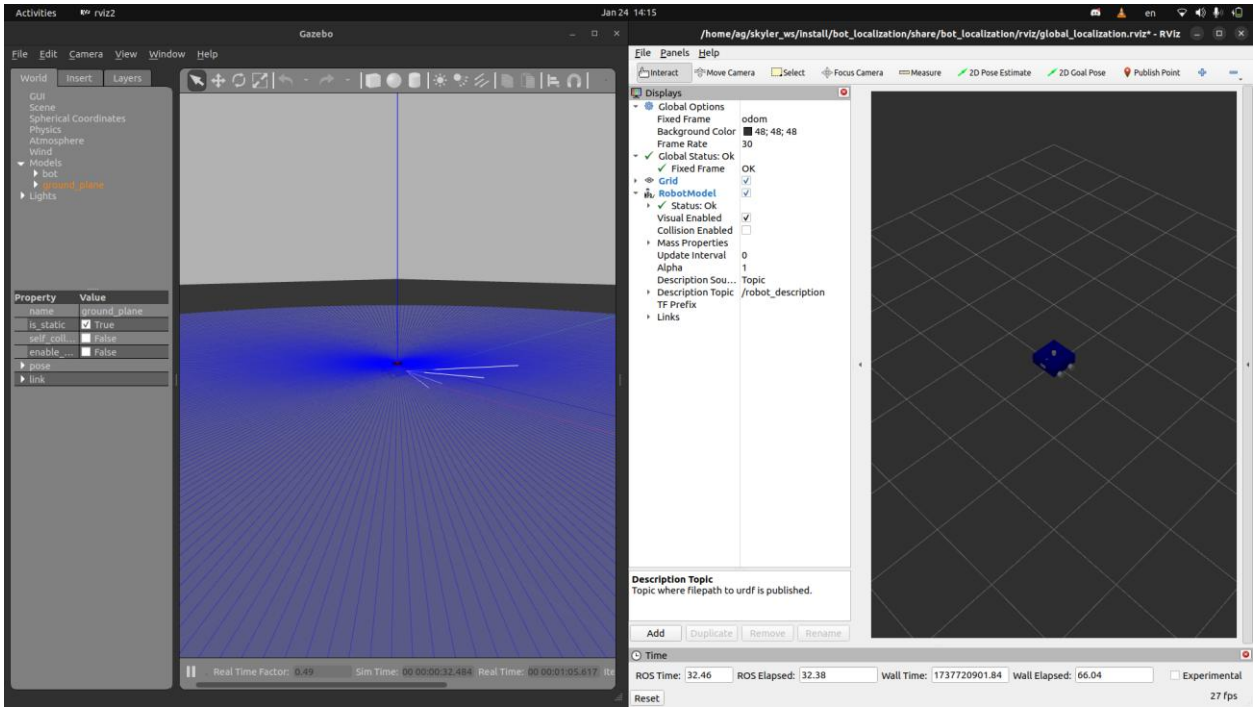


FIGURE 93 DISPLAYING THE PROJECT ON GAZEBO AND RVIZ

4.16 URDF

URDF stands for **Unified Robot Description Format**, and it's an XML format used to define the structure of a robot in the Robot Operating System (ROS). URDF is crucial for describing the physical properties and configuration of robots, such as the arrangement of links (rigid bodies), joints (connections between links), sensors, actuators, and other components.

4.16.1 Key Components of a URDF File:

1. **Links**: These represent the rigid bodies of the robot. A link can be thought of as a part of the robot, such as a wheel, arm, or sensor. Each link can have properties like mass, inertia, and visual appearance.
2. **Joints**: Joints define the connections between two links and allow movement between them. There are several types of joints (e.g., fixed, revolute, prismatic), each allowing different kinds of motion, such as rotating or sliding.
3. **Visual**: This defines how a link looks in a visualization tool (like RViz or Gazebo). You can specify meshes, colors, textures, or simple geometric shapes (e.g., cubes, spheres) to represent the link's appearance.
4. **Collision**: This defines the shape used for collision detection. It's often simplified to reduce computational load during simulation or path planning. For example, a wheel might be represented as a cylinder for collision detection purposes.
5. **Inertia**: Inertia properties define how a link resists rotational motion about its center of mass. This is important for simulating physical behavior, such as how the robot moves and reacts to forces.
6. **Sensors and Actuators**: URDF can define various types of sensors (e.g., cameras, LIDAR, IMUs) and actuators (e.g., motors, servos) attached to different parts of the robot.
7. **Transmission**: This defines how forces from motors are transmitted to the robot's joints and links. It describes the mechanical interfaces for actuating the robot.
8. **Robot Definition**: The entire robot is defined in a URDF file, starting from a root link and building out the structure by attaching other links and joints. This includes everything from the robot's body to the wheels, arms, and sensors.

4.16.2 Why URDF is Important:

- **Simulation**: URDF is widely used in simulation tools like **Gazebo** to model robots. It defines the robot's physical structure, which is used by simulators to compute accurate physical interactions, such as collisions, movement, and sensor behavior.
- **Visualization**: URDF files are used in **RViz** to display the robot's model in 3D. This helps developers visualize the robot in a virtual environment to verify its design, movements, and sensor setups.
- **Control**: ROS nodes can read URDF files to understand the robot's kinematic and dynamic structure, helping to control the robot and plan motions.

4.16.3 Creating a URDF file from SOLIDWORKS

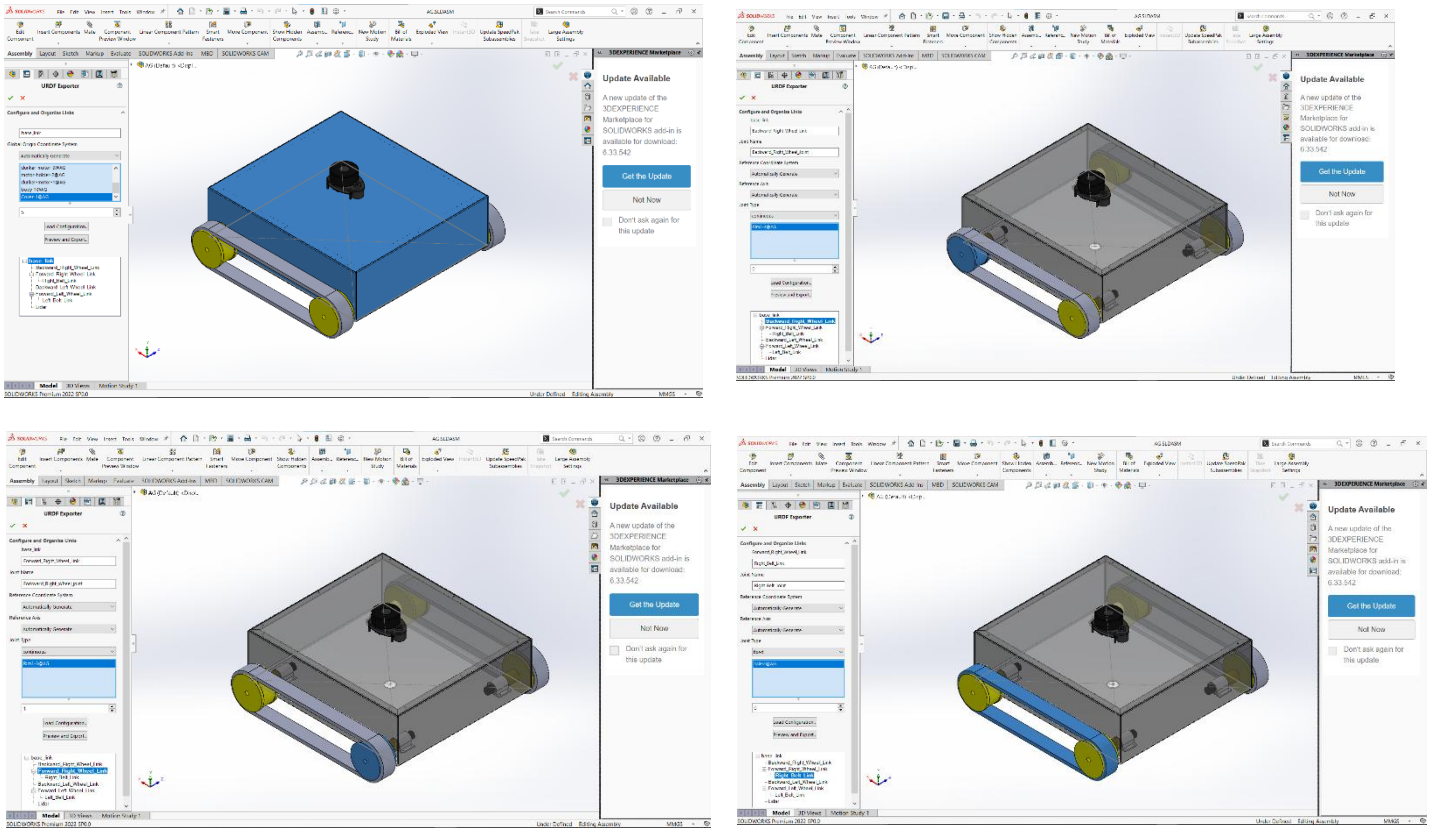


FIGURE 94

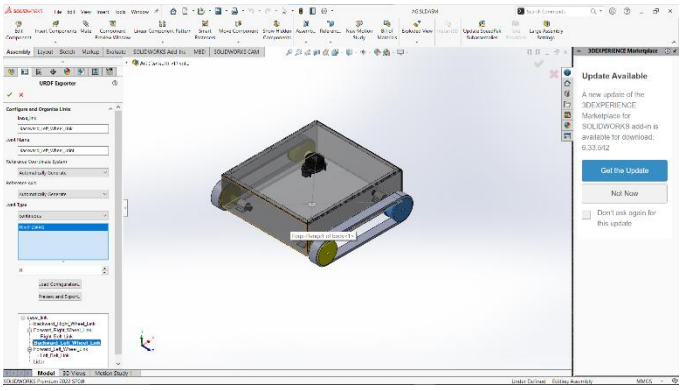
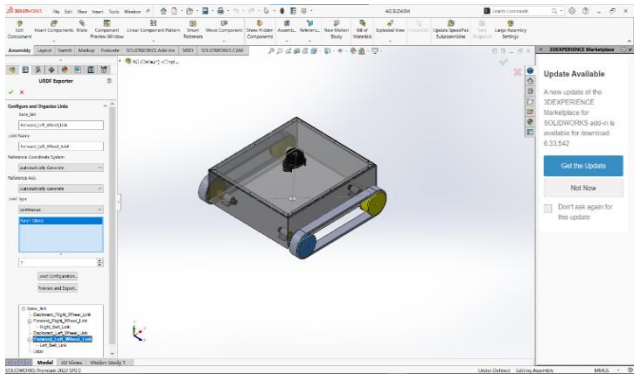
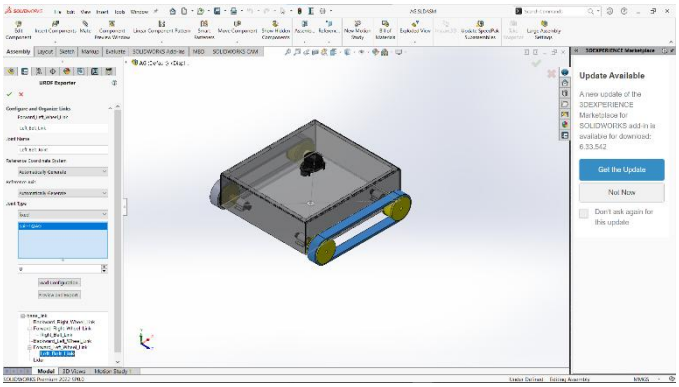
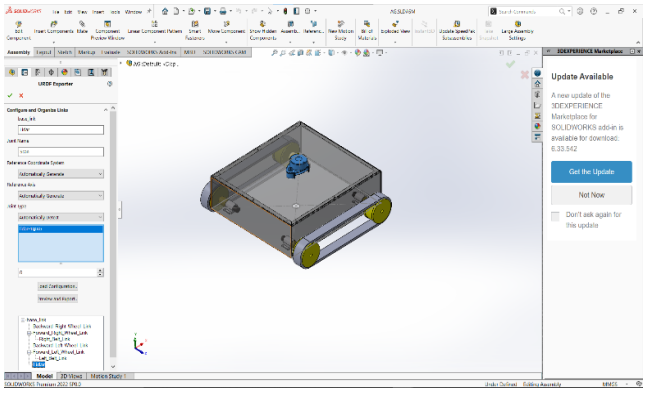
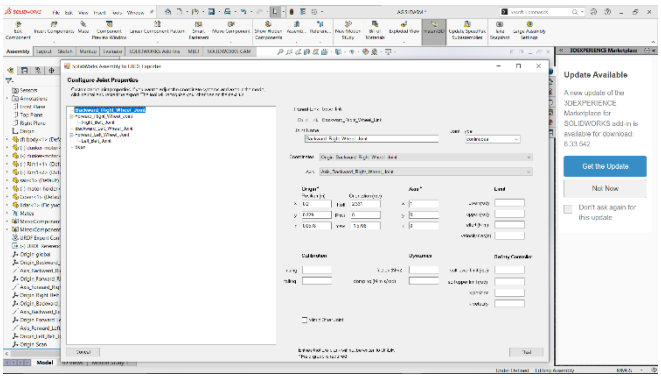
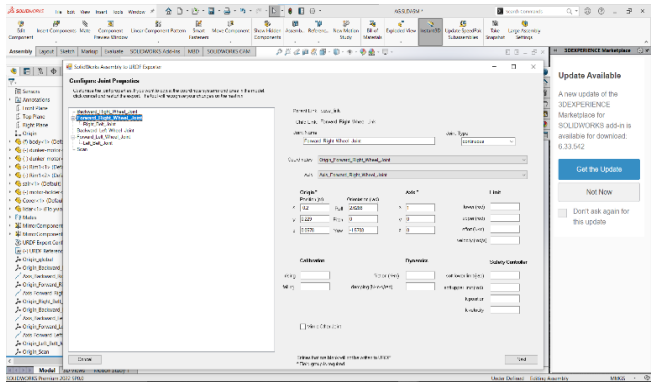
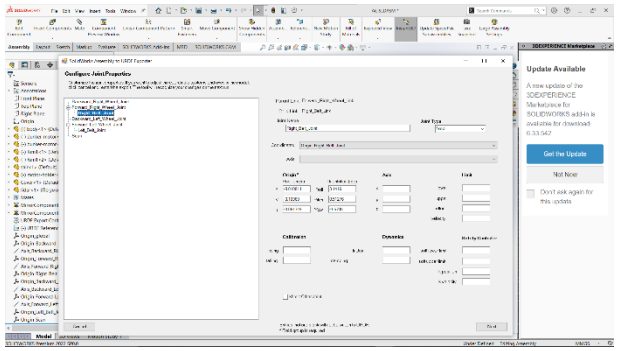








FIGURE 95

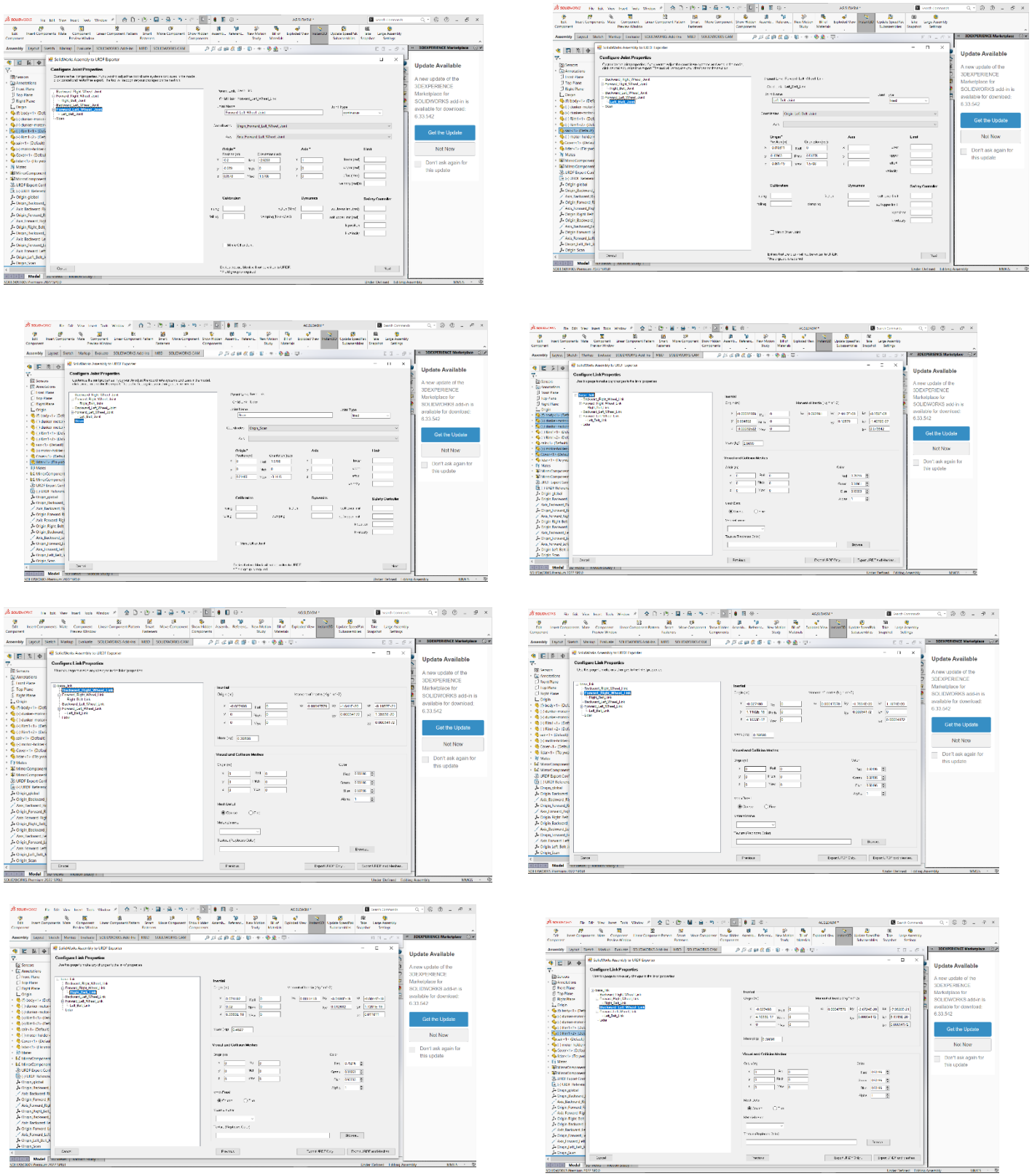


FIGURE 96

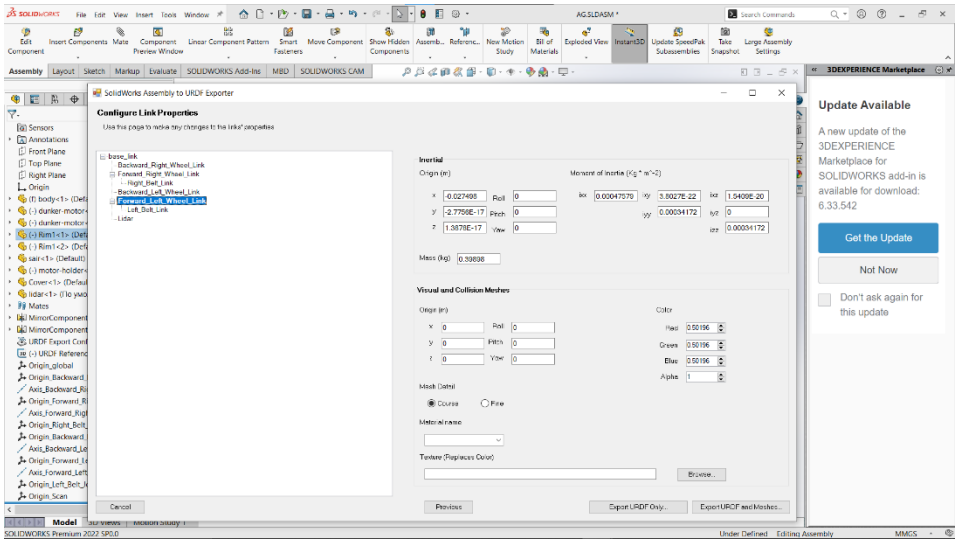


FIGURE 97

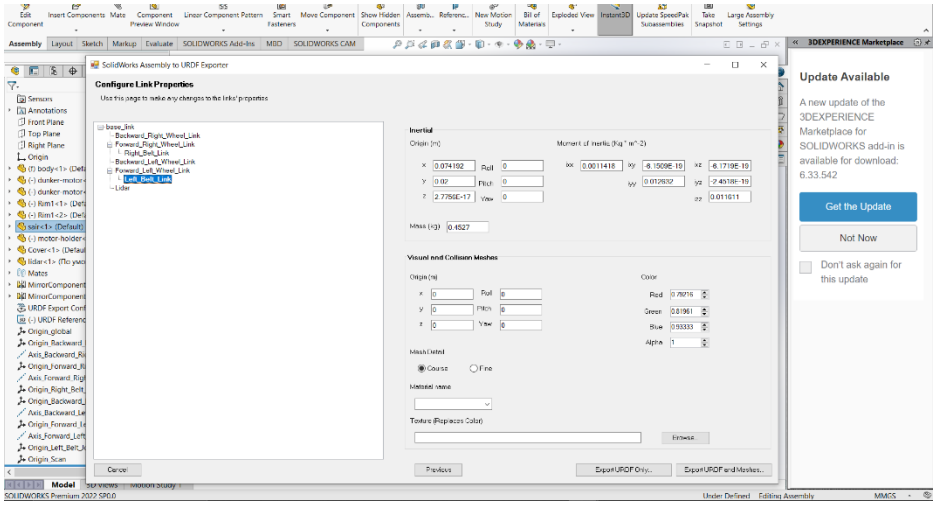


FIGURE 98

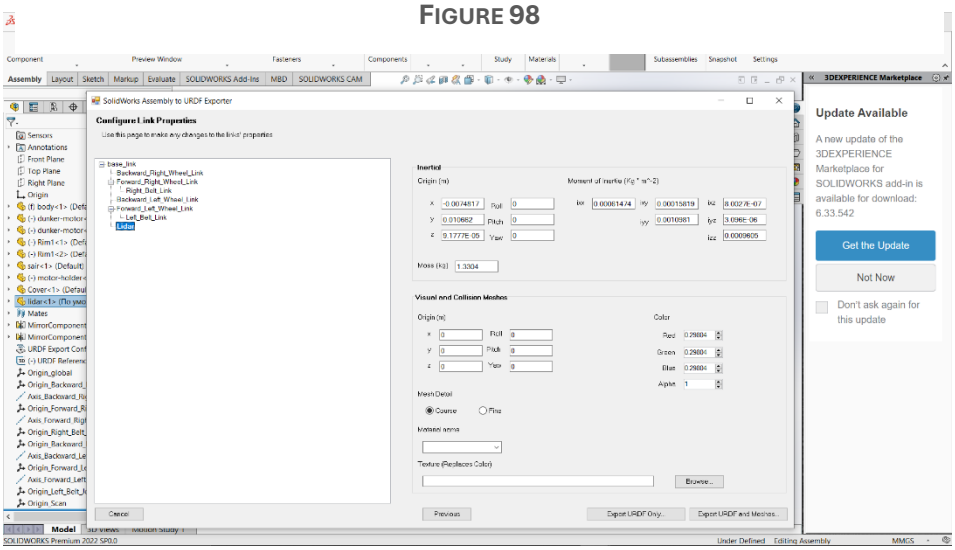


FIGURE 99

4.17 YAML

YAML is commonly used as a format for storing and managing robot parameters, configuration files, and system settings. YAML files allow you to define various settings in a human-readable format, making it easy to configure and modify robot behaviors, sensors, controllers, and other system components. The parameters stored in YAML files are then loaded into the ROS parameter server at runtime, where they can be accessed by different nodes.

4.17.1 Common Use Cases of YAML in ROS:

1. **Robot Configuration (URDF)**: YAML is used to configure robot settings, such as sensor information, actuator parameters, or environment-specific details, often alongside URDF (Unified Robot Description Format) files. For instance, the robot's sensor and actuator parameters could be loaded from a YAML file to define how the robot should behave in each environment.
2. **ROS Parameter Server**: YAML is a standard format for configuring parameters on the ROS parameter server. These parameters might include things like controller gains, sensor calibration settings, robot dimensions, and more. The parameter server allows nodes to access and modify these parameters dynamically at runtime.
3. **Launch Files**: ROS launch files are XML-based files used to start up multiple nodes, set parameters, and configure system behaviors. Often, YAML is used in conjunction with launch files to load configuration parameters for nodes, especially when there are many parameters to configure. This allows for easy management of settings without hardcoding them in launch files.
4. **Tuning and Calibration**: YAML files can be used for storing robot-specific calibration or tuning parameters, such as the sensor noise levels, camera calibration matrices, or PID controller values, which may vary depending on the robot's hardware.

4.17.2 How YAML Is Used in ROS:

1. **Loading Parameters**: In ROS, YAML files are often loaded into the parameter server through **launch files**. A launch file can reference a YAML file to load parameters into the ROS system. For example, this can be done using the 'rosparam' tag
2. **Accessing Parameters in ROS Nodes**: After the YAML file is loaded into the parameter server, ROS nodes can access the parameters via the ROS API (e.g., 'rosparam' or 'ros::param' in C++). Here's an example of how a node might retrieve parameters in Python
3. **Parameter Tuning and Adjustments**: Because YAML files are human-readable, they make it easy to tweak and adjust parameters for testing, simulation, or deployment. For example, you can quickly adjust the PID gains for a controller, or change the robot's sensor configurations, without needing to modify the code.

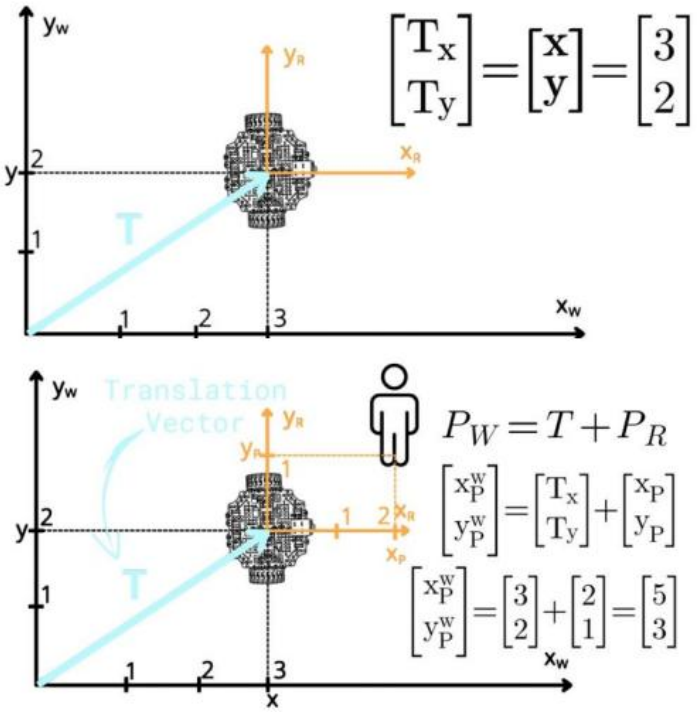
4.17.3 Advantages of Using YAML in ROS:

- **Human-Readable**: YAML is easy to read and write, which simplifies the process of configuring and tuning parameters for robotic systems.
- **Modular and Flexible**: You can organize parameters into separate YAML files for different robots or subsystems (e.g., one for sensors, another for controllers), and then load them as needed.
- **Separation of Configuration and Code**: By using YAML files, robot configuration is separated from the actual code, making it easier to adjust settings without touching the code itself.
- **Compatibility**: YAML is supported by many tools and libraries in ROS, making it a standard choice for configuration and data exchange.

4.18 Translation Matrix

In robotics, we often need to pinpoint a robot's location.

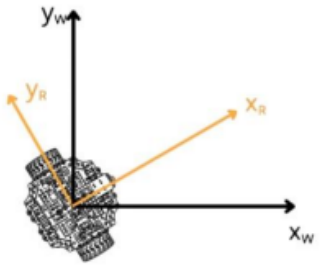
- We focus on translational motion in a 2D space, ignoring rotation.
- We use a vector called "T" to connect the robot's frame (R) to a fixed reference frame (W).
- T has two parts, X and Y, representing horizontal and vertical positions.
- To find the robot's global position, we add its position in R to T.
- This concept applies to sensor data; if a sensor detects an obstacle relative to the robot, we use T to calculate its global position.
- The translation vector, T, simplifies robot localization and helps convert sensor data into global coordinates.



4.19 Rotation Vector

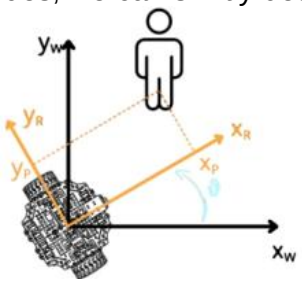
We simplify the task of defining an object's orientation in a two-dimensional world. We focus on determining how a robot's reference frame (R) aligns with the world frame (W), with the assumption of zero translation between them. Orientation is quantified by the angle theta, representing the rotation between these frames, which we calculate using trigonometry.

To apply this concept practically, consider an obstacle (P) in the world with coordinates p_x and p_y. We introduce a rotated frame, sharing its origin with W but offset by theta. By projecting P's coordinates onto this rotated frame, we obtain p_x_first and p_y_first. The challenge is to find P's world coordinates, given p_x_first, p_y_first, and theta.

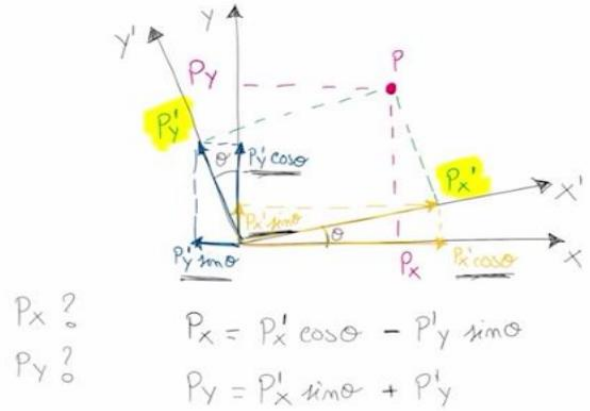


Leveraging trigonometry, we express p_x_first and p_y_first in the world frame. The x-coordinate becomes p_x_first * cos(theta), while the y-coordinate is p_x_first * sin(theta).

Similarly, the y-coordinate is p_y_first * cos(theta), and the x-coordinate is p_y_first * sin(theta). These equations can be represented in matrix form using a rotation matrix that depends on theta, establishing the link between P's coordinates in both frames. In practical applications, such as a robot detecting an obstacle in its reference frame (R), which is rotated relative to the world frame (W), we have access to P's coordinates (X_P and Y_P) in R and knowledge of theta. Utilizing the matrix equation with real values, we can swiftly deduce P's world coordinates. This approach simplifies orientation transformations in a two-dimensional space.



$$\begin{bmatrix} x_P^W \\ y_P^W \end{bmatrix} = \begin{bmatrix} \cos(\vartheta) & -\sin(\vartheta) \\ \sin(\vartheta) & \cos(\vartheta) \end{bmatrix} \begin{bmatrix} x_P \\ y_P \end{bmatrix}$$



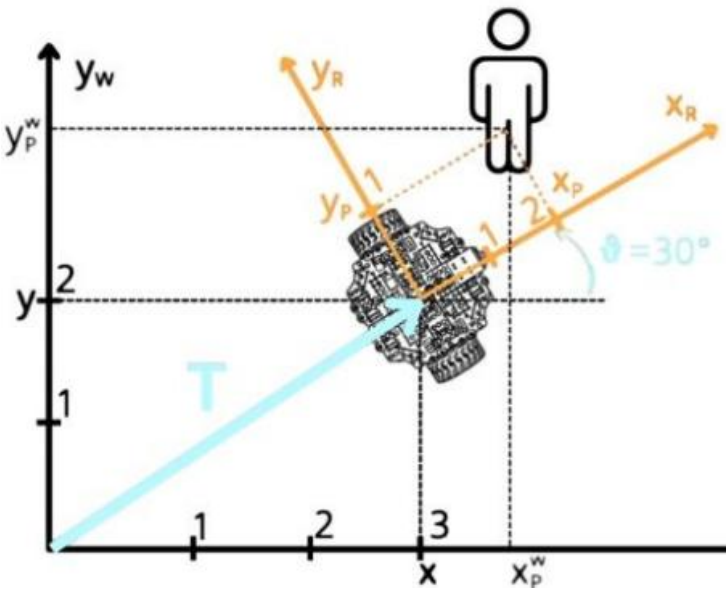
4.20 Transformation Vector

To understand the motion of a robot in two-dimensional space, we combine concepts of translation and rotation. Imagine a robot moving within a reference frame R, which itself can rotate and translate relative to a fixed world reference frame W. The position of R relative to W is defined by a translation vector T and an orientation angle θ.

Suppose this robot is equipped with sensors that detect obstacles, providing their positions in the robot's frame R. Our goal is to translate these positions into the world frame W to understand where these obstacles are in a broader context.

By integrating the rotation (represented by angle θ) and translation (represented by vector T) equations, we construct a transformation matrix. This matrix, often represented in homogeneous form for mathematical convenience, encapsulates the complete motion (rotation and translation) of the robot.

In practice, given an obstacle's coordinates in the robot frame and knowing the robot's orientation θ and translation T relative to the world frame, we can apply the transformation matrix. This lets us calculate the obstacle's position in the world frame, effectively mapping the robot's local perception to a global context. This method is critical for robotic navigation and understanding, as it allows for the precise determination of objects' positions in a broader environment based on localized data.



$$\begin{bmatrix} x_P^w \\ y_P^w \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(30) & -\sin(30) & 3 \\ \sin(30) & \cos(30) & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$$

$$x_P^w = \cos(30)\cdot 2 - \sin(30)\cdot 1 + 3\cdot 1$$
$$y_P^w = \sin(30)\cdot 2 + \cos(30)\cdot 1 + 2\cdot 1$$

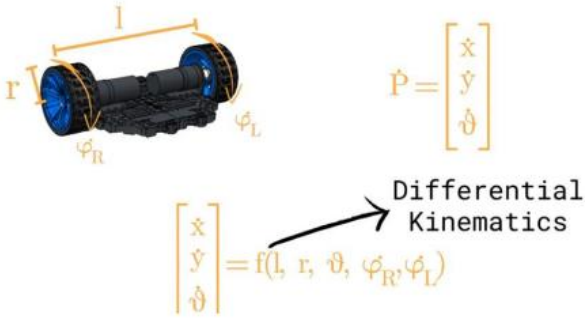
$$x_P^w = 1.73 - 0.5 + 3 = 4.23$$
$$y_P^w = 1 + 0.87 + 2 = 3.87$$

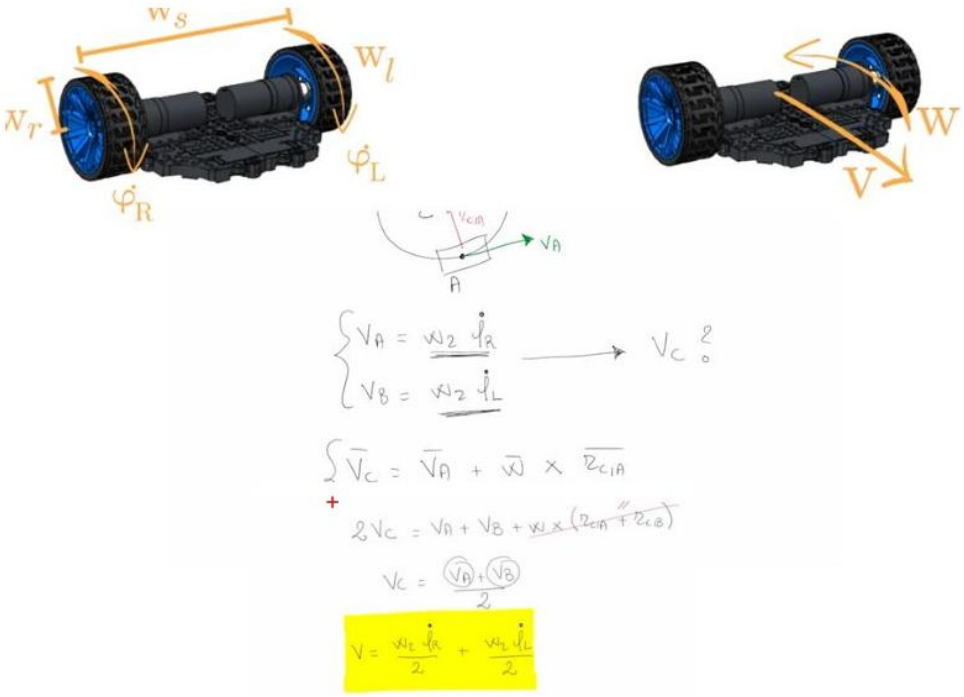
4.21 Differential Kinematics

To understand a robot's velocity in two dimensions, we analyze its pose, defined by its position (X, Y) and orientation (theta). Velocity, a function of time, is the change in the robot's position and orientation over intervals.

Consider the robot at an initial time (t0) with pose P0. At a later time (t1), it moves to pose P1. The robot's velocity consists of linear components (X-dot, Y-dot) in the plane and an angular component (Theta-dot).

In differential kinematics, this concept extends to a two-wheeled robot. The wheels, with radius R and spaced L apart, move at different velocities (V-dot-R for the right, V-dot-L for the left). The challenge is to formulate a function that relates these parameters to the robot's overall velocity in two dimensions, combining both linear and angular motion. This function is key for controlling the robot's movement, such as through a joystick, translating individual wheel speeds into coordinated spatial movement.



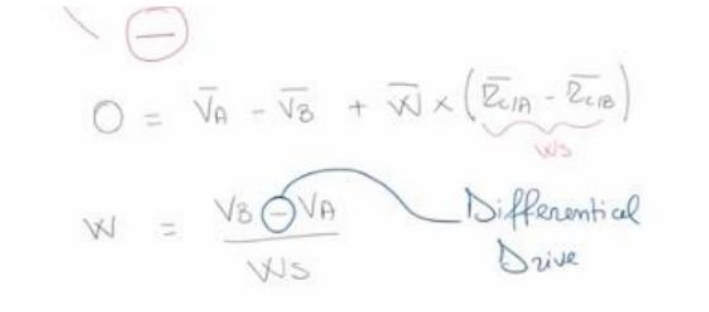


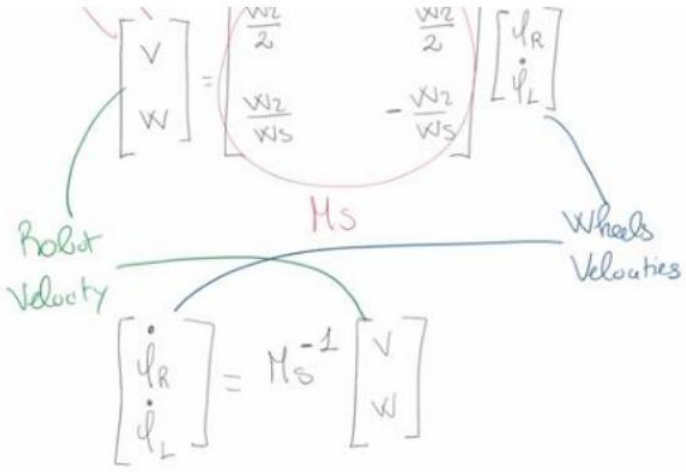
Linear Velocity (V): The linear velocity of the robot is derived by adding the velocities of two reference points (A and B) on the robot.

Angular Velocity (W): The angular velocity of the robot is calculated by subtracting the velocities of points A and B and then dividing by the distance between these points (wheel separation WS). This is based on the differential drive architecture of the robot.

Matrix Equation (M.S): The relationship between the robot's linear and angular velocities and the rotational velocities of its wheels can be expressed in a matrix form. This matrix includes components like wheel radius (WR), wheel separation (WS), and the rotational velocities of the left (φ_dot_l) and right (φ_dot_r) wheels.

Inverse Relationship: By inverting the matrix M.S, it's possible to calculate the rotational speeds of the wheels based on the robot's linear and angular velocities. This is crucial for controlling the robot, for instance, using a joystick to convert command velocities into wheel movements.





4.22 Robot's Velocity

The robot’s velocity when its reference frame to the global reference frame:

Robot Motion on a Plane: Consider a robot moving on an XY plane with linear velocity V and angular velocity W. The robot's position is defined by coordinates X and Y, and its orientation is defined by an angle θ.

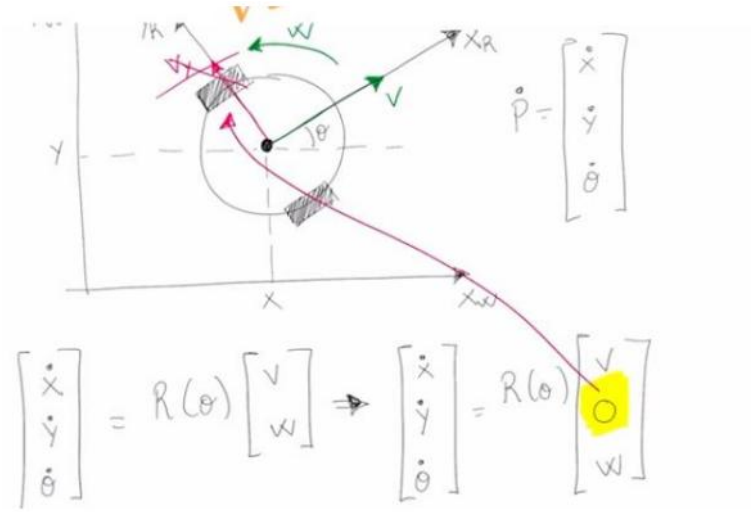
Velocity Vector (Ṗ): The robot's velocity in the plane is represented by a vector Ṗ, which includes the variation of its position along the X and Y axes and the variation of its orientation θ.

Connecting Velocities: The task is to link the robot's linear (V) and angular (W) velocities in the robot's reference frame to its velocity (P) in the global reference frame.

Rotation Matrix and Translation Vectors: This connection is established using rotation matrices and translation vectors. The robot's velocity in the global reference frame is represented by a matrix R (dependent on θ) multiplied by a vector containing the robot's velocities (V and W).

Differential Drive Constraint: Considering the robot's differential drive architecture, it can't move instantaneously along the Y-axis but only along the X-axis. Therefore, the Ycomponent of the linear velocity is set to zero in the equations.

Equation in World's Reference Frame: The final equation representing the robot's velocity in the world's reference frame incorporates the rotation matrix (dependent on θ) and the robot's velocity vector (with the Y-component set to zero). This matrix equation is:



$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = R(\theta)\begin{bmatrix}V\\ W\end{bmatrix} \Rightarrow \begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = R(\theta)\begin{bmatrix}V\\ 0\\ W\end{bmatrix}$$

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}V\\ 0\\ W\end{bmatrix}$$

First Equation (Robot's Reference Frame to World's Reference Frame): This equation uses a rotation matrix dependent on the robot's orientation (θ) to connect the velocity in the robot's reference frame to the velocity in the world reference frame.

Second Equation (Robot's Linear and Angular Velocities to Wheel Rotational Velocities): This equation links the robot's linear and angular velocities to the rotational velocities of its wheels, using a matrix dependent on the wheel radius (WR) and their separation (WS).

Overall Differential Kinematics Equation (Forward Differential Kinematic Equation): By multiplying these two matrices, we derive the equation connecting the robot's overall velocity in the world reference frame to the rotational velocity of its wheels.

The combined equation is formed as follows:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \frac{WR}{2} & \frac{WR}{2} \\ 0 & 0 \\ \frac{WR}{WS} & -\frac{WR}{WS} \end{bmatrix} \times \begin{bmatrix} \dot{\phi}_L \\ \dot{\phi}_R \end{bmatrix}$$

This product results in a matrix that expresses the overall velocity of the robot in the world reference frame as a function of the robot's wheel radius, wheel separation, orientation, and rotational velocity of its wheels.

Jacobian Matrix: The resulting matrix, which relates the robot's wheel velocities to its overall velocity in the world frame, is known as the Jacobian.

4.23 Python-based ROS2 controller node

The controller is designed to:

- Receive TwistStamped messages indicating the desired linear and angular velocities for a differential drive robot.
- Translate these velocities into individual wheel commands using the robot's wheel radius and wheel separation parameters for differential kinematics.
- Publish the wheel commands as Float64MultiArray messages to specific topics that control the robot's wheel actuators.

Key components of the controller include:

- Initialization of the node with parameters for the wheel radius and separation, essential for calculating wheel velocities.
- A publisher for sending out the wheel speed commands to the robot.
- A subscriber listening for velocity commands, typically from a joystick or another control interface.
- A callback function that uses a conversion matrix to translate the robot's velocity into wheel speeds.
- A main function to initialize and run the node, ensuring it continuously listens for commands and responds appropriately.

This node effectively bridges the gap between high-level velocity commands and low-level wheel actuations, allowing for the smooth operation of the robot within its environment.

4.24 TF2 Library

TF2 is a fundamental robotics library in ROS, enabling the management of reference frames and the calculation of transformation matrices. It's essential for determining a robot's position relative to its initial pose (frame O) and the global map (frame M). This library plays a crucial role in various robotic applications, including the development of odometry systems. Transformation Matrices in Robotics In robotics, understanding the relative position and orientation of components is achieved through transformation matrices. These matrices represent the spatial relationship between various reference frames assigned to robot parts, enabling the calculation of movement and changes in orientation over time.

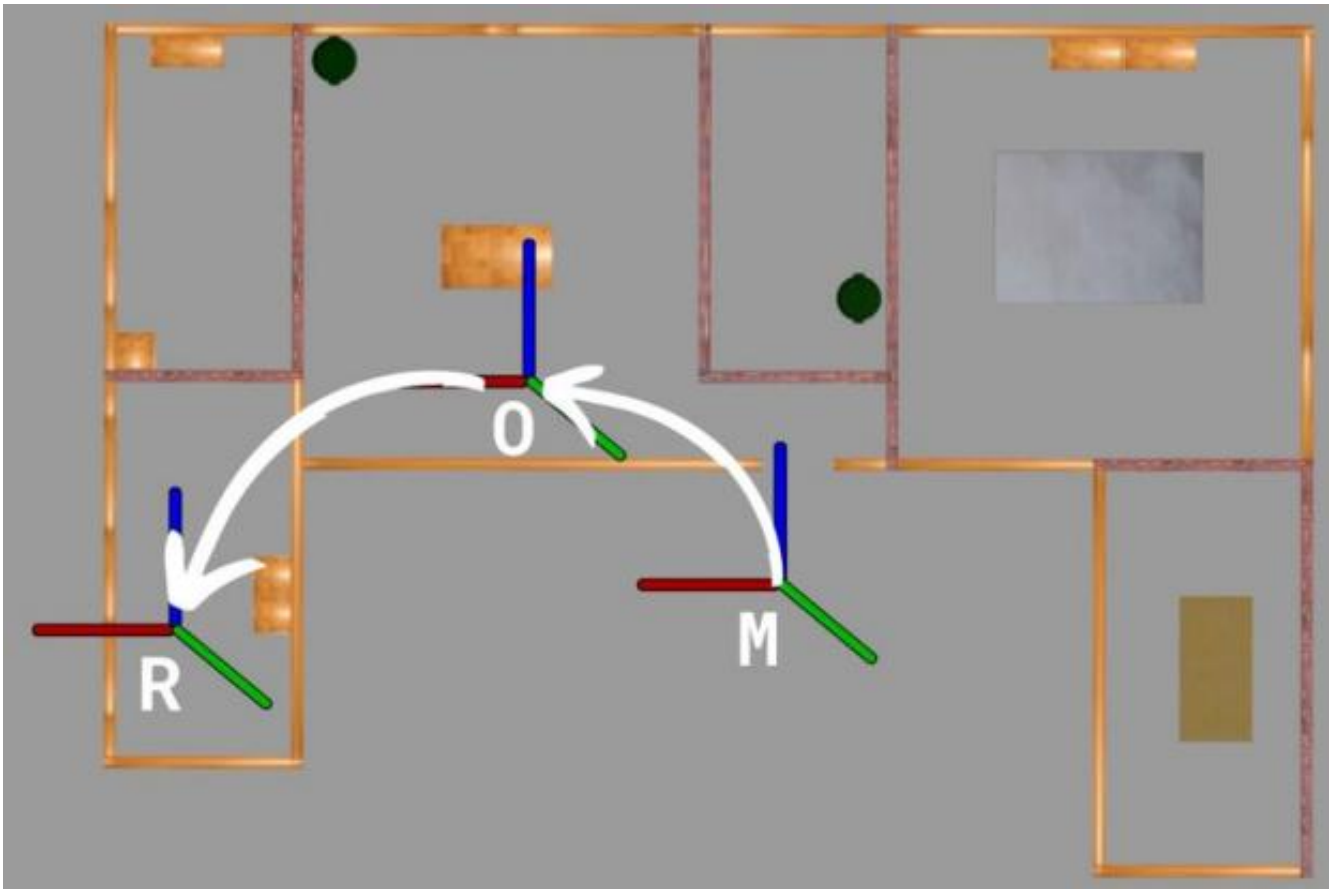


FIGURE 100

Transformation Matrices in Robotics

In robotics, understanding the relative position and orientation of components is achieved through transformation matrices. These matrices represent the spatial relationship between various reference frames assigned to robot parts, enabling the calculation of movement and changes in orientation over time.

A practical example involves a robot with a camera that detects a person. The person is assigned a new reference frame, and the application provides the transformation matrix relative to the camera's frame. The robot's URDF model defines the reference frames and their connections, including the camera's link to the base footprint.

Calculating the person's position relative to the robot requires multiplying the intermediate transformation matrices: first from the person to the camera, and then from the camera to the robot's base footprint. This sequence is crucial, as the order of multiplication affects the outcome, illustrated by the difference in the final orientation of a parallelepiped when rotation orders are varied.

Understanding these relationships is crucial for tasks such as localization and navigation, where the robot must be aware of its position and that of other objects in the environment. The multiplication of transformation matrices in the correct sequence allows the robot to map detected objects onto its own reference frame and onto a global map frame, providing a comprehensive spatial awareness necessary for complex robotic operations.

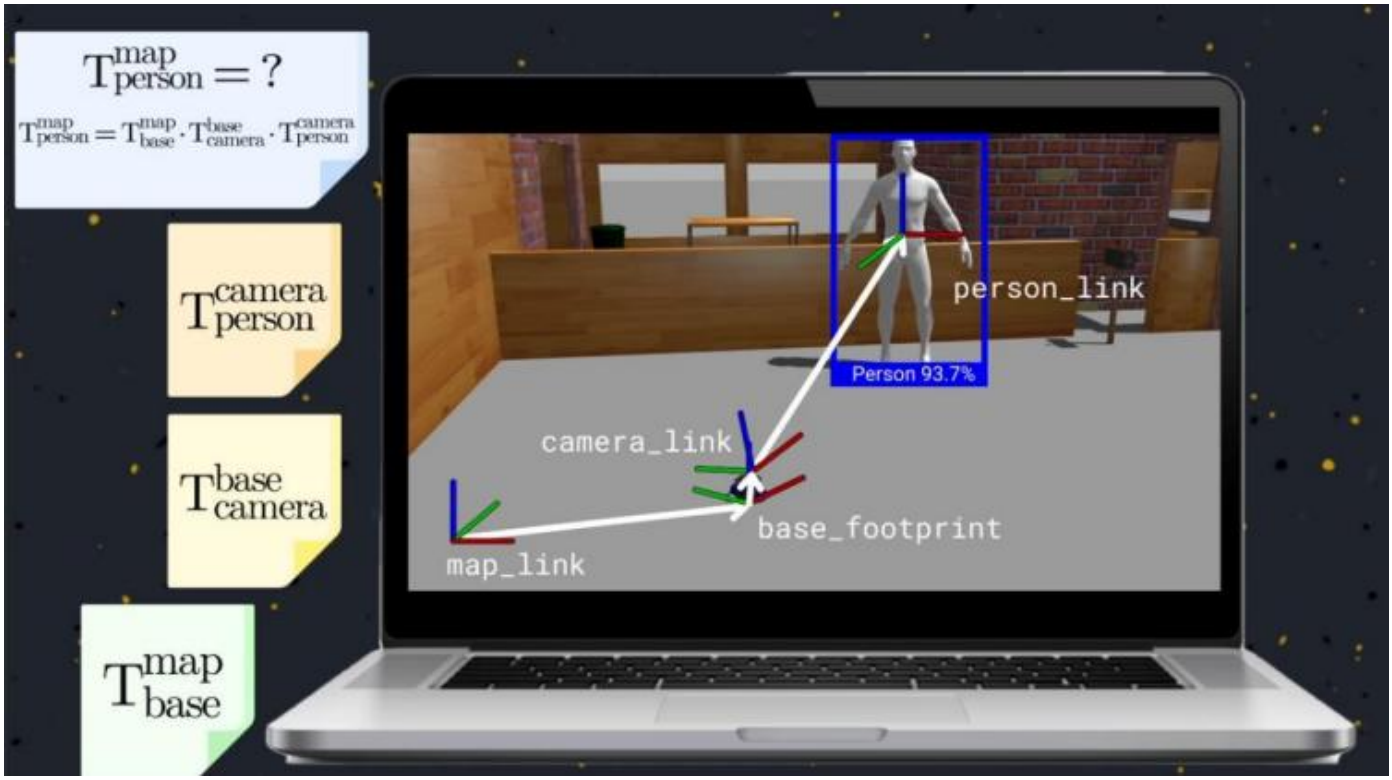


FIGURE 101

4.25 Angle Representations

Rotation matrices and quaternions are two ways to represent the orientation of a reference frame in space. Quaternions are preferred in computer science because they are more efficient to compute and they avoid gimbal lock. Quaternions are used in robotics, video game development, computer vision, animation, and special effects.

4.25.1 Euler

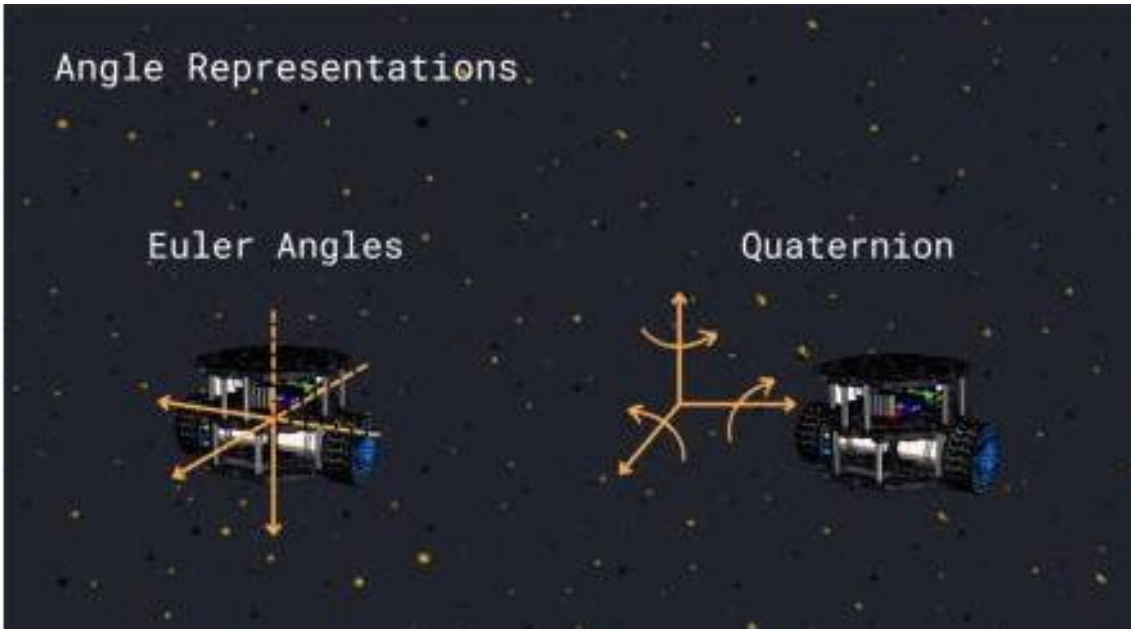
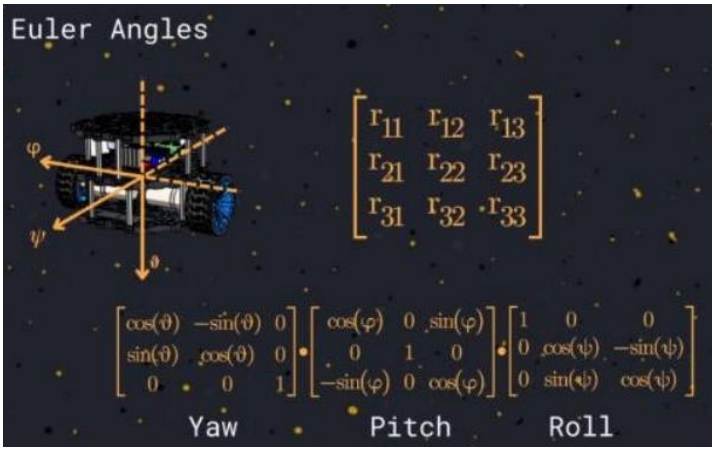


FIGURE 102 EULER

In the context of orientation in three-dimensional space, "Roll, pitch, an ”yaw" are fundamental components. These terms describe the rotations around the X, Y, and Z axes, respectively. When expressing the orientation of an object using Euler angles, we rely on the composition of three elementary rotation matrices corresponding to these rotations. By multiplying these matrices, we derive a new matrix that depends solely on these three angles. In summary, when using Euler angles to represent the rotation of an object in three-dimensional space, a total of nine components are needed to fully describe the orientation.



4.25.2 Quaternions

Quaternions are a mathematical way to represent object orientation in 3D space. They consist of four components: A, B, C, and D. A is the scalar part, and B, C, and D make up the vector part. The advantages of quaternions include simplicity – they need only four components, while Euler angles require nine. Quaternions are also unitary, making calculations and combining rotations more efficient. Multiplying two quaternions involves 16 products and 9 additions, whereas Euler angles require 27 multiplications and 18 sums.

Calculating a quaternion's inverse is straightforward; it flips the sign of the vector part and represents a reverse rotation. This is easier than finding the inverse of a rotation matrix through transposition, which is more complex. In computer science, quaternions are commonly used due to their efficiency, but for humans, Euler angles and matrices are more intuitive. In our upcoming lessons, we'll learn tools to convert between these representations for practical use in Ros2.

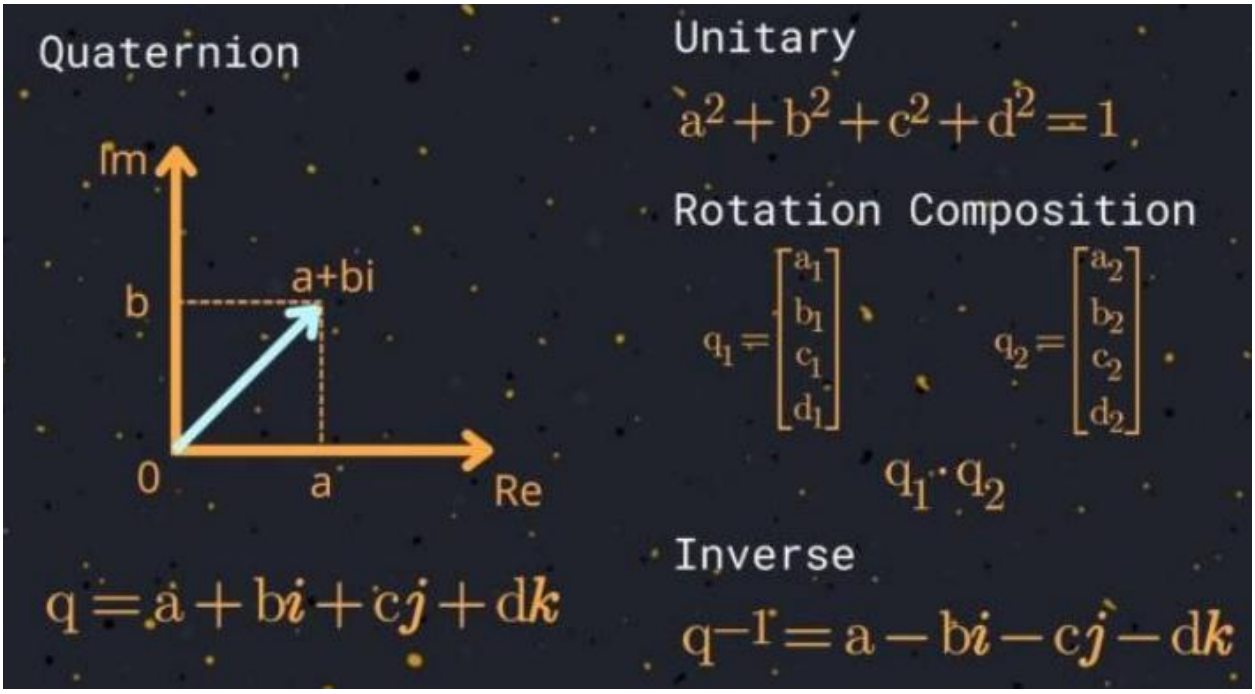


FIGURE 103 QUATERNIONS

4.26 Odometry in ROS2

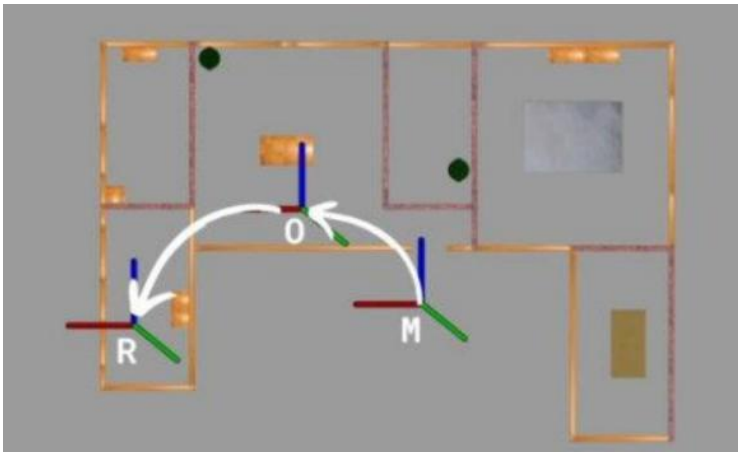
Odometry in ROS 2 (Robot Operating System 2) is a mechanism used to estimate the position and velocity of a robot by integrating sensor data. It provides information about the robot's movement and is important for tasks such as localization, mapping, and navigation.

In ROS 2, odometry is typically calculated using sensor data from wheel encoders, which measure the rotation of the robot's wheels.

The encoder readings are used to estimate the distance traveled by the robot and its rotational movement.

4.26.1 Where is the robot?

- Importance of determining the robot's position in ROS2
- Accurate position information is essential for robot navigation, mapping, and localization tasks.
- Enables the robot to understand its environment and make informed decisions.
- Relevance for navigation, mapping, and localization
- **Navigation:** Knowing the robot's position allows it to plan and execute optimal paths to reach desired locations.
- **Mapping:** Accurate position estimation aids in creating precise maps of the environment.
- **Localization:** Determining the robot's position relative to a known map enables it to localize itself within the environment.



4.26.2 The Local Localization Challenge:

- Local localization refers to estimating the robot's position relative to a known map or environment.
- It involves fusing sensor data, extracting relevant features, and applying probabilistic algorithms.
- Estimating the robot's position relative to a known map or environment
- **Fusion of sensor data:** Combining information from sensors like wheel encoders, IMU, and visual odometry.
- **Feature extraction:** Identifying distinctive features in the environment to match against the known map.
- **Probabilistic algorithms:** Using techniques like particle filters or Kalman filters to estimate the robot's position based on sensor measurements.

FIGURE 104

4.26.3 Wheel Odometry

- Wheel odometry estimates the robot's displacement based on the rotation of its wheels.
- It assumes that there is no slipping or skidding of the wheels.

Estimating robot displacement based on wheel rotation

- Calculating the wheel velocities using wheel encoders.
- Integrating the wheel velocities over time to estimate the robot's change in position.
- The initial position is usually determined using external localization techniques like GPS or visual markers.

Assumptions, limitations, and implementation details in ROS2

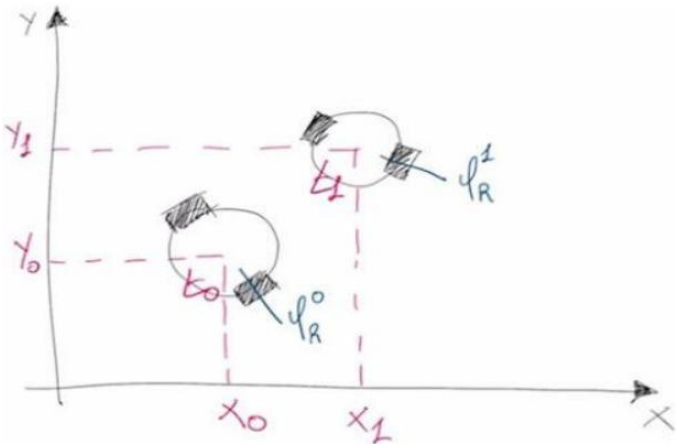
- Assumptions: No wheel slipping, no skidding, and accurate wheel encoder measurements.
- Limitations: Accumulated errors over time, sensitivity to wheel slippage, and inability to handle sudden changes in motion.
- Implementation details: ROS2 provides libraries and APIs for accessing wheel encoder data and performing the necessary calculations.

Differential Inverse Kinematics:

We know that the velocity is given by the ratio of the travel distance and the time it took to cover that distance.

$$\begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} \frac{w_2}{2} & \frac{w_2}{2} \\ \frac{w_2}{w_3} & -\frac{w_2}{w_3} \end{bmatrix} \begin{bmatrix} \dot{\varphi}_R \\ \dot{\varphi}_L \end{bmatrix} \qquad V = \frac{S}{t}$$

If we consider a robot moving in the two-dimensional space and so on an x y plane during its motion, it will find itself in a different position of this plane. If we name x zero and Y zero the coordinate of the robot when the emotion started. So at the time T zero and we call x one and y one the coordinates of the robot at the current moment in time t one. Then we can also identify a certain orientation of the wheel at each moment in time. Let's say that the right wheel at the time t zero is at position phi r zero and due to the rotation at the time t one it is in the position phi r one. This information on the current angle of each wheel is exactly the one that is provided by the encoders and that we have used to estimate the rotational velocities of each wheel.

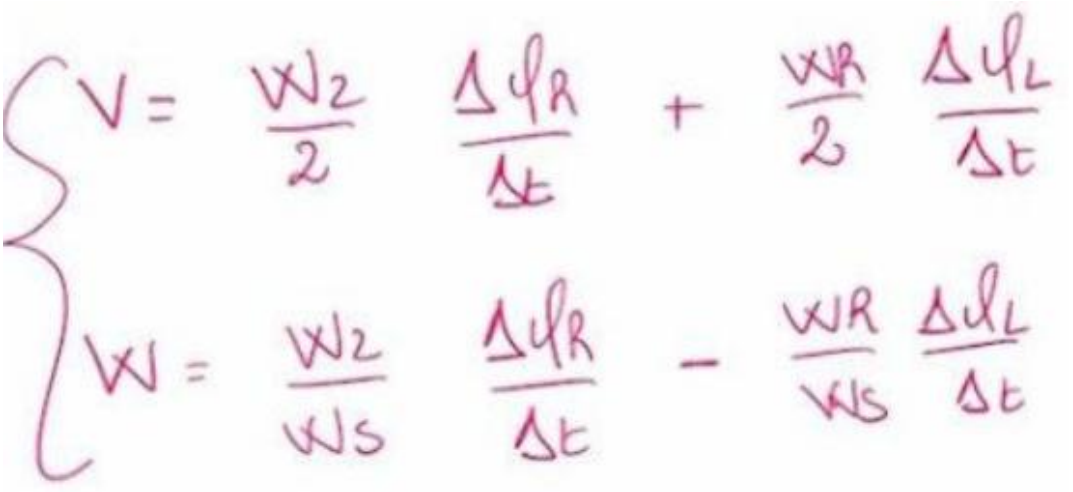


For the right wheel. The rotational velocity is given by the travel distance. And so the difference between the position of the wheel at time T one and its initial position at time t zero is divided by the time needed to perform such a rotation. And so the difference between the current time t one and the initial time t zero that we can also alternatively indicate as delta t r divided by delta t. The same way to calculate the rotational velocity of the left wheel of the robot. This is given by the difference between the position of the left wheel at time t one and its position at time t zero divided by the elapsed time to complete this movement and so divided by the difference between the time t one and the time t zero that we can also alternatively write as Delta Phi l divided by Delta t.

$$\dot{\varphi}_R = \frac{\varphi_R^1 - \varphi_R^0}{t_1 - t_0} = \frac{\Delta \varphi_R}{\Delta t}$$

$$\dot{\varphi}_L = \frac{\varphi_L^1 - \varphi_L^0}{t_1 - t_0} = \frac{\Delta \varphi_L}{\Delta t}$$

By rewriting the differential kinematic equation with these new equations that we just calculated, we can write This equation that will allow us to calculate the linear and the angular velocities of the robot. By knowing the movement of the wheels of the robot.



Implementing the Differential Inverse Kinematics equation:

We can now move on to implement the differential kinematics equation to calculate the linear and the angular velocity of the robot based on the information that we are retrieving from the encoders of the robot.

- We can do this in the same python node where we developed in the previous section, the wheel controller. And so, within the boot controller package, let's continue modifying this simple_controller.py. Let's start by adding some support variables to the simple controller class.

- Then we create a new variable called Deep Left, which will contain the variation of the position of the left wheel. And this is given by the message that we have received and let's access to position number one. And so, this is the position of the left wheel at the current moment in time, and we have to subtract the previous position. So, the value of the left wheel, previous position variable, we can do the same also for the calculation of the movement, so of the depth of the right wheel. So, we can actually copy this instruction, let's paste it and the deep right.

- After that, we can save this file and now we can actually run our controller in order to see how all of this works. So how the calculation of the linear and the angular velocity works.

- As we expect, the robot starts performing an in-place rotation. And if we look at the output of our controller. So, if we stop for a moment, the output, we can see that the angular velocity that our controller is estimating based on the information provided from the encoders.

$$POS = \int_t v\,dt \rightarrow \int_t \frac{w_2}{2}\dot{\varphi}_R + \frac{w_2}{2}\dot{\varphi}_L\,dt \rightarrow$$

$$\int_t \frac{w_2}{2}\dot{\varphi}_R\,dt + \int_t \frac{w_2}{2}\dot{\varphi}_L\,dt \rightarrow$$

$$\frac{w_2}{2}\int_t \dot{\varphi}_R\,dt + \frac{w_2}{2}\int_t \dot{\varphi}_L\,dt \rightarrow$$

$$\frac{w_2}{2}\int_0^t \frac{d\varphi_R}{dt}\,dt + \frac{w_2}{2}\int_0^t \frac{d\varphi_L}{dt}\,dt \rightarrow \quad t = 1$$

$$\frac{w_2}{2}\left(\varphi_R^1 - \varphi_R^0\right) + \frac{w_2}{2}\left(\varphi_L^1 - \varphi_L^0\right) \rightarrow$$

$$POS = \frac{w_2}{2}\Delta\varphi_R + \frac{w_2}{2}\Delta\varphi_L$$

Estimating the robot's position using wheel odometry

- Integration of wheel velocities over time for position relative to a reference frame
- Potential sources of error in position estimation
- Techniques and algorithms for estimating the robot's orientation using wheel odometry
- Calculation of heading or orientation based on wheel velocities
- Limitations and sources of error in orientation estimation

$$\text{Orientation} = \int_t w\,dt \Rightarrow \int_t \frac{w_2}{w_s}\dot{\varphi}_R - \frac{w_2}{w_s}\dot{\varphi}_L\,dt \Rightarrow$$

$$\int_t \frac{w_2}{w_s}\dot{\varphi}_R\,dt - \int \frac{w_2}{w_s}\dot{\varphi}_L\,dt =$$

$$\frac{w_2}{w_s}\int_t \dot{\varphi}_R\,dt - \frac{w_2}{w_s}\int_t \dot{\varphi}_L\,dt \Rightarrow$$

$$\frac{w_2}{w_s}\int_0^t \frac{d\varphi_R}{dt}\,dt - \frac{w_2}{w_s}\int_0^t \frac{d\varphi_L}{dt}\,dt \Rightarrow t =$$

$$\frac{w_2}{w_s}\left(\varphi_R^1 - \varphi_R^0\right) - \frac{w_2}{w_s}\left(\varphi_L^1 - \varphi_L^0\right) \Rightarrow$$

$$\text{Orientation} = \frac{w_2}{w_s}\Delta\varphi_R - \frac{w_2}{w_s}\Delta\varphi_L$$

implementing the wheel odometry:
We can now move on to implementing the equations we calculated in the previous lesson in the Python controller node to calculate also the position and orientation of the robot.

Publish Odometry Message
Now we are publishing an odometry message in ROS2, a message that contains information about a robot's position and orientation. This information is used by other nodes to track the robot's location and plan its motion.

- To publish an odometry message, a publisher object is created. This object specifies the message type and the topic name. The publish() method of the publisher object is used to publish a message, taking a message as an argument.

- The provided code example creates a publisher object and publishes an odometry message to the 'bumper_bot_controller/odom' topic. The odometry message includes information about the robot's position, orientation, linear velocity, and angular velocity.

Broadcast Odometry Transform:

- Broadcasting odometry transform allows other nodes to access the robot's pose information.
- Typically done by a dedicated node responsible for collecting and processing odometry data.
- The transform includes position (x, y, z) and orientation (roll, pitch, yaw) information.

Implementation Steps

1. Create a publisher node for odometry data.
2. Use appropriate sensor inputs (e.g., wheel encoders, IMU) to calculate robot pose.
3. Converts pose information to a ROS 2 "geometry_msgs/TransformStamped" message.
4. Set the relevant frame IDs and timestamps for the message.
5. Publish the message on the odometry transform topic.

Subscribing to Odometry Transform

- Other nodes interested in robot pose can subscribe to the odometry transform topic.
- Receive and process the transform for localization, mapping, or control purposes.
- Ensure synchronization with other sensor data for accurate fusion and motion estimation.

Benefits and Use Cases

- Enables real-time access to robot pose information for multiple components in a robotics system.
- Supports localization, mapping, path planning, and control algorithms that rely on odometry data.
- Use cases: mobile robots, autonomous vehicles, drones, collaborative robots, etc.

Example:

a few metrics of the robot, such as the wheel radius and the wheel separation.

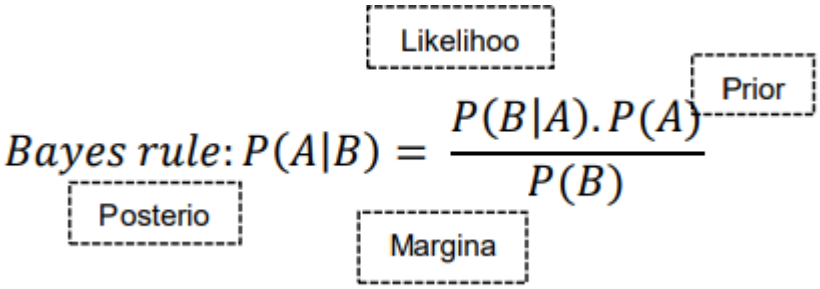
- Now we can see that the robot is start rotating in a circle. And so we can see that also the transform that is published. So the here is the gazebo simulation and here is the transform that is published. We can see that they match perfectly.

4.27 Probability of the robot

In the real world, the environments and surroundings introduces various sources of uncertainty, such as sensor noise, occlusions, and unexpected obstacles. Probability provides a principled way to model and reason about these uncertainties, offering a systematic approach to decision-making in the face of incomplete information which play a vital role in robot localization.

4.27.1 Bayes theory:

Theory that describes step by step how to update the probability of a certain event after getting a new knowledge (observing new evidence).



Posterior: Updated probability of event A, given that we observed the occurrence of a new event B.

Prior: The initial probability (guess) estimated for event A that we will update based on the new observations.

Marginal: represents the overall probability of observing the event B (the probability that event A occurs to influence event A).

Likelihood: The probability of event A assuming event B occurred.

4.27.2 Sensor Noise:

The Noise controller is initialized in the same way as the simple controller and Adding the error to the noisy controller launching the noisy controller to modify the radius & separation to add the error values Comparison between the robot motion values with/without the noisy controller.

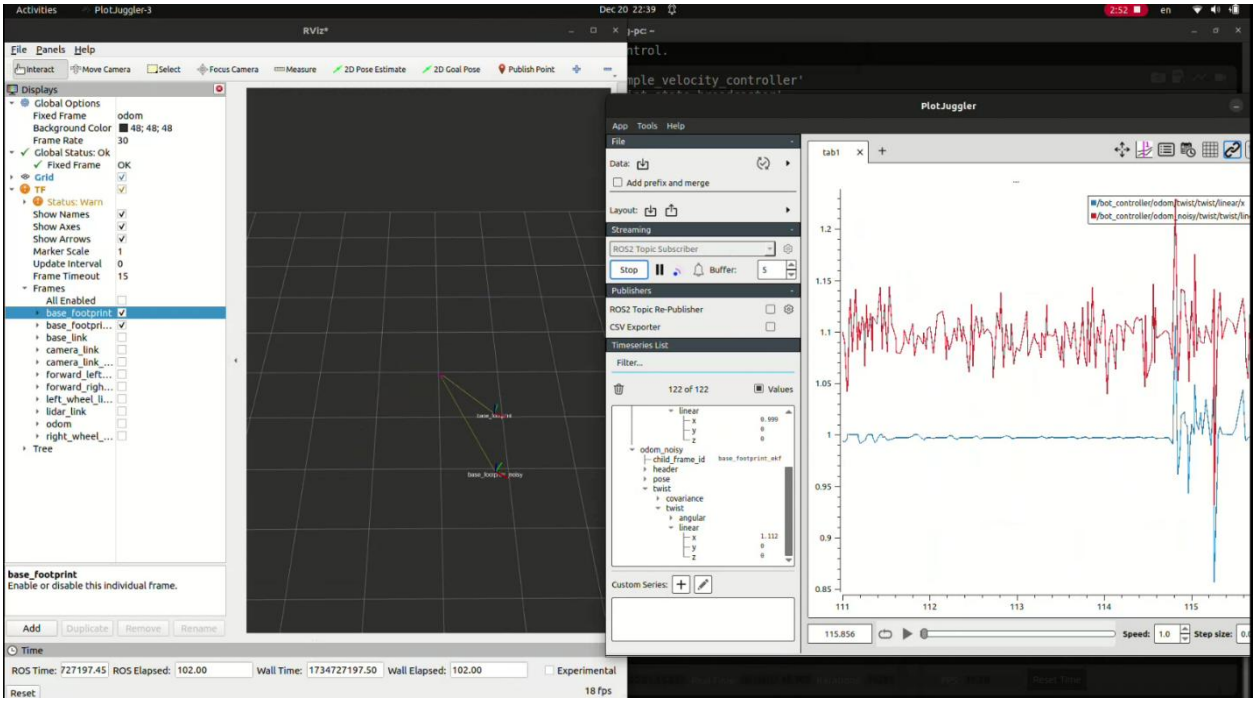


FIGURE 105 SENSOR NOISE

4.28 Sensor Fusion

In the realm of self-driving technology, our focus on probability theory aims for a nuanced understanding of odometry errors. Now, we pivot to sensor fusion, a key aspect. Much like the human body's intricate sensory network, a self-driving robot employs wheel encoders, gyroscopes, accelerometers, cameras, laser sensors, and GPS.

However, each sensor has its strengths and limitations. Wheel encoders struggle on rough terrain, while GPS falters in closed spaces. Enter sensor fusion—a logical system that combines signals from diverse sensors, refining accuracy. This method fortifies the robot's localization, reducing errors and optimizing overall efficiency.

4.28.1 Gyroscopes

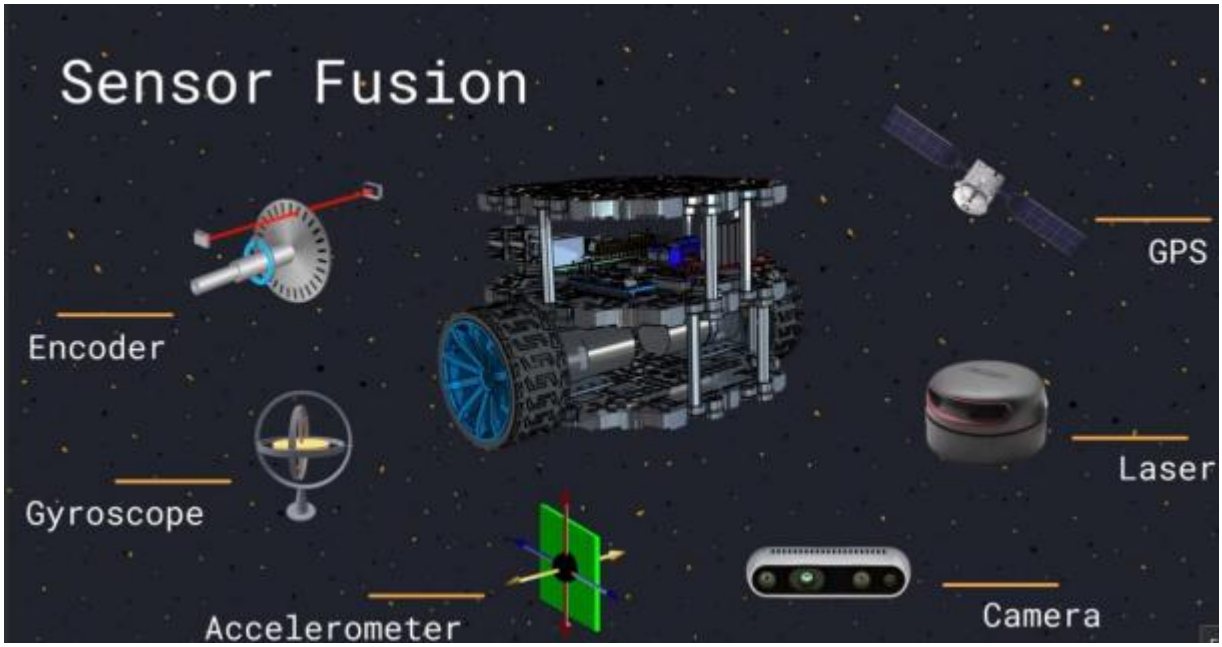


FIGURE 106 GYROSCOPES

Gyroscopes are pivotal sensors that determine a robot's orientation and tilt. These sensors maintain their orientation with respect to a defined reference frame and measure any force attempting to alter the absolute orientation.

To understand the gyroscope's working principle, consider a static disc with a reference frame at its center. Applying torque around an axis induces movement in the same direction as the torque. If the disc is rotating along its central axis and a torque is applied, it starts rotating along a different, perpendicular axis.

In a practical setup, a rotating disc's orientation changes are measured using springs. When torque is applied, the springs compress, and by measuring this compression, the force applied to the spring—and consequently the torque and orientation change—can be determined.

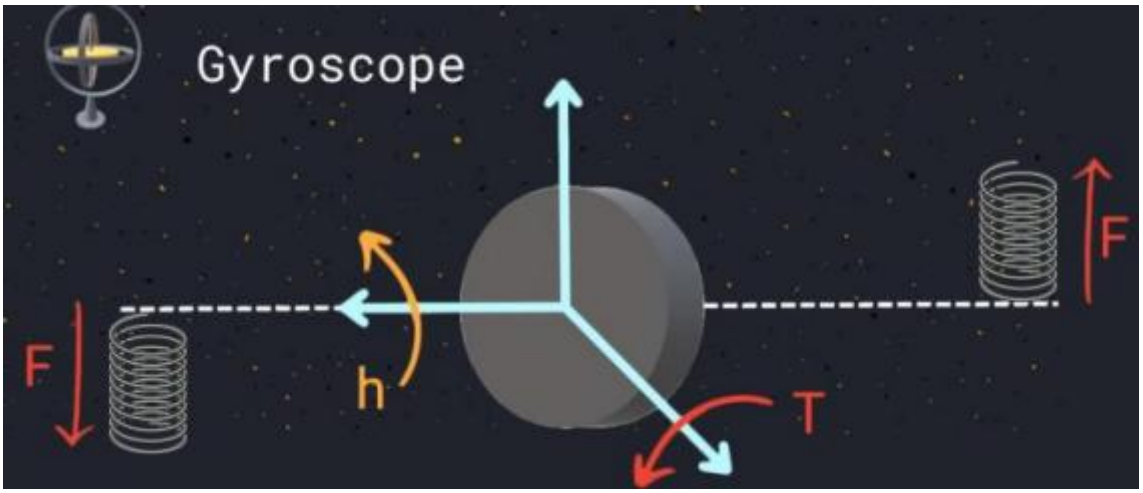
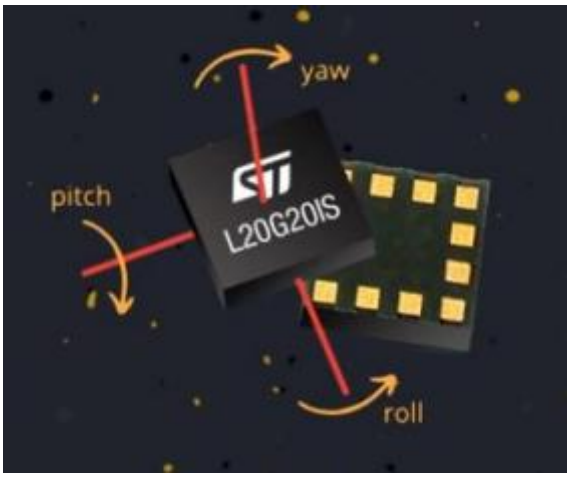


FIGURE 107

Modern gyroscopes, compact as a coin, integrate these principles into small, advanced sensors. These devices, often integrated into a single chip, detect orientation along the three rotational degrees of freedom—roll, pitch, and yaw. While mobile robots typically use only the relevant measurement for their plane of rotation, drones, capable of three-dimensional movement, leverage all three orientation measurements to ascertain their current status, including tilt and orientation.



4.28.2 Accelerometer

Measuring applied acceleration and estimate the robot's velocity. Understanding the physics behind the accelerometer's working principle involves envisioning it as an electromechanical device—a mass connected to a spring.

In simple terms, the spring's equation, featuring its elastic constant (K), reveals how it deforms when subjected to forces. Applying force (F) to the mass causes deformation (X), which is influenced by the elastic constant (K) and the applied force (F).

Newton's Law, connecting force, mass, and acceleration, combined with the elastic force equation, yields an equation linking mass, acceleration, elastic constant, and deformation. Solving for acceleration (A), we find it equals the product of the elastic constant (K) and deformation (X).

With known constants (K and M), the key lies in measuring spring deformation (X) to calculate acceleration. Modern accelerometers, often condensed into a coin-sized chip, use various methods to measure deformation. Some advanced devices integrate three sensors into one chip, enabling the measurement of accelerations along all three degrees of freedom.

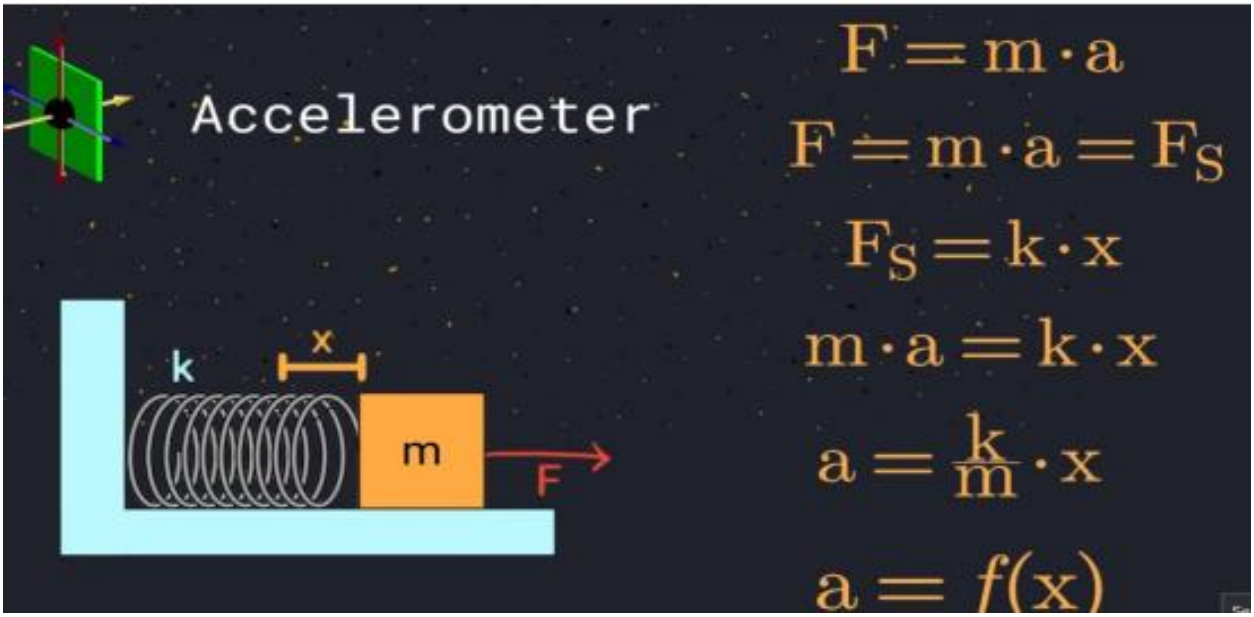


FIGURE 108 ACCELEROMETER

4.28.3 IMU

In the realm of robotics, gyroscopes and accelerometers are frequently integrated into a single chip known as an Inertial Measurement Unit (IMU). The IMU consolidates a gyroscope for orientation calculation (roll, pitch, and yaw axes) and an accelerometer for acceleration calculation along the X, Y, and Z axes, offering a comprehensive solution for dynamic sensing.

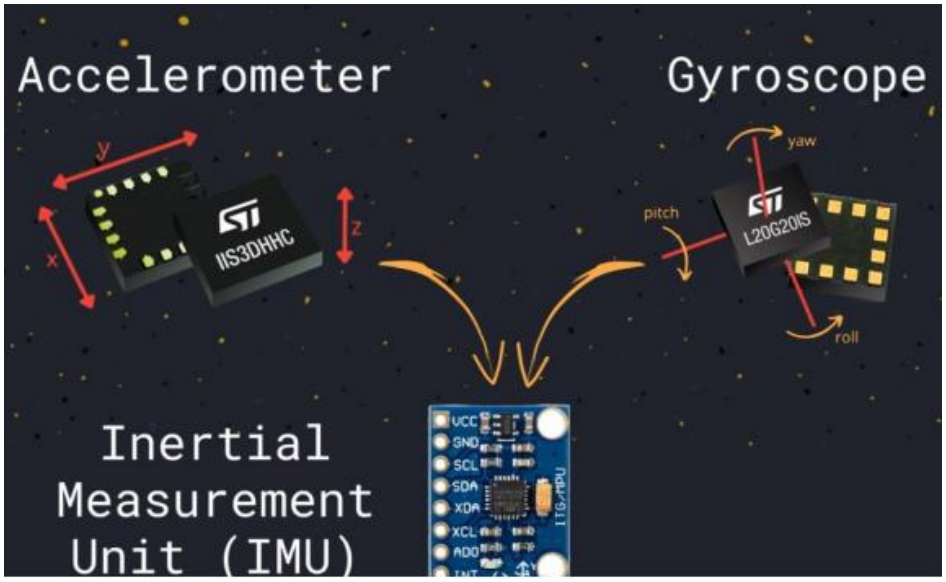


FIGURE 109 IMU

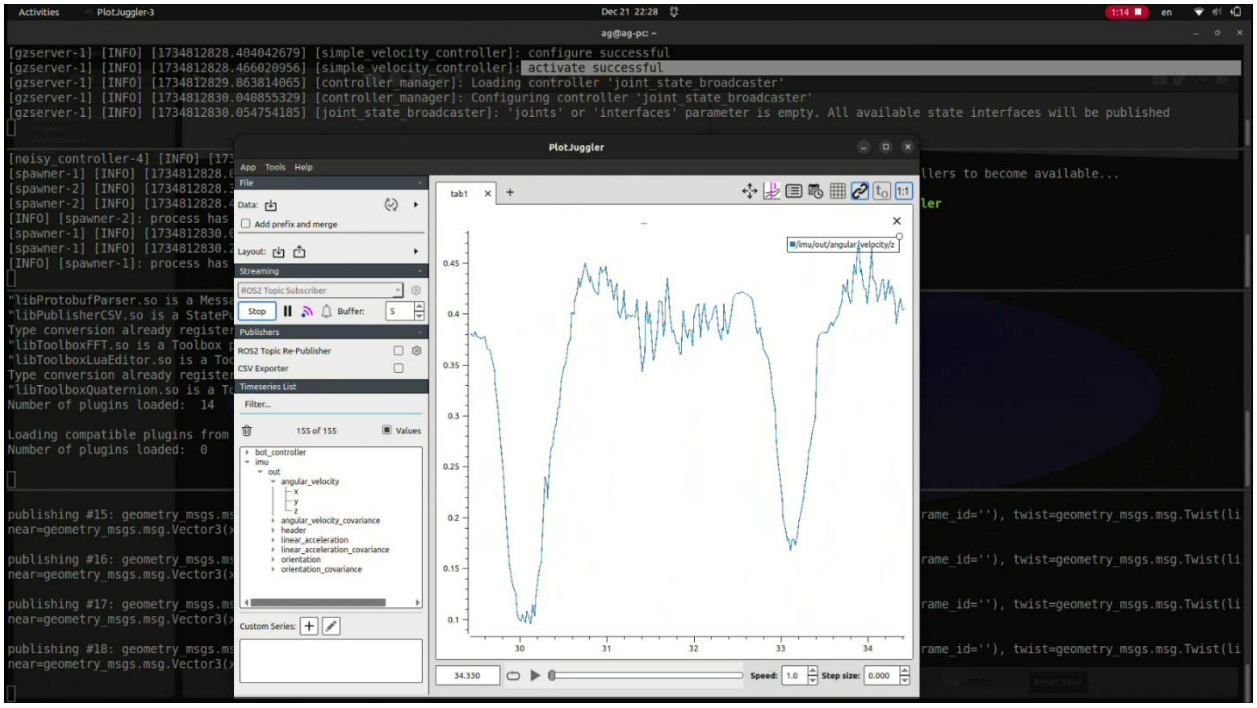


FIGURE 110 IMU

4.29 Kalman Filter

Initial Guess: We begin with an initial estimate of where the robot is, but this estimate is uncertain.

Sensor Measurement Update: The robot has sensors (like cameras or GPS) that provide information about its surroundings. When the sensor takes a measurement, it's not perfect. There is some uncertainty or error associated with it.

The Kalman filter takes this imperfect sensor measurement and combines it with our initial estimate. This combination gives us a new and improved estimate of the robot's position. This new estimate has less uncertainty than the initial guess because the sensor data helps refine it.

Motion Update: The robot moves, but its movement is not perfect either. There is some

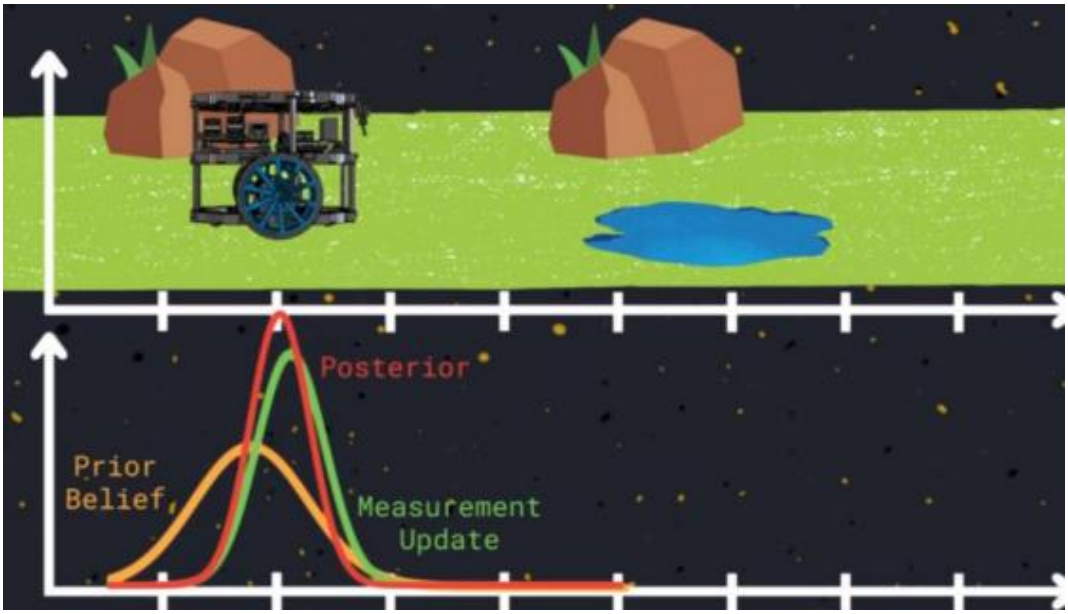


FIGURE 111 KALMAN FILTER

uncertainty in how far and in what direction it actually moved.

The Kalman filter predicts where the robot should be after its movement based on our refined estimate and adds the uncertainty from the robot's motion. This step gives us a new estimate of the robot's position, but now with increased uncertainty due to both the initial guess and the uncertainty introduced by the robot's motion.

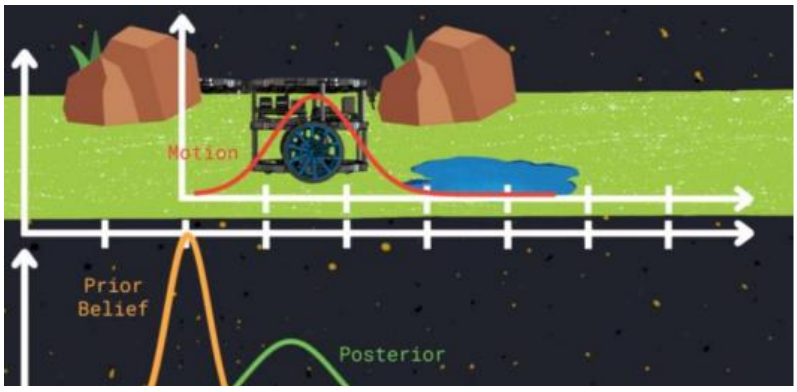


FIGURE 112

Repeat: This process repeats as the robot continues to move and take sensor measurements.

With each iteration, the estimate of the robot's position becomes more accurate because we continuously refine it based on sensor data and predictions of the robot's movement.

In essence, the Kalman filter is like a smart system that starts with a rough idea of where the robot is, uses sensor data to make a better guess, predicts where the robot should be after moving, and refines its estimate as new data comes in. Over time, this iterative process leads to a more accurate understanding of the robot's position, taking into account the uncertainties associated with both sensor measurements and the robot's motion.

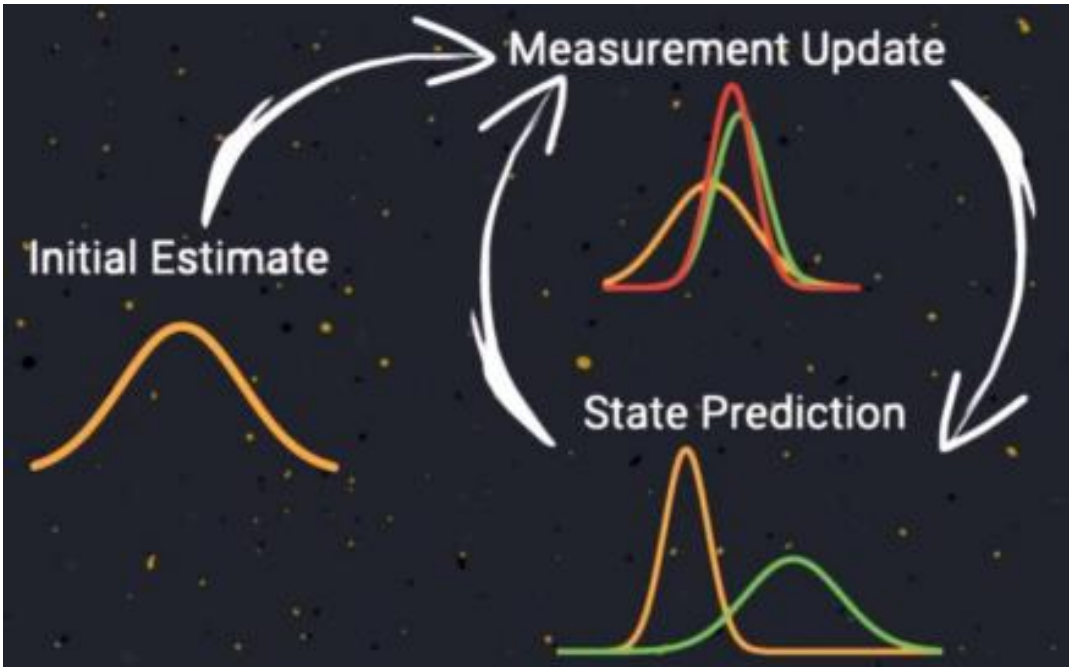


FIGURE 113

4.29.1 Initializing the Kalman filter:

Initialization: Create a new ROS 2 node called "Kalman Filter." Set up subscribers for noisy odometry (from encoders) and IMU data. Create a publisher for the filtered odometry.

Variables: Define variables like mean and variance for the Gaussian distribution of angular velocity. Initialize an IMU angular velocity variable. Create flags and variables for tracking the first wheel odometry message and storing the last angular velocity.

4.29.2 Callback Functions:

IMU Callback: Update the variable storing the last IMU angular velocity measurement.

Odometry Callback: Initialize the Kalman Odometry message. If it's the first odometry message, update the mean and last angular velocity. Set the "first Odom" flag to false after the first odometry message.

Next Iterations: For subsequent iterations (after the first odometry message), execute the Kalman filter algorithm in two separate functions: state prediction and measurement update.

Measurement Update Function: Update the estimate using the IMU measurement. Calculate the Kalman gain and update the mean and variance.

State Prediction Function: Predict the next estimate of angular velocity based on the motion model. Update the mean and variance accordingly.

Publishing: Publish the updated estimation of angular velocity using the Kalman filter algorithm.

Measurement update: in the measurement update step of the Kalman filter, we enhance our initial angular velocity estimate with new IMU sensor readings. Here's a concise breakdown:

Initial Estimate (Prior Belief): From wheel encoders, represented by Gaussian distribution. Parameters: Mean μ1, Variance σ12. New Measurement (IMU Sensor): Another Gaussian distribution. Parameters: Mean μ2, Variance σ22. Assumption: IMU uncertainty is smaller than initial estimate.

Update Step: Multiply the two Gaussians for a refined estimate. Result: New Gaussian (posterior probability). Parameters: Mean μposterior, Variance σposterior2.

Calculation of Mean and Variance: Implement equations to compute mean and variance.

Involves means and variances of prior belief and measurement update.

4.29.3 State Prediction:

Current Estimate (Posterior Probability):

- Result of the measurement update step.
- Gaussian distribution.
- Parameters: Mean μ1, Variance σ12.

Robot Movement (Encoder Sensors):

- New Gaussian distribution.
- Parameters: Mean μ2, Variance σ22.

Prediction Step:

- Combine the two Gaussians for a refined estimate.
- Result: New Gaussian (posterior probability for prediction).
- Parameters: Mean μ prediction, Variance σ prediction2.

Calculation of Mean and Variance:

- Implement equations to compute mean and variance.
- Involves means and variances of the current estimate and the movement estimate.

4.29.4 Visualizing the Kalman Filter

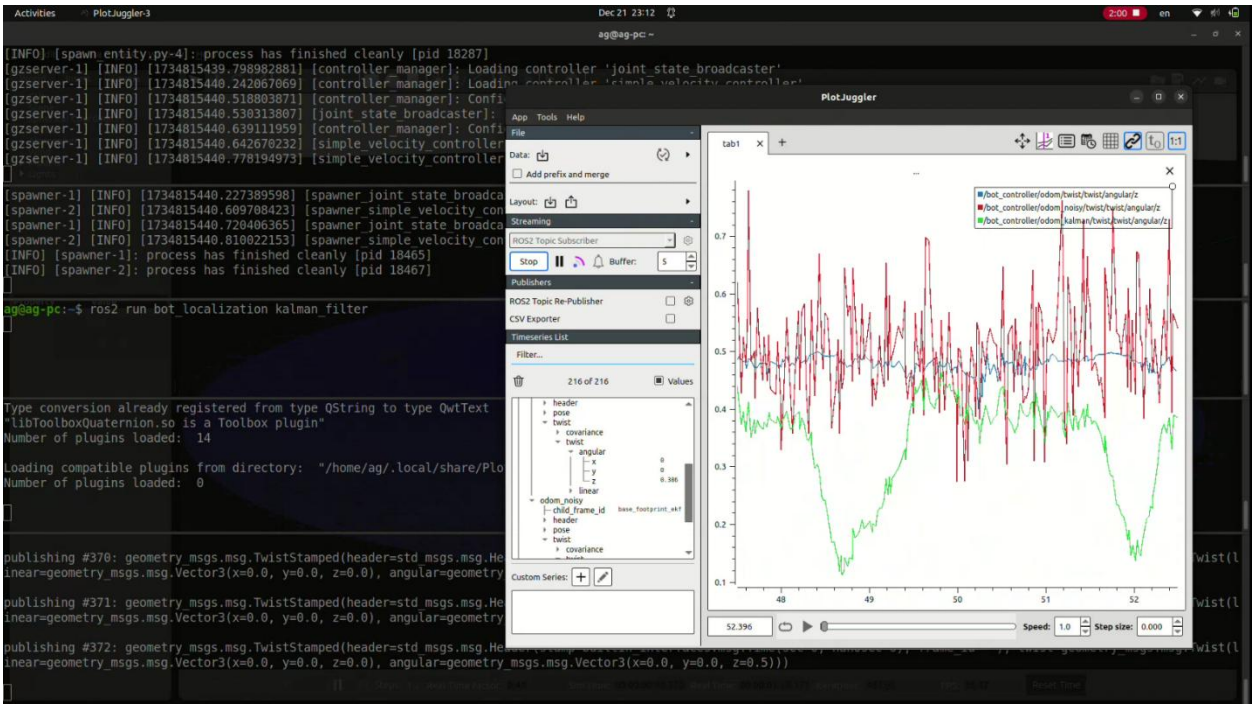


FIGURE 114 VISUALIZING THE KALMAN FILTER

4.30 Extended Kalman Filter:

The Kalman filter enhances the accuracy of angular velocity estimation by mitigating sensor noise. However, its effectiveness is constrained by assumptions that the robot's motion and sensor measurements follow linear models, with the state estimation process adhering to a Gaussian distribution.

In reality, these assumptions often fall short, necessitating the use of the extended Kalman filter in robotics. Unlike the Kalman filter, it accommodates nonlinearities in both the motion and measurement models. When applying nonlinear functions to Gaussian distributions, the extended Kalman filter introduces a linearization step to approximate the nonlinear model with a linear function, allowing the continued use of Gaussian distributions for mean and variance calculations.

This algorithm, akin to the Kalman filter, initiates with a prior belief represented by a Gaussian distribution of the robot's state. The linearization step computes the best linear approximation of the motion and measurement models at a specific point on the nonlinear curve. Subsequently, these linear models are employed in the measurement update and state prediction steps.

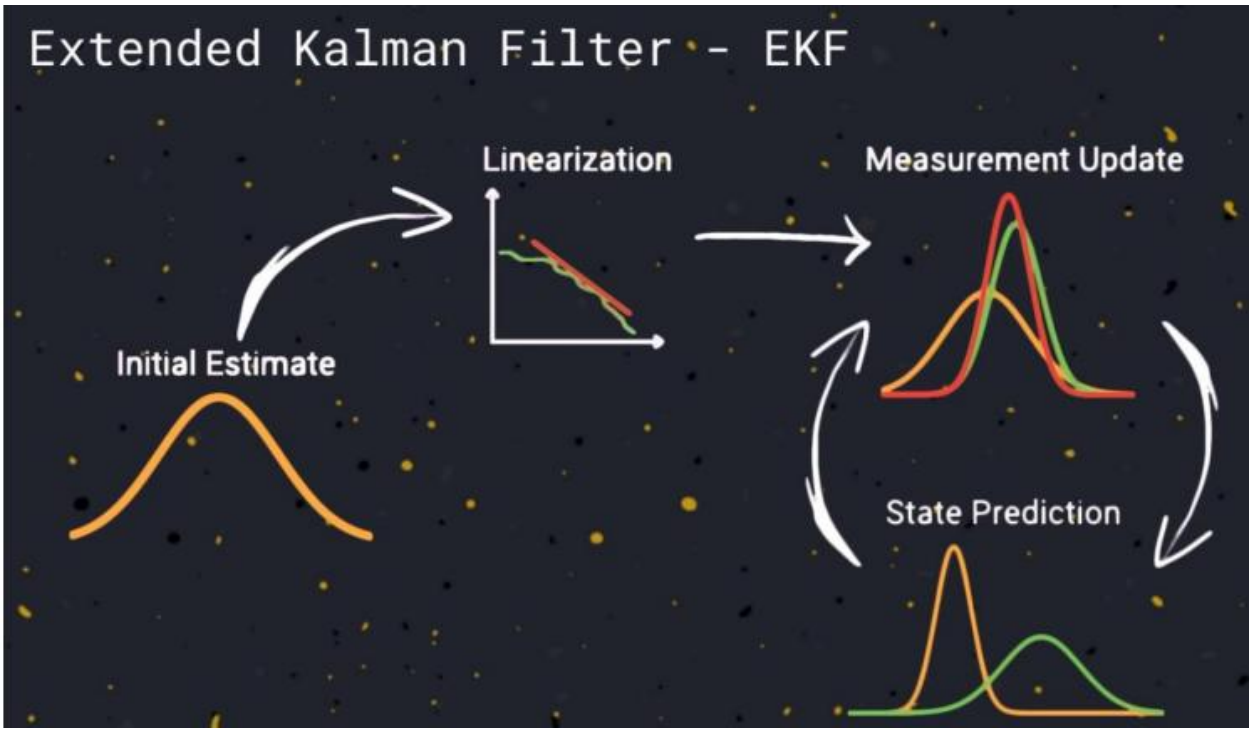


FIGURE 115 EXTENDED KALMAN FILTER

4.30.1 IMU Republisher:

This Python script, named "IMU Republisher" is designed to facilitate the relay of IMU (Inertial Measurement Unit) data between topics in the bumper bot localization package.

The primary purpose is to modify the frame ID within the IMU message, a traightforward yet essential task for configuring the robot localization package.

4.30.2 Robot Localization:

The provided Python script is a ROS2 launch file that configures and launches nodes for robot localization. It utilizes the robot localization library to implement an extended Kalman filter (EKF) for fusing odometry and inertial measurement unit (IMU) data. The launch file creates and sets parameters for nodes such as a static transform publisher, EKF node, and IMU republisher nodes in both Python and C++. The EKF is configured through a YAML file, defining settings such as frame names, input sources, and process noise covariances. This setup enhances the robot's position and orientation estimation by combining information from wheel encoders and IMU readings.

4.30.3 The EKF YAML file:

The ekf.yaml configuration file configures the Ekf filter node with settings such as frequency, two-dimensional mode, frame names, IMU configuration, differential mode for pose measurements, and process noise covariance matrices.

Note: The YAML file contains detailed configurations for IMU and odometry inputs, and specific covariance matrices. These configurations are essential for the extended Kalman filter's operation and can be adjusted based on the robot's characteristics and sensor properties.

4.30.4 Extended Kalman Filter Characteristics

Frequency

Description: The frequency at which the filter will output a position estimate.

Default: 30.0 Hz

two_d_mode

Description: If set to true, no 3D information will be used in the state estimate.

Default: False

publish_tf

Description: Whether to broadcast the transformation over the /tf topic.

Default: True

map_frame

Description: Frame name for the global map.

Default: "map"

odom_frame

Description: Frame name for odometry data.

Default: "odom"

base_link_frame

Description: Frame name affixed to the robot.

Default: "base_link"

world_frame

Description: Frame used to relate multiple map frames.

Default: Same as odom_frame

imu0

Description: Name of the IMU sensor.

Default: "imu_ekf"

imu0_config

Description: A vector specifying which values from the IMU should be used in the filter.

Default: [false, false, false, false, false, true, false, false, false, false, false, true, true, false, false]

imu0_differential

Description: If true, converts absolute pose data to velocity data for IMU.

Default: True

imu0_pose_use_child_frame

Description: If true, use the child frame for IMU pose

Default: False

odom0

Description: Topic name for the odometry input.

Default: "bumperbot_controller/odom_noisy"

odom0_config

Description: A vector specifying which values from odometry should be used in the filter.

Default: [false, false, false, false, false, false, true, true, false, false, false, false, false, false, false]

odom0_differential

Description: If true, converts absolute pose data to velocity data for odometry.

Default: True

odom0_pose_use_child_frame

Description: If true, use the child frame for odometry pose.

Default: False

process_noise_covariance

Description: Matrix representing the noise added to the total error after each prediction step.

Default: A large matrix specifying noise values for different state variables.

initial_estimate_covariance

Description: Initial value for the state estimate error covariance matrix.

Default: A matrix with small values for diagonal entries, encouraging rapid convergence for initial measurements.

4.30.5 Visualizing the Robot:

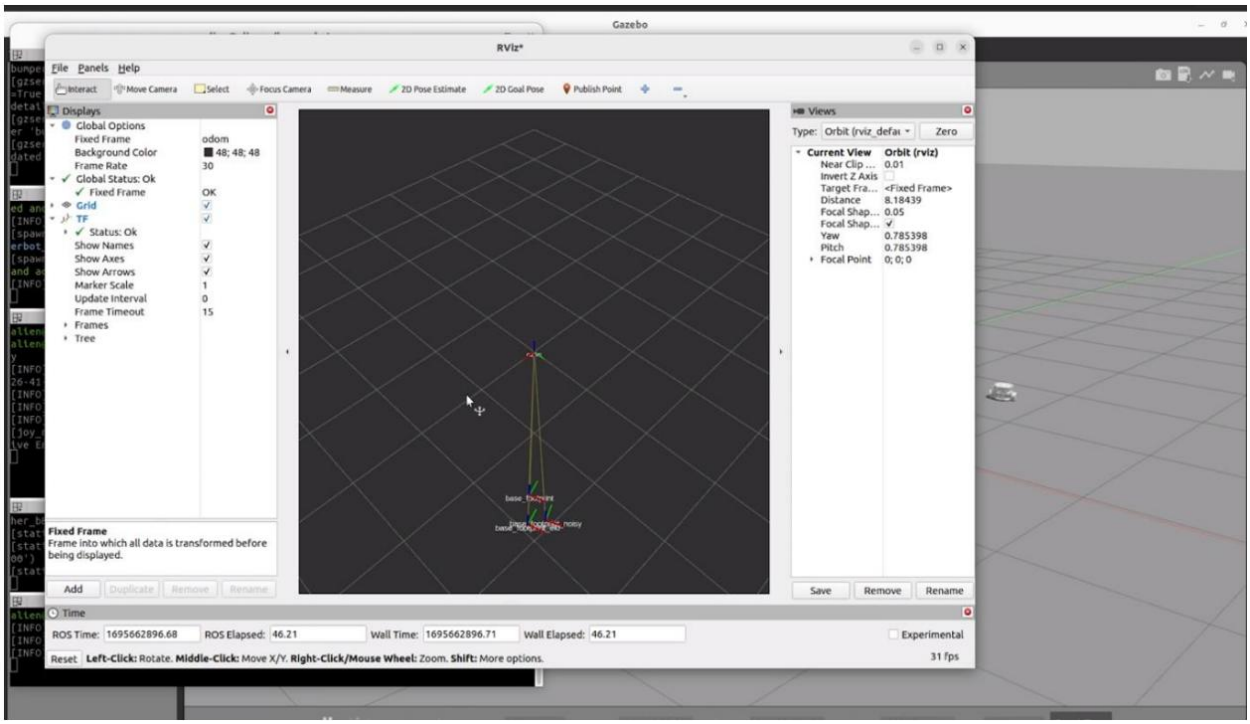


FIGURE 116

4.31 ROS2 Localization, SLAM and navigation

Introduction

Robot localization (Localization) is one of the most critical challenges faced by modern robotics, as it serves as the foundation for autonomous navigation. This course aims to provide a comprehensive explanation of the localization process using tools, mathematical models, and virtual environments, as well as covering errors and how to handle them.

4.31.1 Localization Concept

Localization is the process of accurately determining the robot's position relative to its surrounding environment. This process relies on multiple sensors and mathematical models to ensure precise performance.

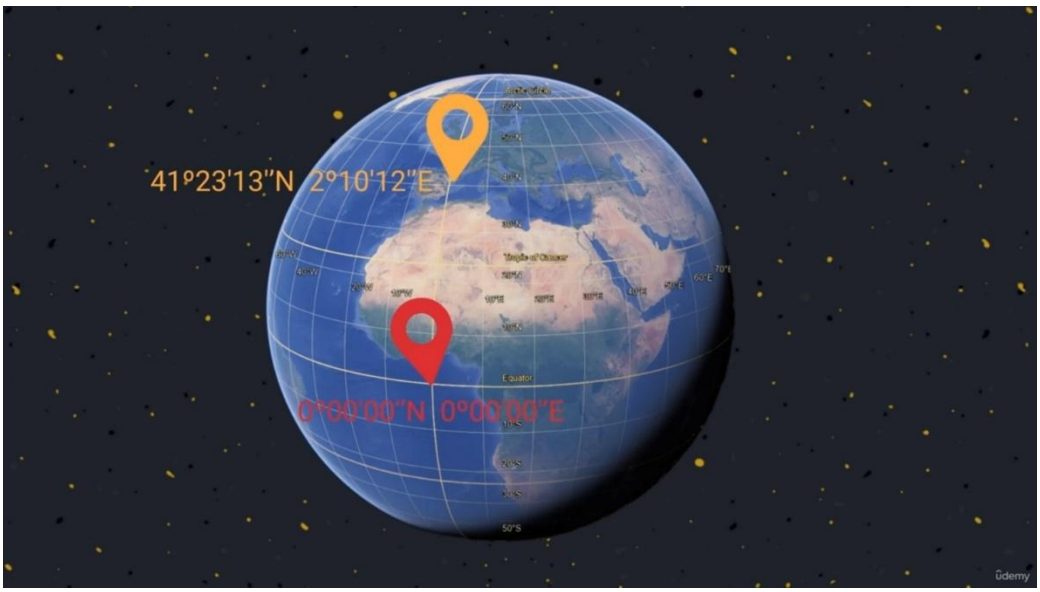


FIGURE 117 LOCALIZATION CONCEPT

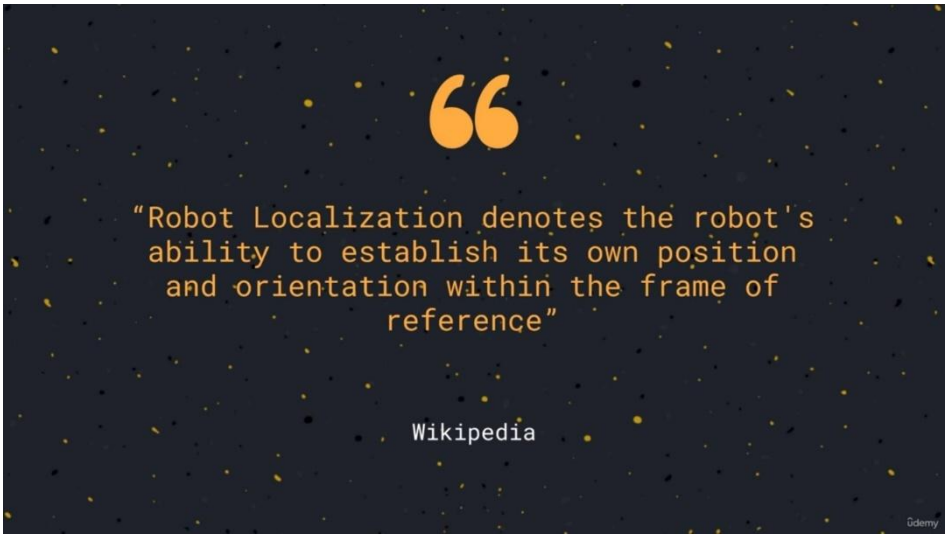


FIGURE 118 LOCALIZATION CONCEPT

FIGURE 119

Types:

1. **Local Localization:**
   - Used when the robot knows its initial position.
   - Operates in known and fixed environments.
2. **Global Localization:**
   - Used when the robot is unaware of or has lost its position.
   - Relies on searching for the location across the entire map.

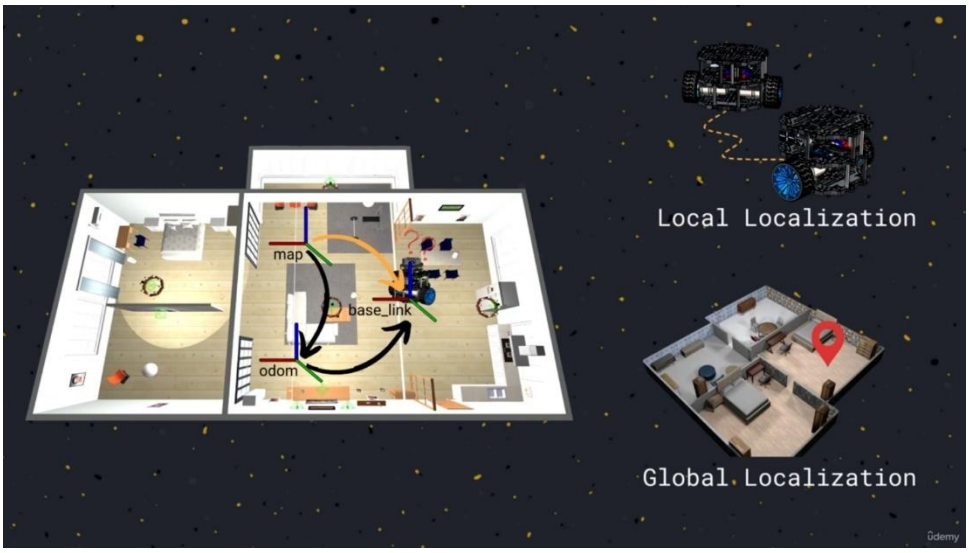


FIGURE 120

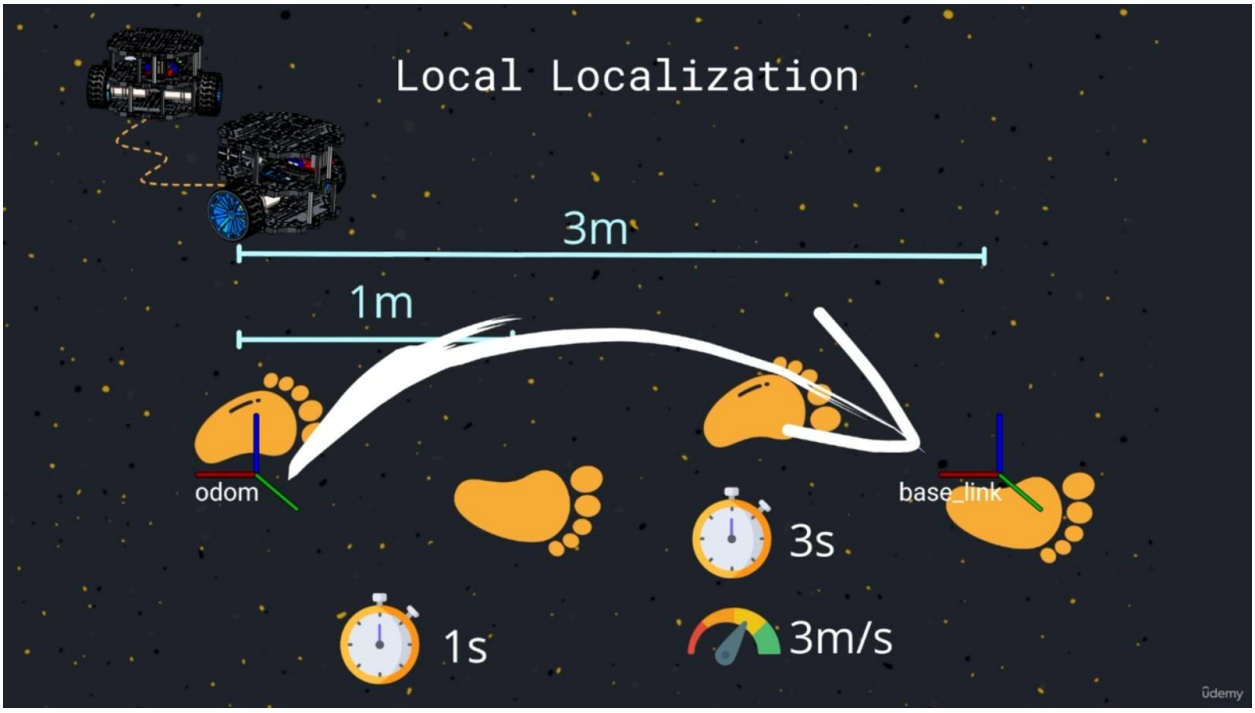


FIGURE 121

Applications:

1. Self-driving cars.
2. Industrial robots.
3. Cleaning and service robots.

Sensors Used:

1. **LiDAR:**

- Measures distances using laser beams.
- Wheels (Wheel Odometry):
- Tracks movement through wheel rotations.

2. **Cameras:**

- Identifies visual landmarks.

Errors and Localization Challenges:

1. **Wheel Odometry Errors:**
- Occur due to Wheel slippage.
2. **Mechanical or kinematic inaccuracies.**
- These errors accumulate over time, making odometry alone insufficient for accurate localization.
3. **Laser Odometry Errors:**
- Arise from sensor noise or laser beam reflection off unexpected surfaces.

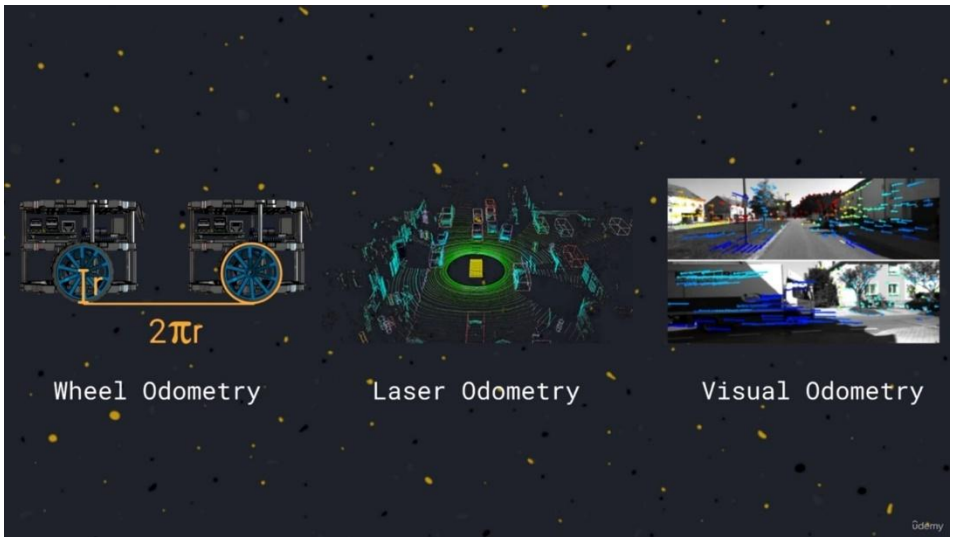


FIGURE 122

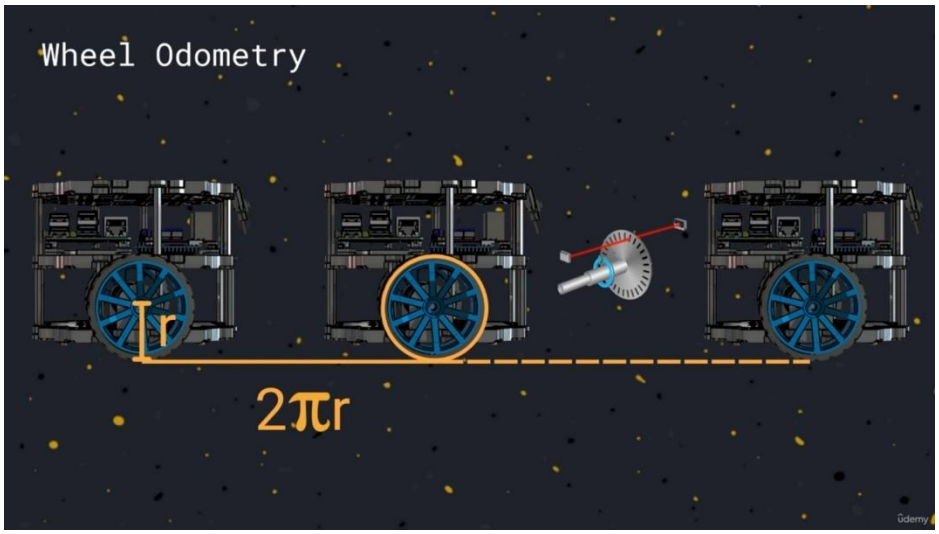


FIGURE 123

Error Propagation:

Small measurement errors magnify over time if left uncorrected.

Mitigated using tools like:

- **Kalman Filter** to reduce noise.
- **Sensor Fusion** to combine data from multiple sensors.

FIGURE 124

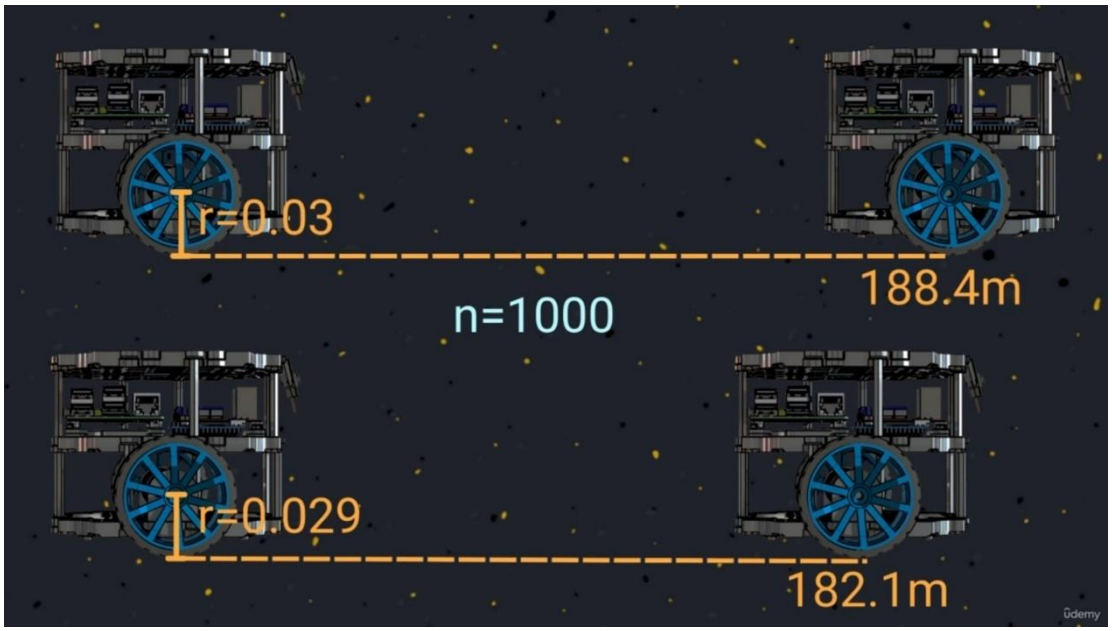


FIGURE 125

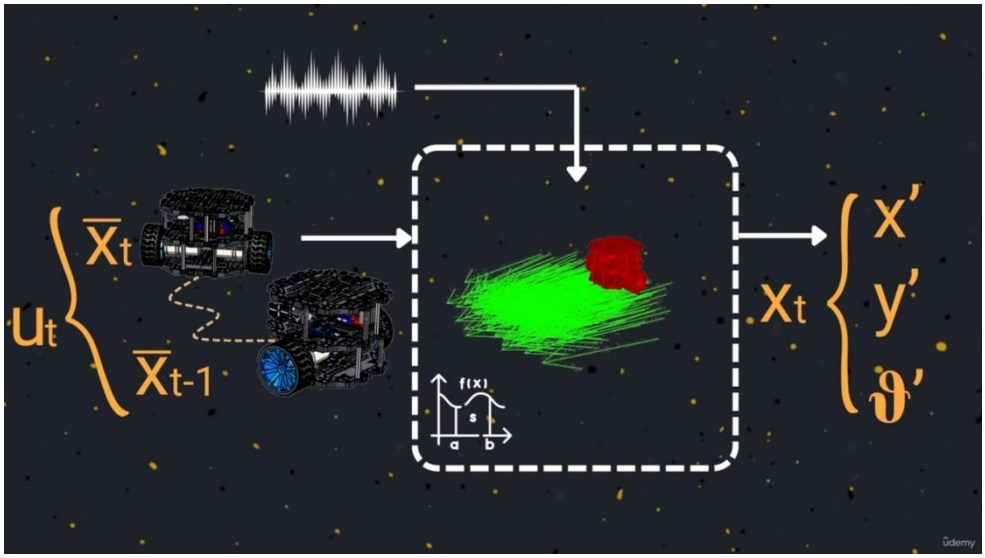
Models and Algorithms Used:

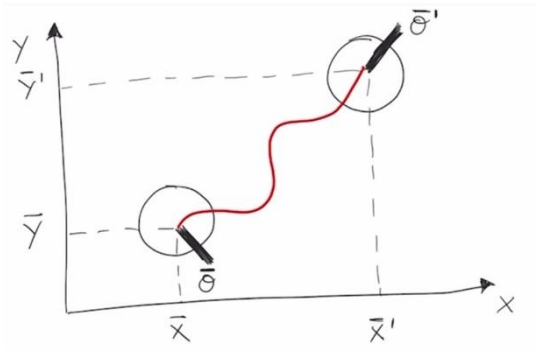
1. **Odometry Motion Model:** A mathematical model describing how the robot moves, based on equations involving: Angles, Speed, Distance traveled.
2. **Global Localization Using MCL (Monte Carlo Localization):** Relies on probability distribution to determine the robot's position.

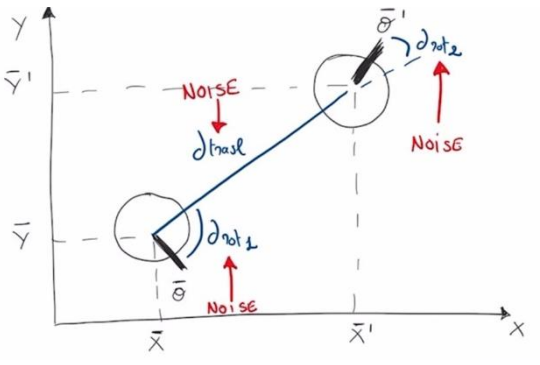
Tests multiple possible locations and compares them to sensor data to find the actual position:

- **Kalman Filter**: A technique to merge data from various sensors and reduce noise in measurements.

Odometry motion model





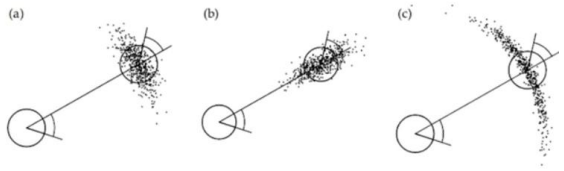


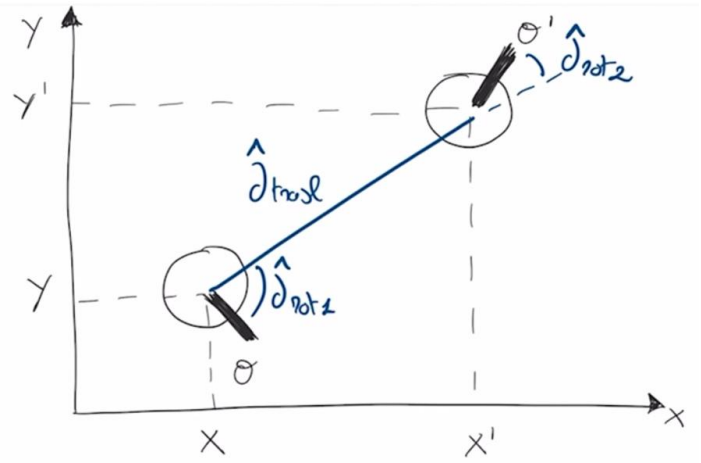
$$\begin{cases} \delta_{trasl} = \sqrt{(\bar{x}-\bar{x}')^2 + (\bar{y}-\bar{y}')^2} \\ \delta_{rot1} = \operatorname{atan2}(\bar{y}'-\bar{y}, \bar{x}'-\bar{x}) - \bar{\theta} \\ \delta_{rot2} = \bar{\theta}' - \bar{\theta} - \delta_{rot1} \end{cases}$$

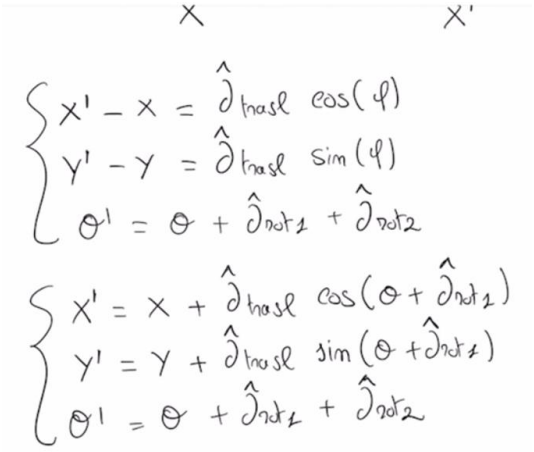
NOISE

$$\begin{cases} \hat{\delta}_{rot1} = \delta_{rot1} - prob(\alpha_1 \delta_{rot1} + \alpha_2 \delta_{trasl}) \\ \hat{\delta}_{trasl} = \delta_{trasl} - prob(\alpha_3 \delta_{trasl} + \alpha_4 (\delta_{rot1} + \delta_{rot2})) \\ \hat{\delta}_{rot2} = \delta_{rot2} - prob(\alpha_1 \delta_{rot2} + \alpha_2 \delta_{trasl}) \end{cases}$$

NOISE FREE







$$\begin{cases} x' - x = \hat{\delta}_{trasl} \cos(\varphi) \\ y' - y = \hat{\delta}_{trasl} \sin(\varphi) \\ \theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2} \end{cases}$$

$$\begin{cases} x' = x + \hat{\delta}_{trasl} \cos(\theta + \hat{\delta}_{rot1}) \\ y' = y + \hat{\delta}_{trasl} \sin(\theta + \hat{\delta}_{rot1}) \\ \theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2} \end{cases}$$

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \hat{\delta}_{trasl} \cos(\theta + \hat{\delta}_{rot1}) \\ \hat{\delta}_{trasl} \sin(\theta + \hat{\delta}_{rot1}) \\ \hat{\delta}_{rot1} + \hat{\delta}_{rot2} \end{bmatrix}$$

New Position — Old Position

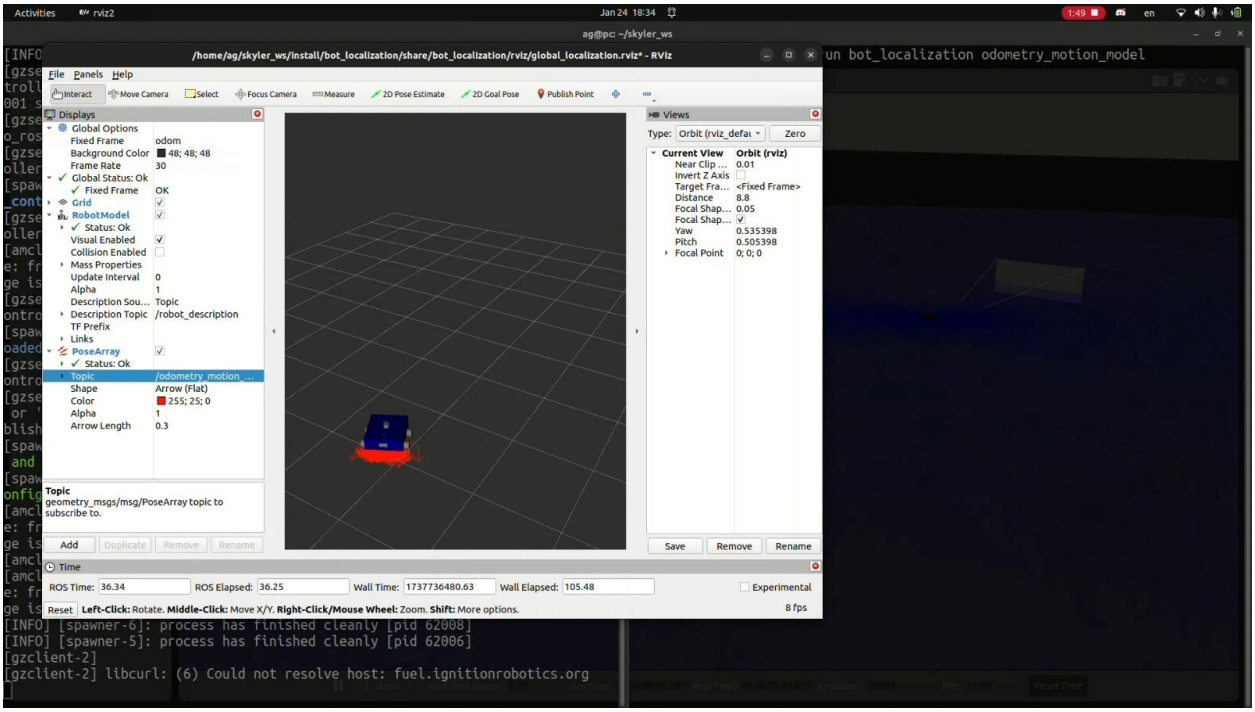


FIGURE 126 LAB MOTION ODOMETRY

4.31.2 Mapping Concept

A map in robotics is a representation of the environment that the robot uses to navigate and localize itself. It can be:

- **Metric Map:** Represents the environment in a detailed geometric form (e.g., grids, coordinates).
- **Topological Map:** Represents the environment as a network of connected points or areas.
- **Semantic Map:** Adds meaning to the environment, such as labeling objects (e.g., "chair," "table").

FIGURE 127

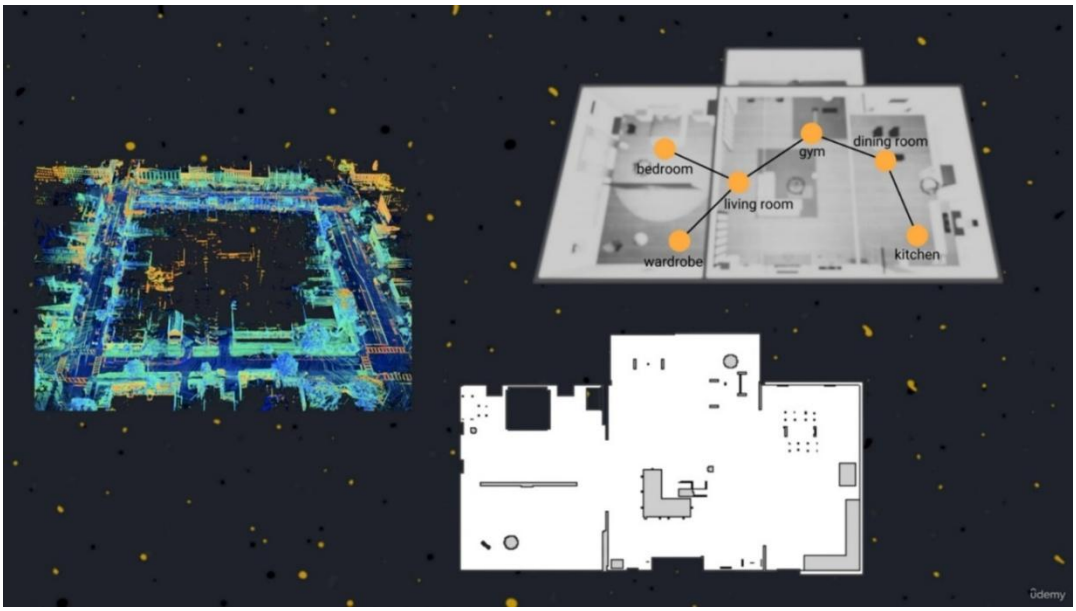


FIGURE 128

Maps are used for:

- Path planning.
- Obstacle avoidance.
- Localization.

How Robots Perceive the World?

Robots perceive the world through sensors that collect data from the environment. This perception process involves:

1. **Environmental Features:**
- Detecting objects, walls, or landmarks.
- Spatial Understanding:
- Understanding distances, angles, and relative positions.
2. **Data Processing:**
- Analyzing raw sensor data and converting it into usable information, such as maps or obstacle locations.
3. **Techniques include:**
- Point Cloud Mapping: Using sensors like LiDAR to create 3D models of the surroundings.
- Visual Perception: Analyzing images from cameras to recognize objects and features.

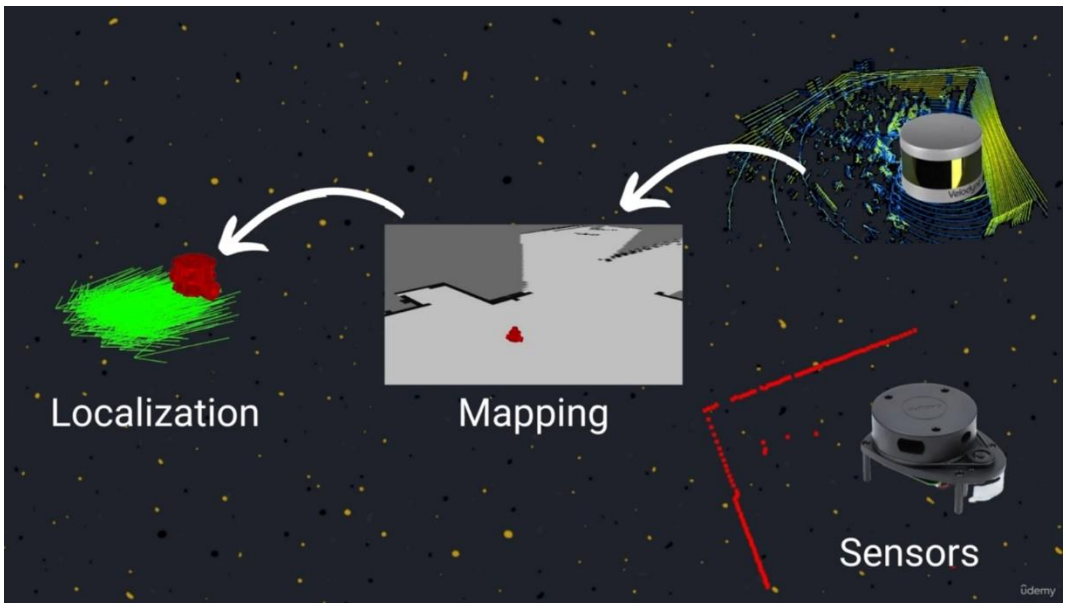


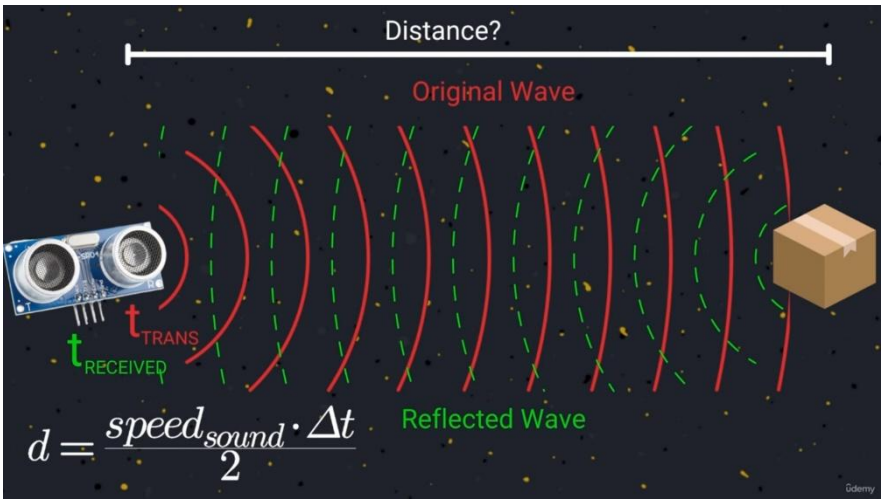
FIGURE 129

Sensors for Mapping

Types of Sensors Used:

FIGURE 130

1. **1D Sensors:**
   Ultrasonic Sensors:
   - Measure distance to nearby objects.
   - Commonly used for obstacle detection.



   - Advantages:
     - Inexpensive.
     - Simple to implement.

- Disadvantages:
    - Limited range and resolution.
    - Prone to noise in complex environments.

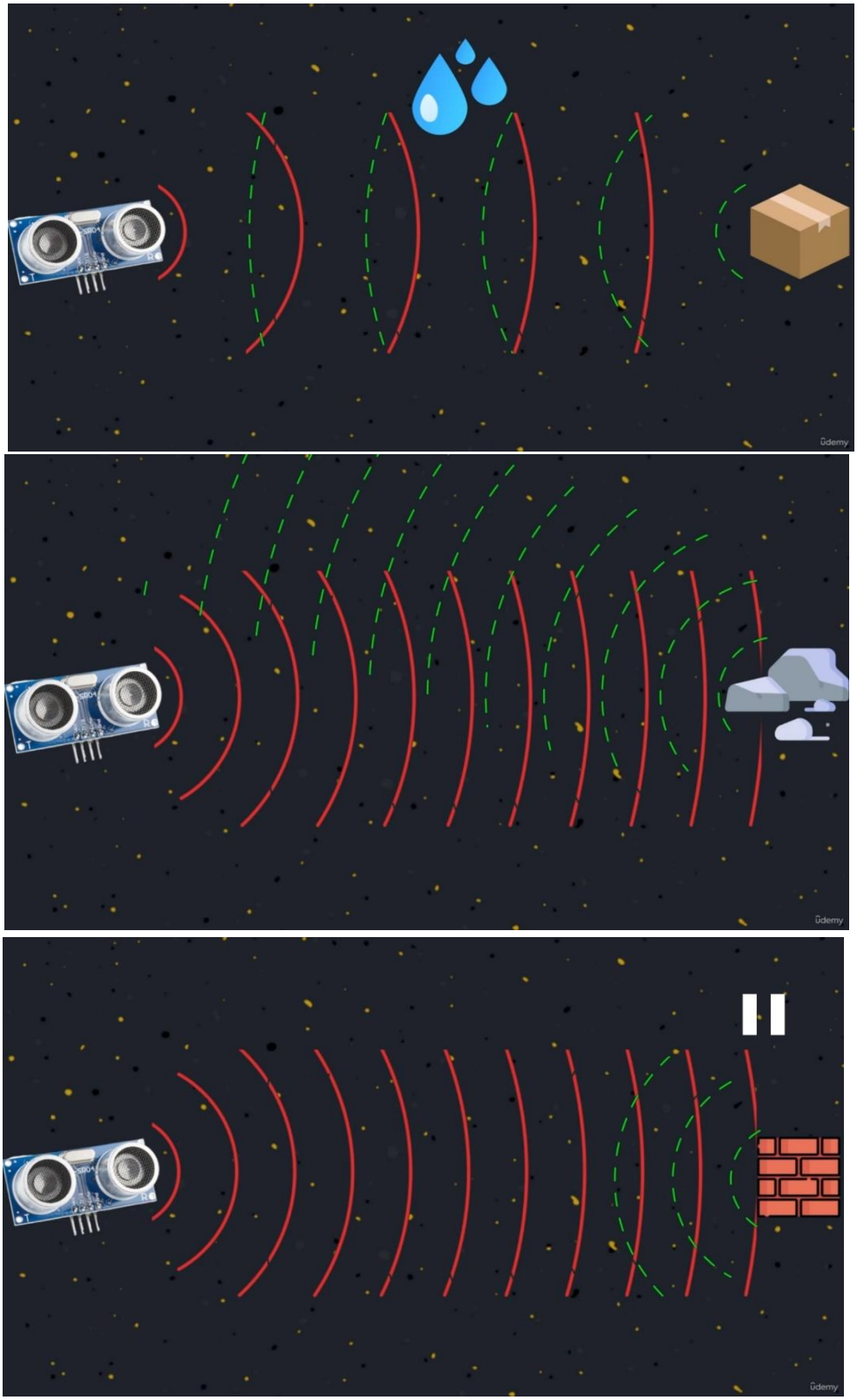
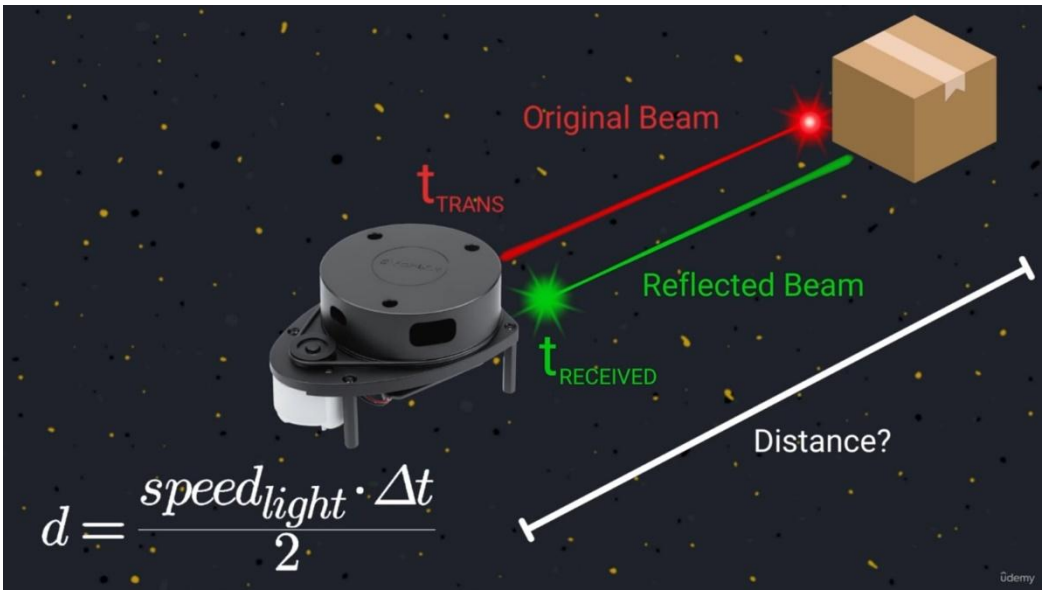


FIGURE 131

2. **2D Sensors:**

LiDAR:

- Scans the environment in 2D to create a detailed map.



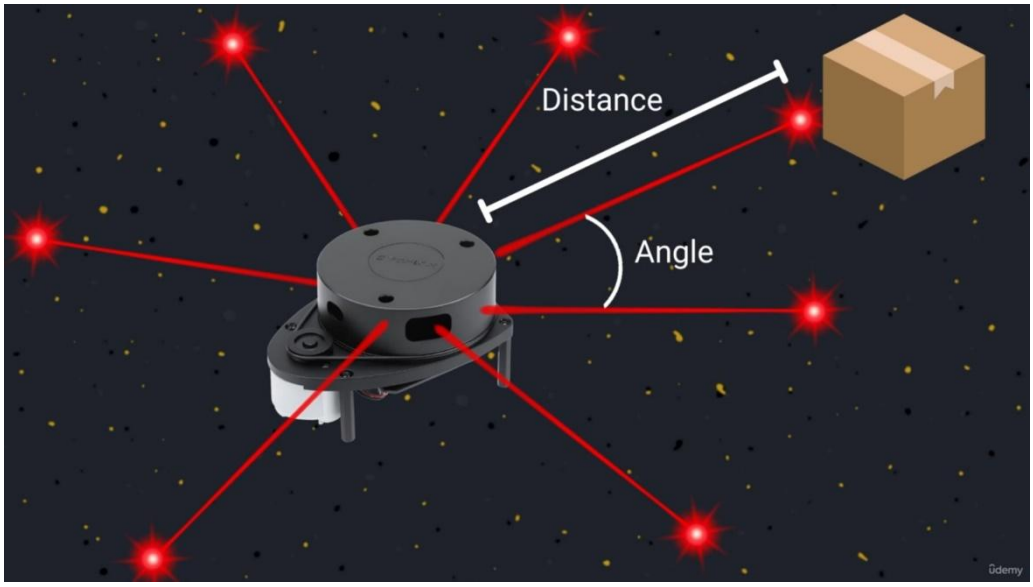


FIGURE 132

- Advantages:
  - High precision.
  - Works well in various lighting conditions.

- Disadvantages:
    - Expensive.
    - Can struggle with reflective or transparent surfaces.

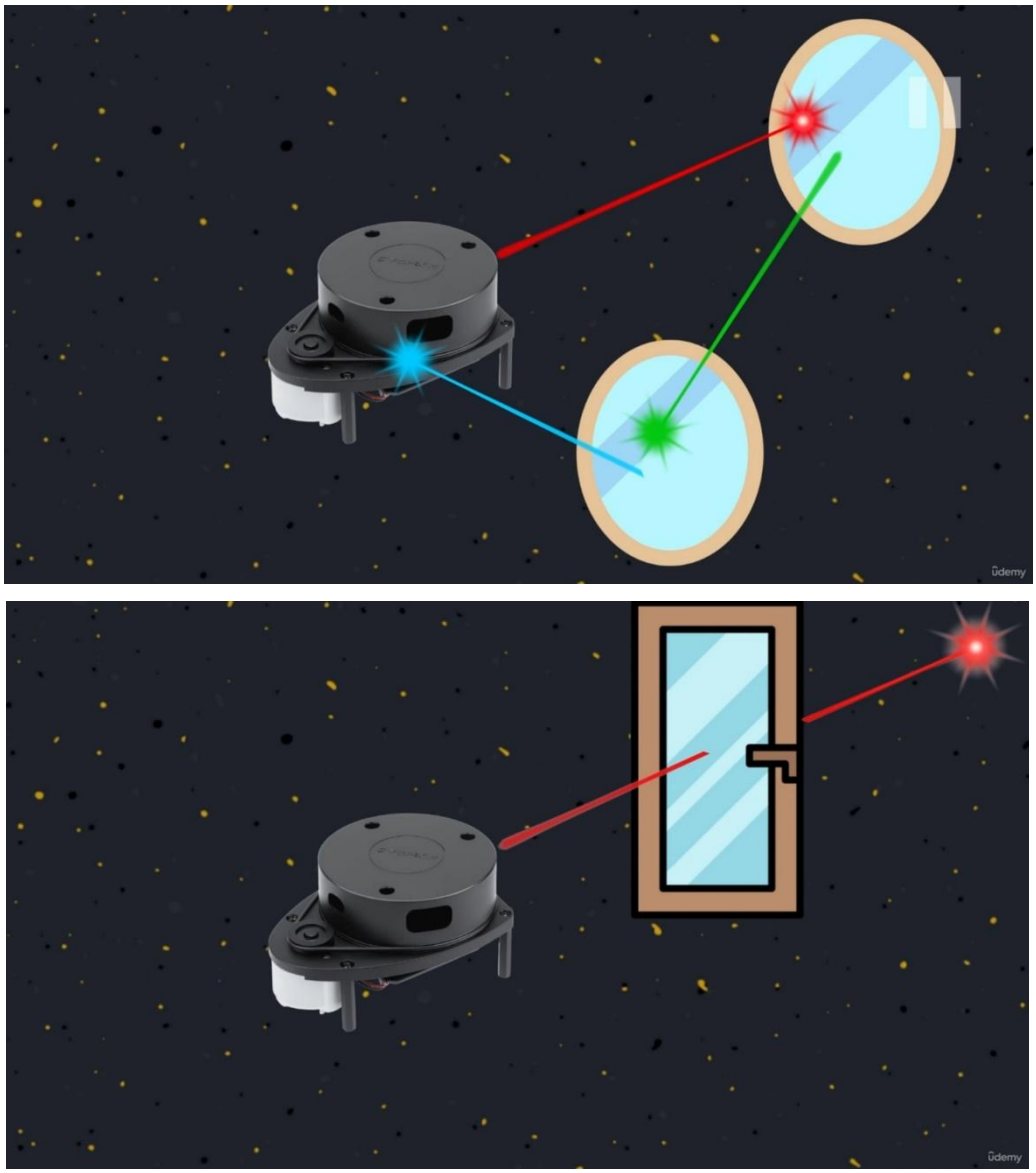


FIGURE 133

Cameras (Monocular):

- Captures 2D images of the environment.
- Advantages:
  - Inexpensive compared to LiDAR.
  - Provides rich visual information.
- Disadvantages:
  - Requires significant computational power for processing.
  - Limited depth perception.

3. 3D Sensors:

Stereo Cameras:

- Use two lenses to estimate depth and create 3D maps.
- Advantages:
  - Rich visual and depth information.
  - Cost-effective compared to LiDAR.
- Disadvantages:
  - Computationally intensive.
  - Affected by lighting and texture conditions.

LiDAR (3D):

- Scans the environment in 3D to create highly detailed maps.

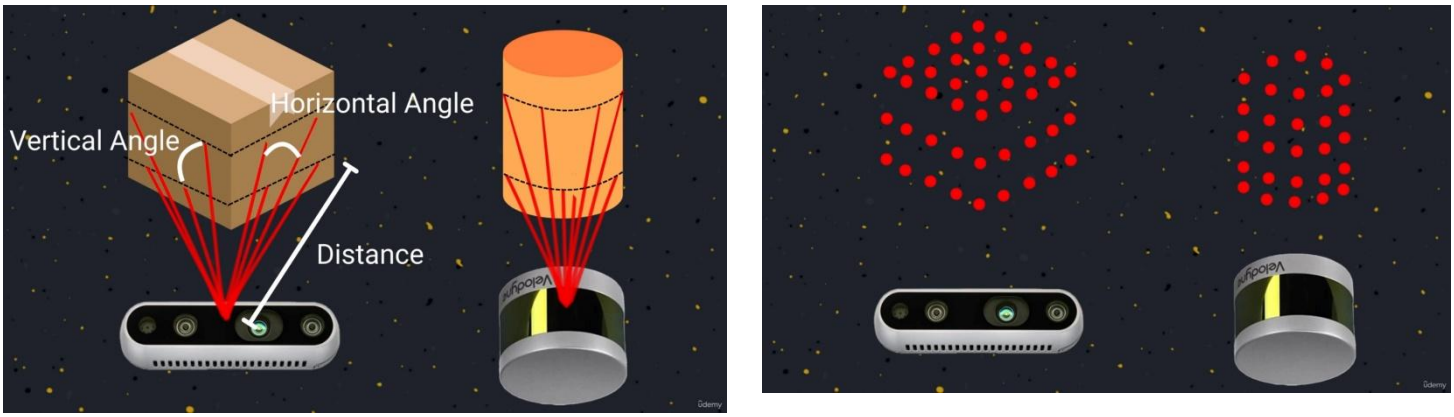


FIGURE 134

- Advantages:
  - Extremely accurate.
  - Ideal for dynamic and complex environments.
- Disadvantages:
  - Very expensive.
  - Requires advanced processing.
  - RGB-D Cameras:

Combine RGB images with depth information.

- Advantages:
  - Affordable compared to LiDAR.
  - Provide both color and depth data.
- Disadvantages:
  - Limited range.
  - Performance depends on lighting conditions.

4.31.2.1 Safety Monitoring and Speed Regulation

The interaction between robots and humans has advanced significantly, with a key focus on ensuring safety in shared environments. One of the most critical aspects of this interaction is dividing the robot's operational area into zones that define the level of interaction and response speed. These zones ensure that the robot reacts appropriately to the presence of a human, reducing risks and enhancing safety:

1. **Stop Zone:**
   - This is the closest range to the robot. If a human enters this zone, the robot immediately stops all operations to ensure safety. This is achieved through precise sensors, such as LiDAR or cameras, which detect proximity and potential hazards.
2. **Slow Down Zone:**
   - Positioned between the stop zone and the safe zone, this area prompts the robot to reduce its speed gradually as a precautionary measure. This ensures smooth operation while minimizing the risk of collision if the human continues to approach.
3. **Safe Zone:**
   - The outermost area where the robot can operate freely without needing to adjust its speed or trajectory. The human's presence in this zone poses no threat, allowing the robot to function at full efficiency.

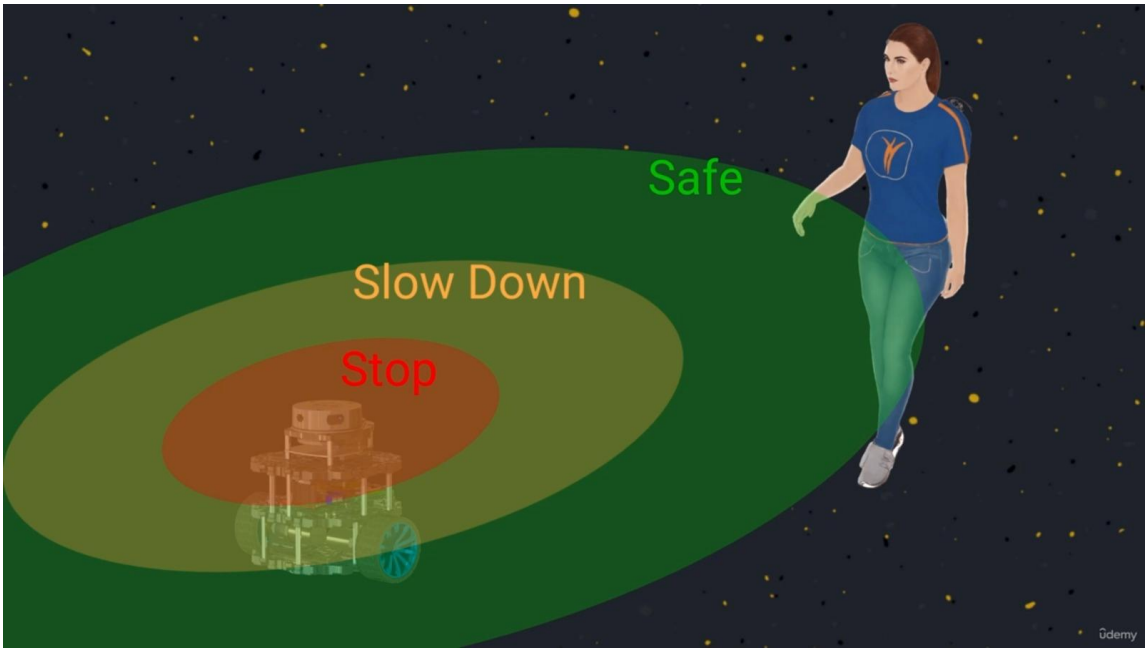


FIGURE 135

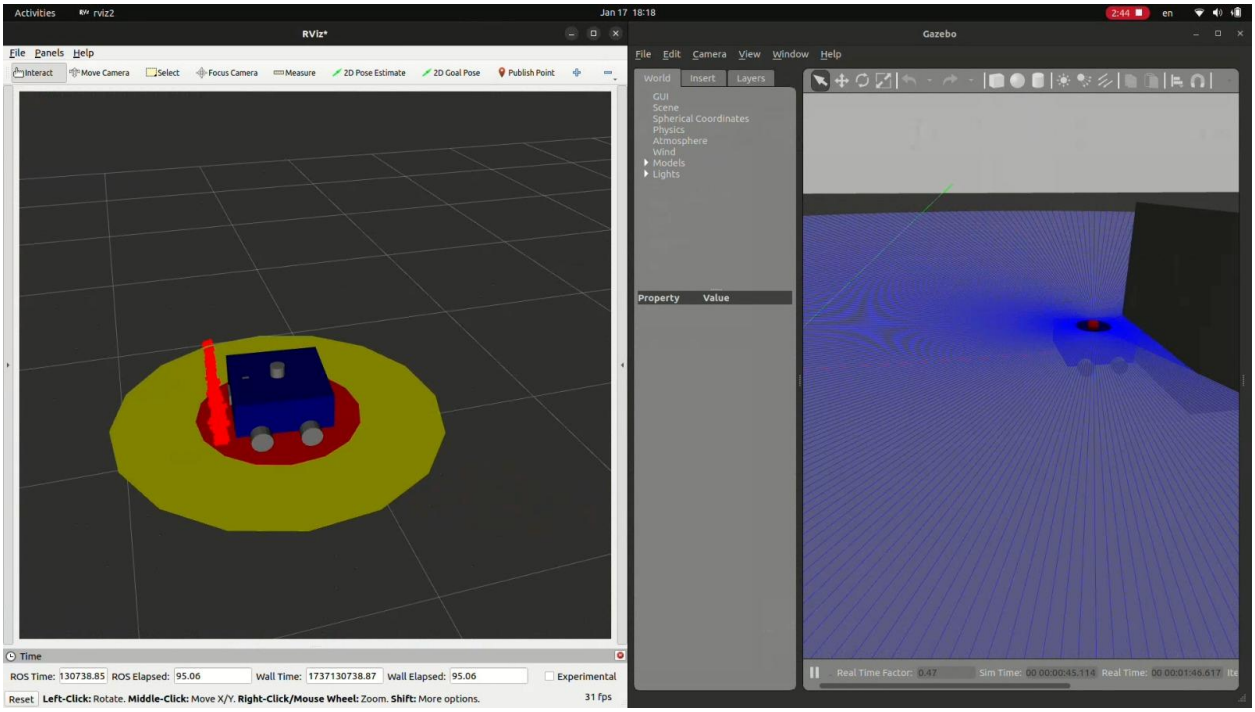


FIGURE 136 LAB ZONES

How It Works:

These systems rely on advanced algorithms that process real-time data from sensors to accurately determine the human's location relative to the robot. Based on the location, the robot dynamically adjusts its behavior. Such systems are commonly implemented in industries like manufacturing, warehouses, and healthcare to maintain a safe working environment.

Benefits of the System:

- Protects humans from accidents during interactions with robots.
- Improves the efficiency of human-robot collaboration.
- Minimizes unnecessary downtime for robots while ensuring safety.
- Feel free to expand or modify this content based on your book's specific focus or audience.

4.31.2.2 Twist Mux (Twist Multiplexer)

The Twist Mux is a crucial node in ROS (Robot Operating System) that handles multiple velocity command inputs (geometry_msgs/Twist) and ensures that only one of them is sent to the robot's actuators at a time. It operates based on priorities and timeouts for each input source, allowing developers to manage control commands from various sources, such as teleoperation, autonomous navigation, or web interfaces.

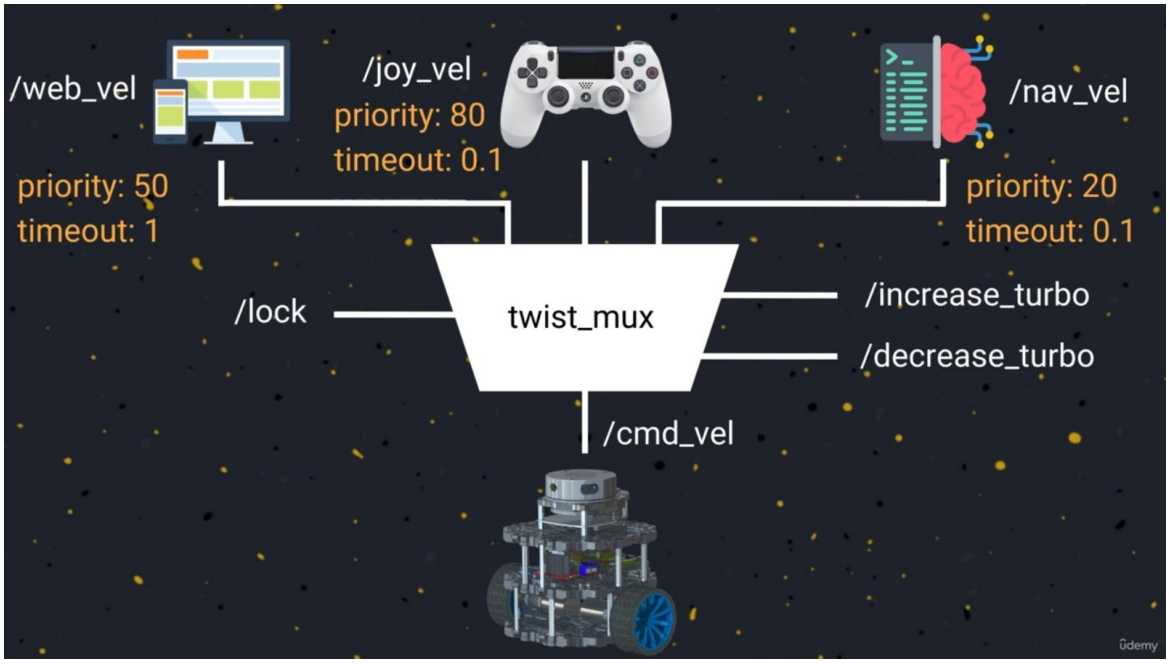


FIGURE 137 TWIST MULTIPLEXER

Key Components in the Diagram

1. **Input Sources:**

- /web_vel:
  - Source: Web interface (e.g., remote dashboard).
  - Priority: 50.
  - Timeout: 1 second. If no command is received within this period, the input is ignored.
- /joy_vel:
  - Source: Joystick or game controller.
  - Priority: 80 (higher than /web_vel).
  - Timeout: 0.1 seconds.
- /nav_vel:
  - Source: Navigation stack or autonomous planner.
  - Priority: 20 (lower than /web_vel and /joy_vel).
  - Timeout: 0.1 seconds.

2. **Lock:**
   - /lock is a control mechanism to enable or disable certain inputs dynamically.
   - It can override all inputs to pause or restrict commands.
3. **Turbo Controls:**
   - /increase_turbo and /decrease_turbo allow for modifying the robot's maximum speed dynamically.
4. **Twist Mux Node:**
   - Combines all the input sources based on their priorities and timeouts.
   - Forwards the selected command to /cmd_vel, which the robot uses to move.
5. **Output:**
   - /cmd_vel: The unified velocity command that is sent to the robot's motors.

How It Works

1. **Priority-Based Selection:**
   - Twist Mux ensures that higher-priority sources (e.g., joystick) override lower-priority ones (e.g., navigation commands).
   - This is especially useful for safety, such as stopping an autonomous robot using a joystick.
2. **Timeout Handling:**
   - If an input source stops publishing commands within its timeout period, its command is ignored.
   - This prevents stale commands from affecting the robot.
3. **Dynamic Switching:**
   - By adjusting priorities or locking certain inputs, Twist Mux allows seamless switching between manual and autonomous modes.

Advantages of Twist Mux

1. Safety:
   - Prevents conflicting commands by prioritizing inputs.
   - Timeout ensures outdated commands do not control the robot.

2. Flexibility:
   - Easy to integrate multiple control methods.
   - Dynamic adjustment of priorities and locks during runtime.

3. Ease of Use:
   - Simplifies command handling in multi-source control systems.
   - Provides a structured way to manage velocity commands.

Applications

1. **Teleoperation:**
   - Allowing joystick input to override autonomous navigation in case of emergencies.
2. **Autonomous Robots:**
   - Integrating planner outputs with human intervention.
3. **Robotic Systems with Multiple Interfaces:**
   - Combining web, joystick, and AI navigation seamlessly.

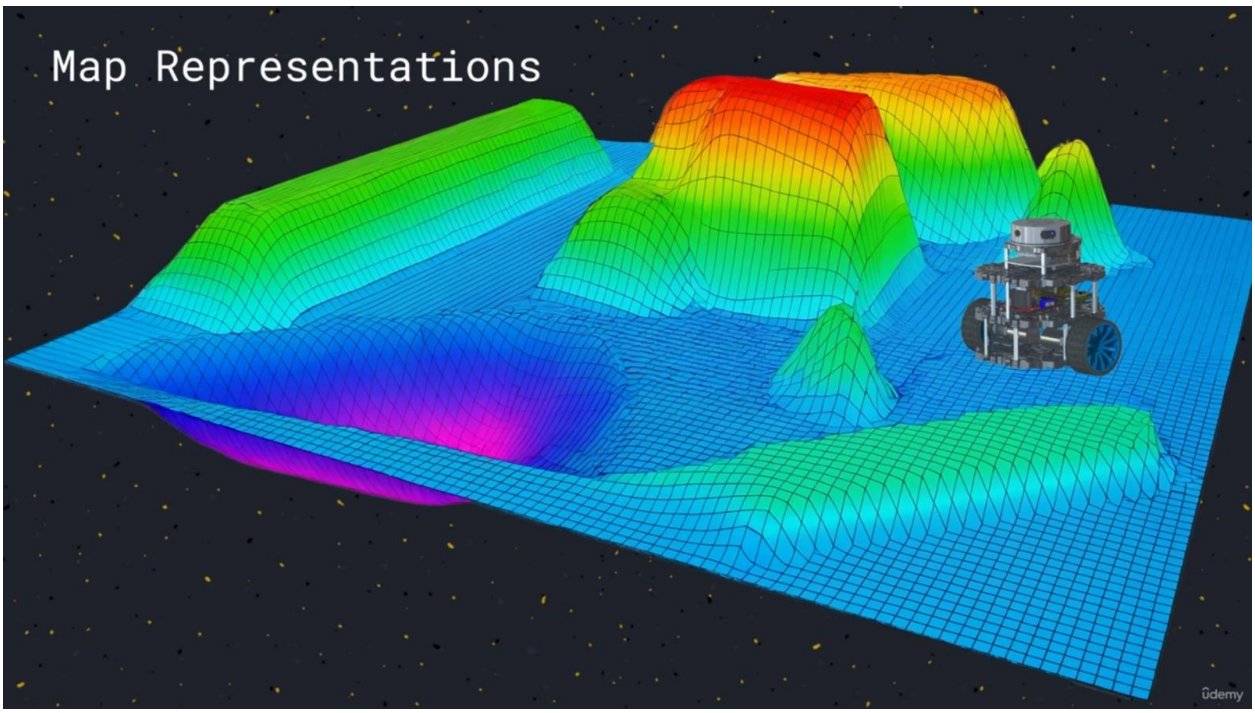


FIGURE 138

4.31.2.3 Topological Maps

Definition: A topological map represents the environment using nodes (places or regions) and edges (connections between these places). It does not provide exact location data but focuses on the relationships between different areas in the environment. It's more about the connectivity of the environment rather than the precise geometry of it.

Usage: Topological maps are ideal for high-level navigation tasks where the robot needs to move between different areas (e.g., rooms, hallways) without needing detailed spatial information. It's useful for tasks like path planning over long distances or in large environments where the exact coordinates aren't as critical.

Example: A robot in a building might recognize different rooms as nodes and the hallways as edges connecting these rooms. The robot doesn't need to know the exact position, just how the rooms are connected to each other.

4.31.2.4 Occupancy Grid

Definition: An occupancy grid is a 2D representation of the environment where the space is divided into a grid of cells, and each cell can have one of three values: occupied, free, or unknown. The occupancy grid can be used in a probabilistic or binary manner, where it indicates the likelihood of a space being occupied (obstacle) or free.

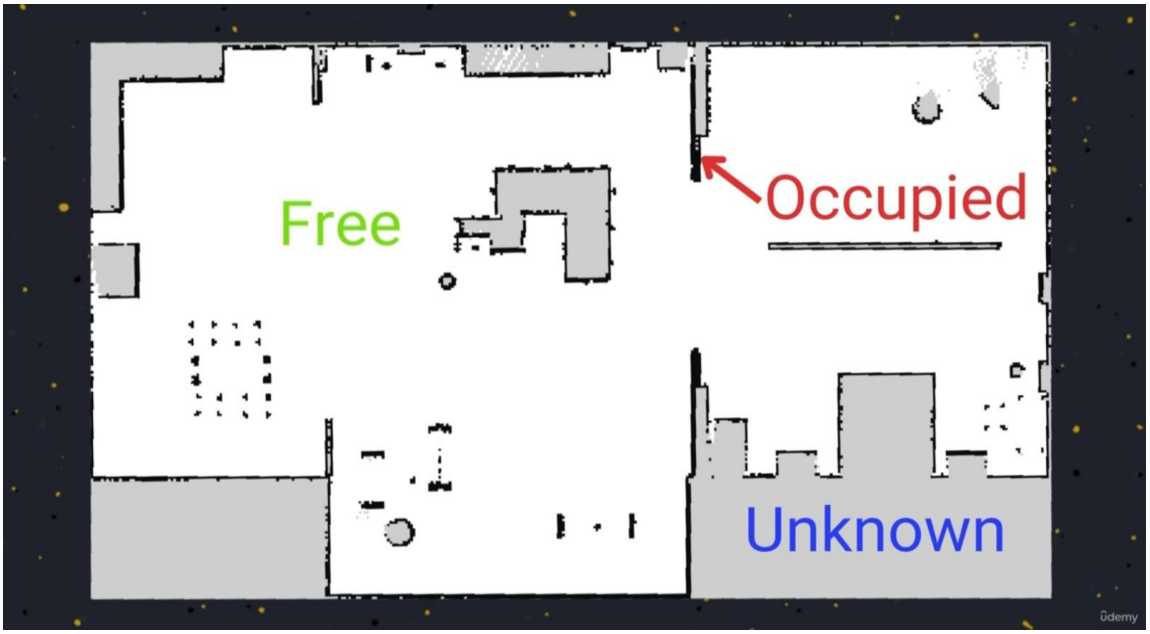


FIGURE 139

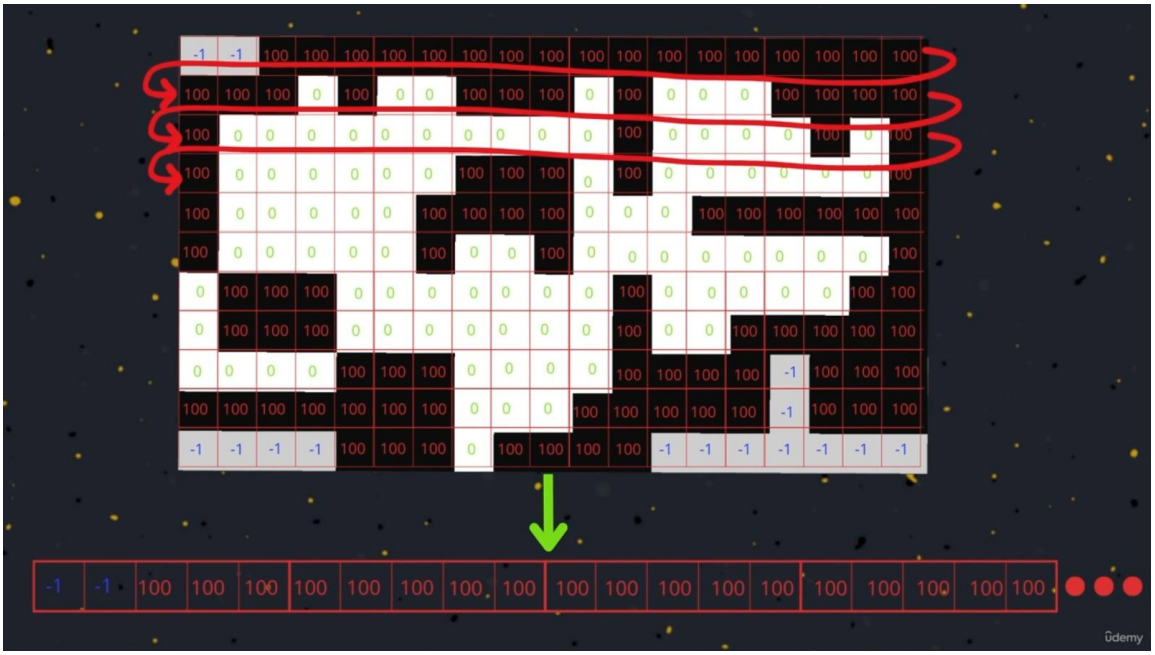


FIGURE 140

Usage: This type of map is commonly used in mobile robots for localization and obstacle avoidance. It's particularly useful when the robot needs to navigate in a known environment and avoid obstacles based on the grid data. In ROS2, an occupancy grid is often created using sensor data such as LiDAR or sonar.

Example: A robot using a LiDAR sensor to map its environment will generate an occupancy grid where occupied cells indicate obstacles (like walls or furniture) and free cells represent navigable space. Unknown cells may indicate areas where sensor data is not available.

ROS2 Message: The nav_msgs/OccupancyGrid message is used to represent this map in ROS2.

4.31.2.5 Octomap

Definition: Octomap is a 3D representation of the environment using octrees (a tree structure that recursively subdivides the space into smaller cubes). It is an extension of the occupancy grid into the third dimension, and it stores occupancy information in a hierarchical manner, where each level of the octree represents a progressively finer resolution of the environment.

Usage: Octomaps are especially useful when dealing with complex, 3D environments where obstacles are not only on the ground but also above, such as in warehouses, urban environments, or environments with varying heights. It provides better data for navigation and planning in 3D spaces compared to traditional 2D occupancy grids.

Example: A robot navigating in a warehouse might use an octomap to represent the environment in 3D, where it can account for obstacles on the floor as well as above the robot, such as shelves or overhangs.

ROS2 Message: The octomap_msgs/Octomap message is used to represent octomap data in ROS2.

4.31.2.6 Voxel Grids

Definition: A voxel grid is similar to the occupancy grid but works in 3D space, dividing the environment into small cubic cells called voxels. Each voxel can hold occupancy data, indicating whether the space is free, occupied, or unknown. It's essentially a 3D grid that represents the environment in discrete chunks, allowing for more complex representations compared to 2D grids.

Usage: Voxel grids are useful for applications requiring 3D mapping, such as for autonomous drones or robots that need to navigate in environments where obstacles exist in all three dimensions (e.g., flying around buildings or avoiding overhead obstacles).

Example: A robot in a 3D environment (such as a drone flying inside a building) would use a voxel grid to understand obstacles not just on the ground but also in the air (e.g., ceilings, light fixtures, or other drones).

ROS2 Message: Voxel grids can be represented using the sensor_msgs/PointCloud2 message or custom voxel grid representations.

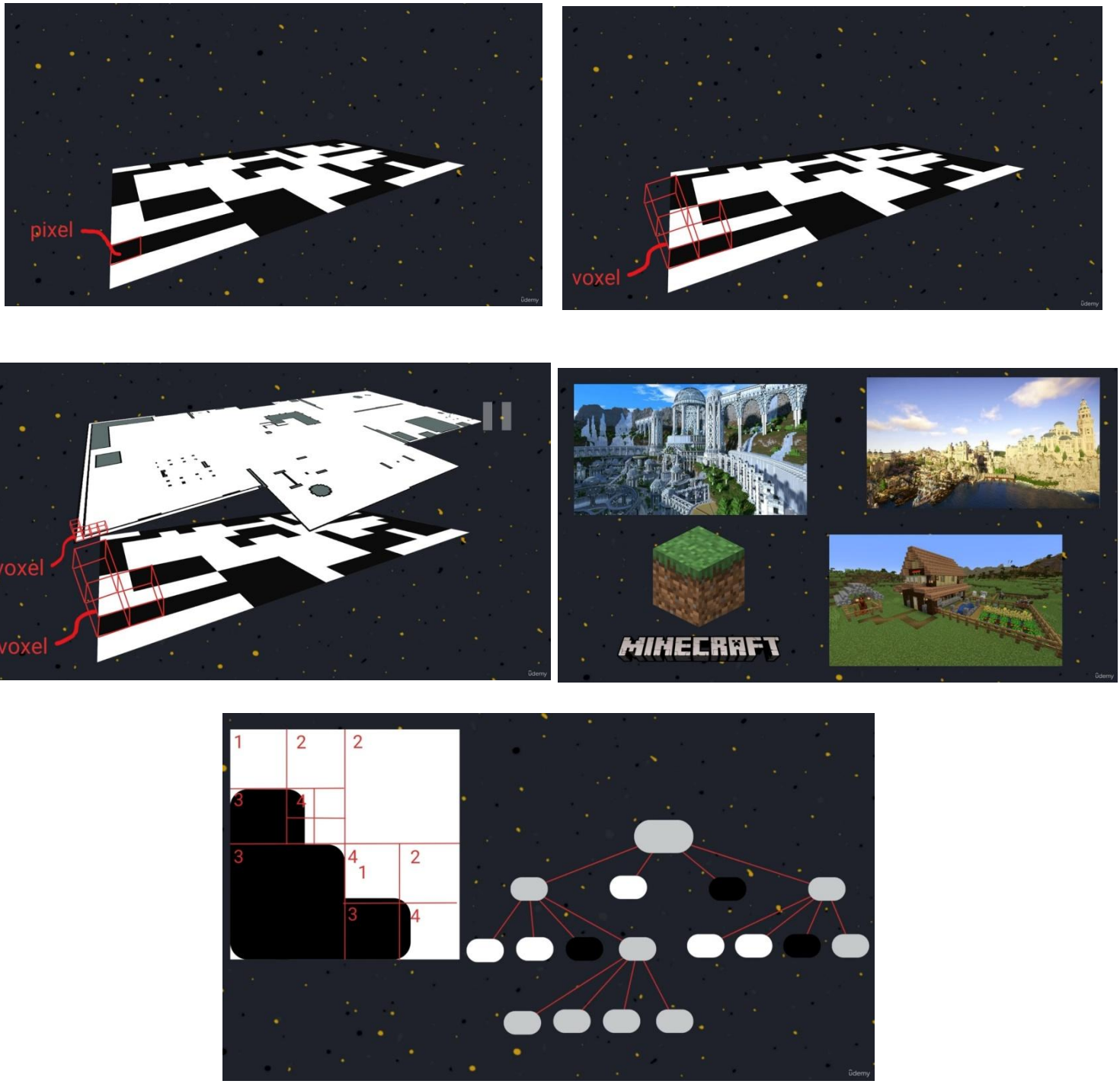


FIGURE 141

4.31.2.7 Mapping with Known Poses

Definition: Mapping with known poses involves using the robot's known position at each point in time to build a map of the environment. This is typically done using Simultaneous Localization and Mapping (SLAM) techniques, where the robot's position (pose) is tracked, and sensor data is collected at that specific pose to incrementally build a map.

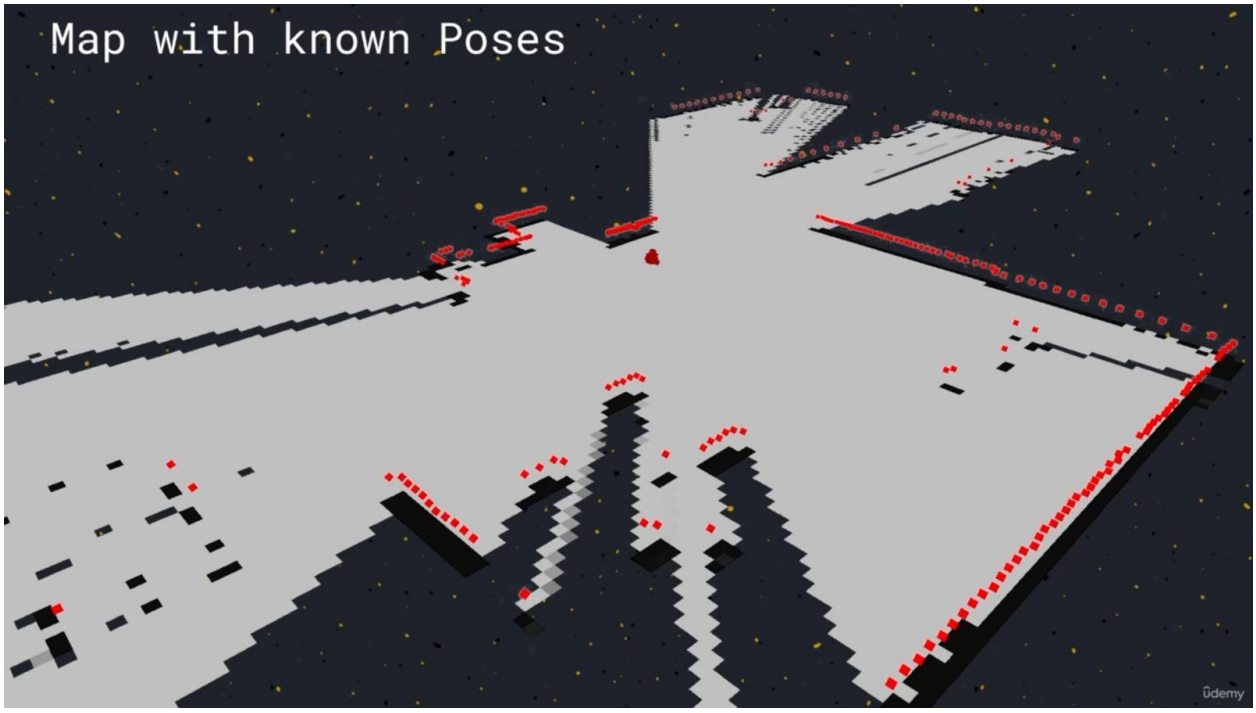


FIGURE 142

How It Works:

- The robot moves through the environment and uses sensors (like LiDAR, cameras, or sonar) to gather data.
- As the robot moves, its position (pose) is known or estimated. This could come from odometry, GPS (for outdoor robots), or other localization methods like AMCL (Adaptive Monte Carlo Localization).
- With each pose, the sensor data is used to update the map by marking areas as occupied (obstacles) or free (open space).

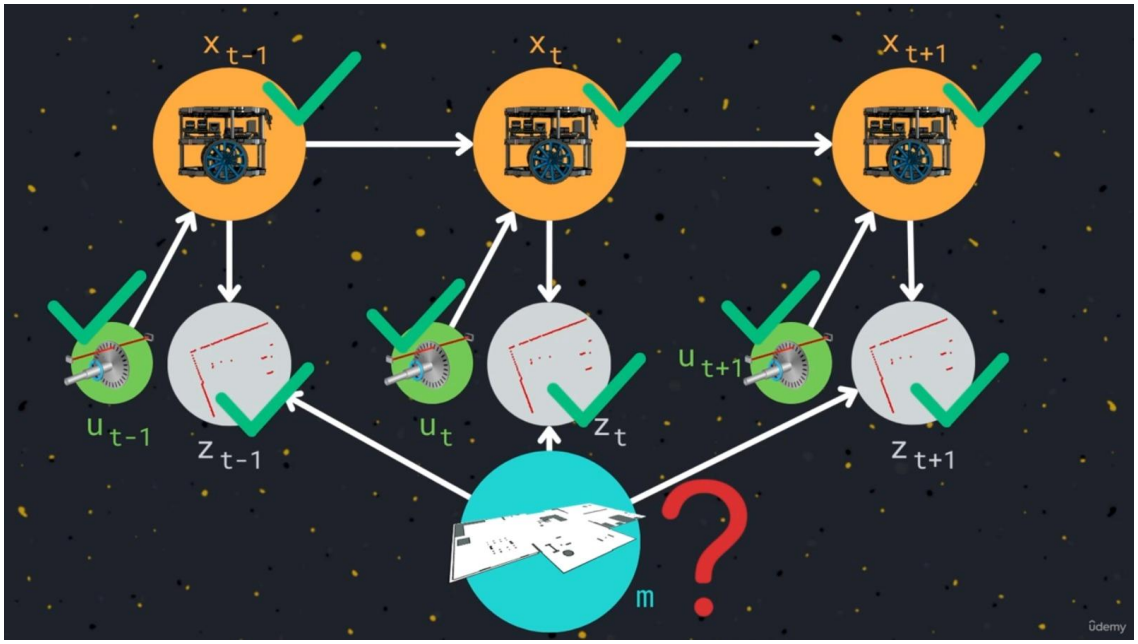


FIGURE 143

Steps:

1. The robot starts at an initial known position (pose).
2. As it moves, it periodically records the current pose (position and orientation).
3. The robot collects sensor data and updates the map based on its known position.
4. The robot continues moving and mapping the environment while adjusting the map based on new sensor readings and changes in the robot's pose.

Applications:

This is particularly useful when you always have a pre-existing map or you're confident about the robot's position, and you want to use this data to build and refine the map. It is common in indoor robots and robots equipped with precise localization systems.

4.31.2.8 Occupancy Grid Mapping

Definition: Occupancy Grid Mapping is a technique used to represent the environment in a grid format, where each grid cell can hold information about whether the area is free or occupied. This map is used to help robots navigate and avoid obstacles.

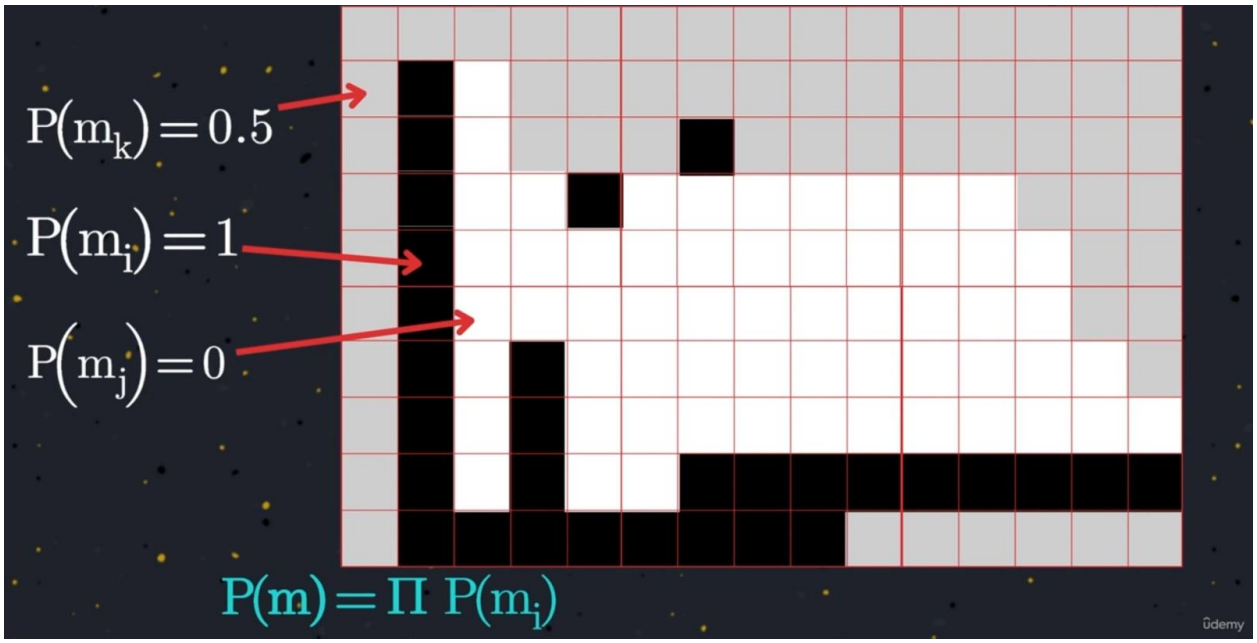


FIGURE 144 OCCUPANCY GRID MAPPING

How It Works:

- The environment is divided into a 2D, or 3D grid, where each cell represents a small area of the environment.
- The robot moves around the environment, collecting data from its sensors (like LiDAR, sonar, or cameras) and updating the grid cells.
- Each cell is marked with values representing the occupancy probability, indicating whether that space is occupied (an obstacle) or free (open space).

Grid Representation: Each grid cell can have a value:

- Occupied (1): The area is occupied by an obstacle (e.g., a wall, chair).
- Free (0): The area is free of obstacles, meaning the robot can safely navigate through that space.
- Unknown (0.5 or a neutral value): The occupancy of the area is unknown because no sensor data is available for that region yet.

Steps:

- The robot collects data from its sensors and updates the occupancy grid with the sensor readings.
- If the robot detects an obstacle in a region, the corresponding grid cell is marked as occupied.
- If the region is clear, the cell is marked as free.
- Over time, the map is refined as more data is collected.

Applications:

1. This method is very effective for indoor robots and autonomous vehicles in environments with clear boundaries and obstacles.
2. It helps robots localize themselves and plan paths while avoiding obstacles.

4.31.2.9 Probabilistic Occupancy Grid Mapping

Definition: Probabilistic Occupancy Grid Mapping* extends the basic occupancy grid by adding a probabilistic model to represent the uncertainty in sensor readings. Rather than simply marking cells as occupied or free, each grid cell has a probability that represents how likely it is to be occupied or free, based on the sensor data.

How It Works:

- Instead of just updating grid cells to binary values, probabilistic occupancy grids use probabilities to represent uncertainty in the sensor readings.
- This approach uses Bayesian update rules to continuously update the occupancy probability of each grid cell based on new sensor measurements. This is crucial in environments where sensors are noisy or imperfect.

LASER MODEL:

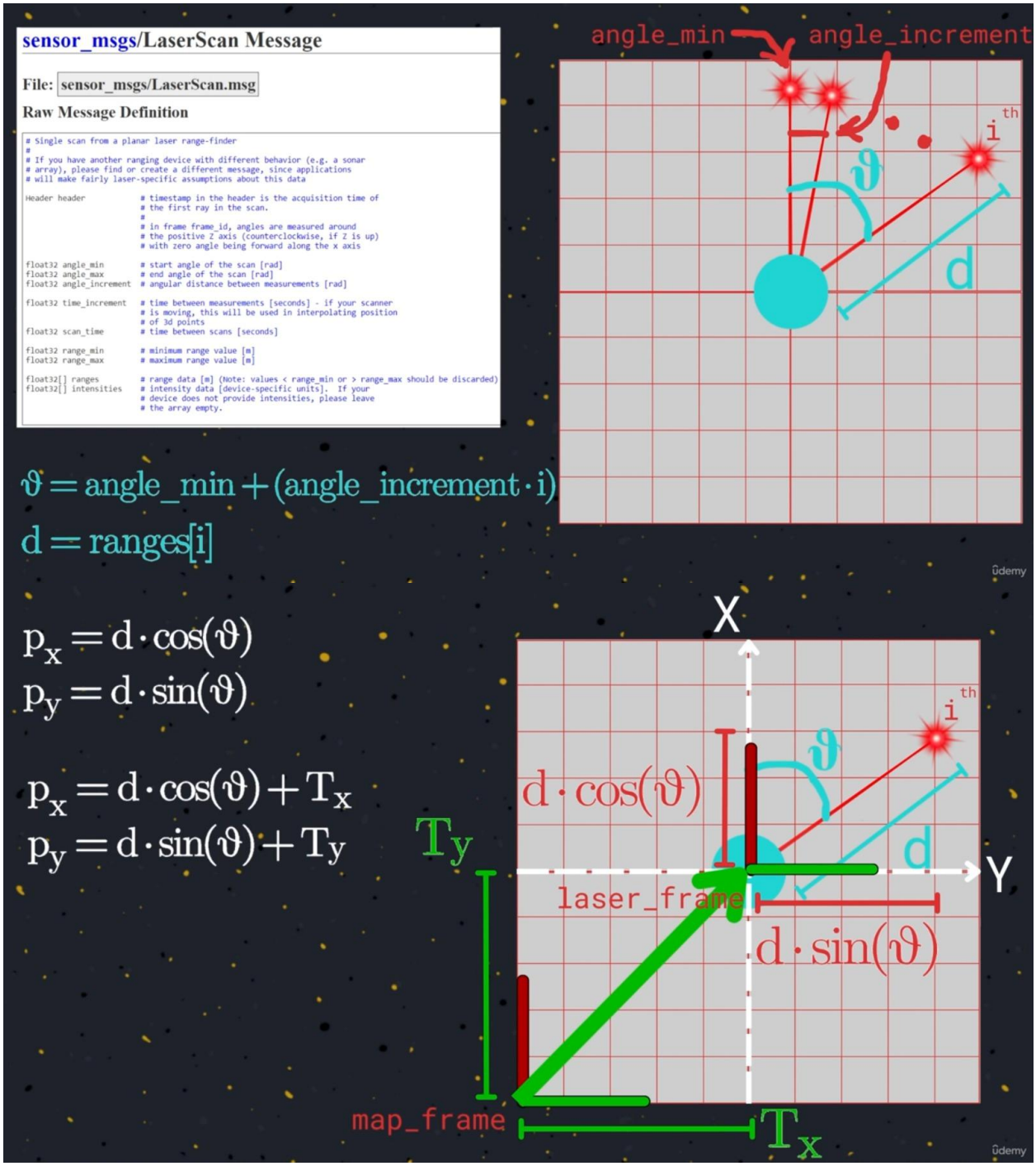


FIGURE 145 LASER MODEL

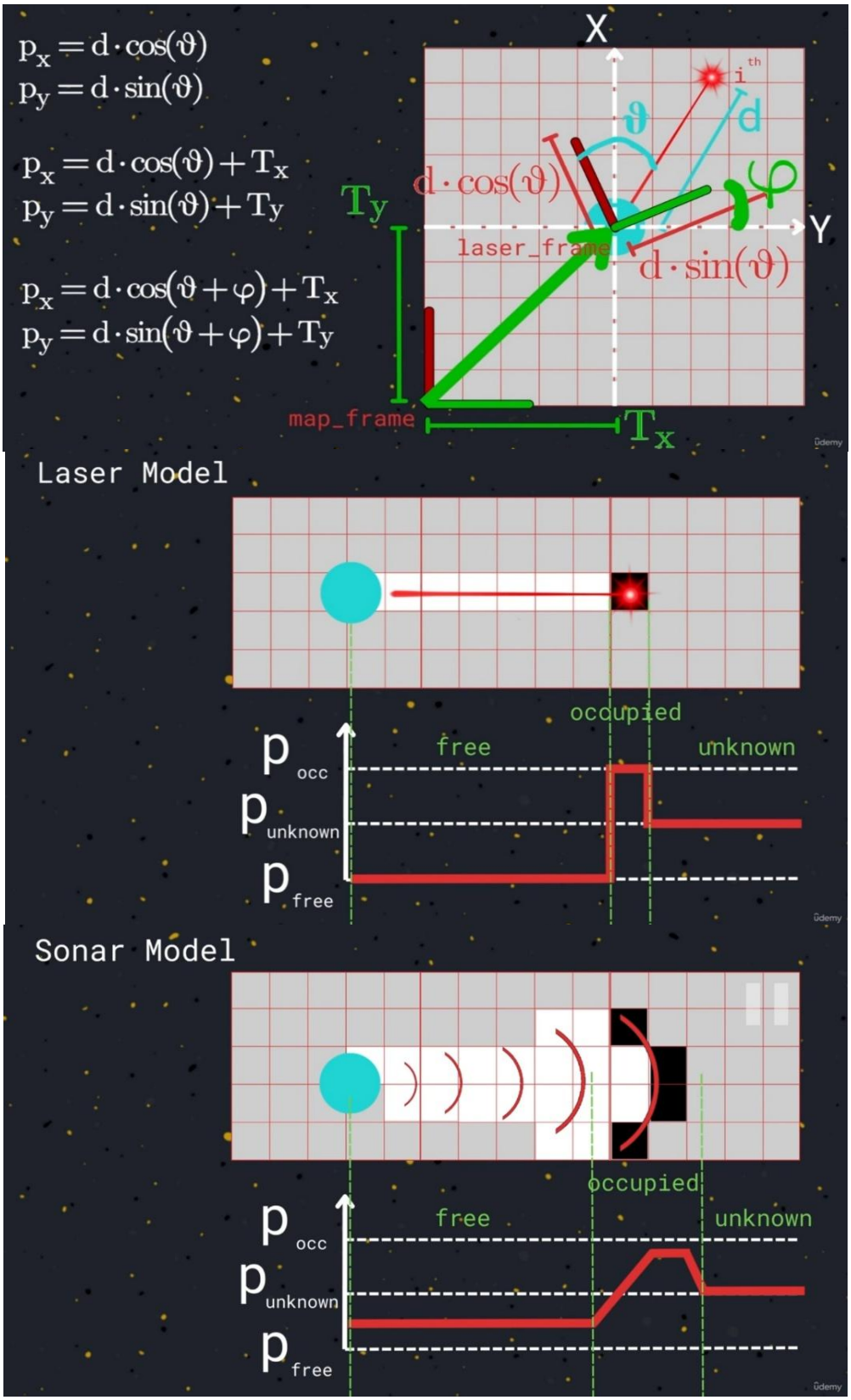


FIGURE 146 LASER MODEL

Bresenham's Line Algorithm

Bresenham's Line Algorithm is an efficient method used for drawing a straight line between two points on a raster grid (such as pixels on a screen) by incrementally plotting points along the line. This algorithm uses only integer operations, which makes it computationally efficient, especially for systems with limited processing power.

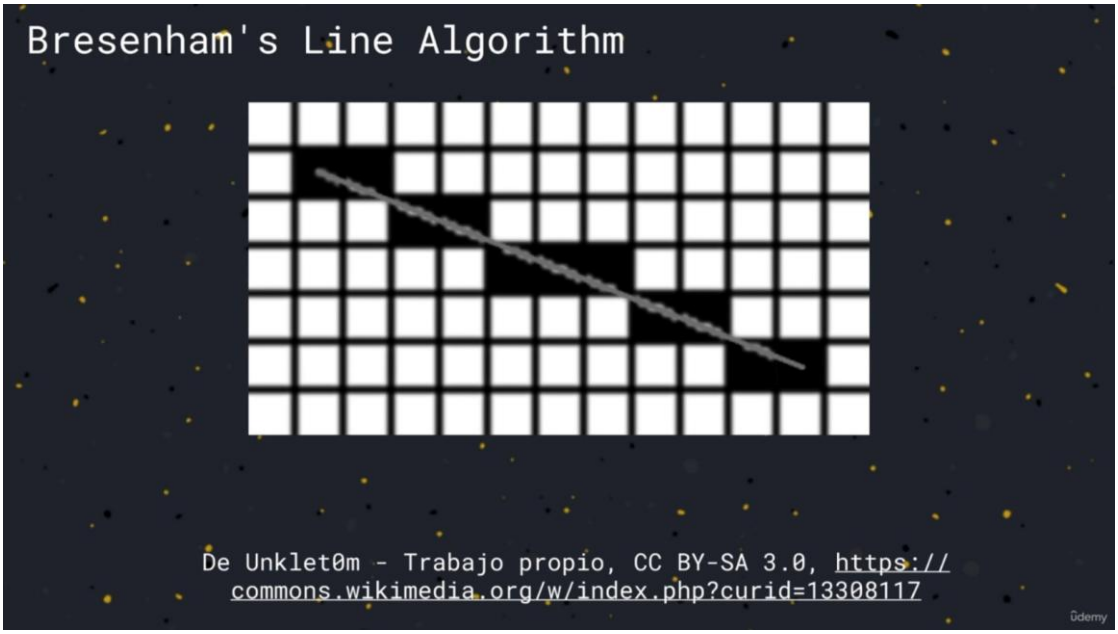


FIGURE 147 BRESENHAM'S LINE ALGORITHM

Basic Idea:

The goal of Bresenham's algorithm is to approximate the ideal straight line between two given points. Instead of using floating-point arithmetic (which can be computationally expensive), Bresenham's algorithm uses only integer additions, subtractions, and comparisons, which are much faster.

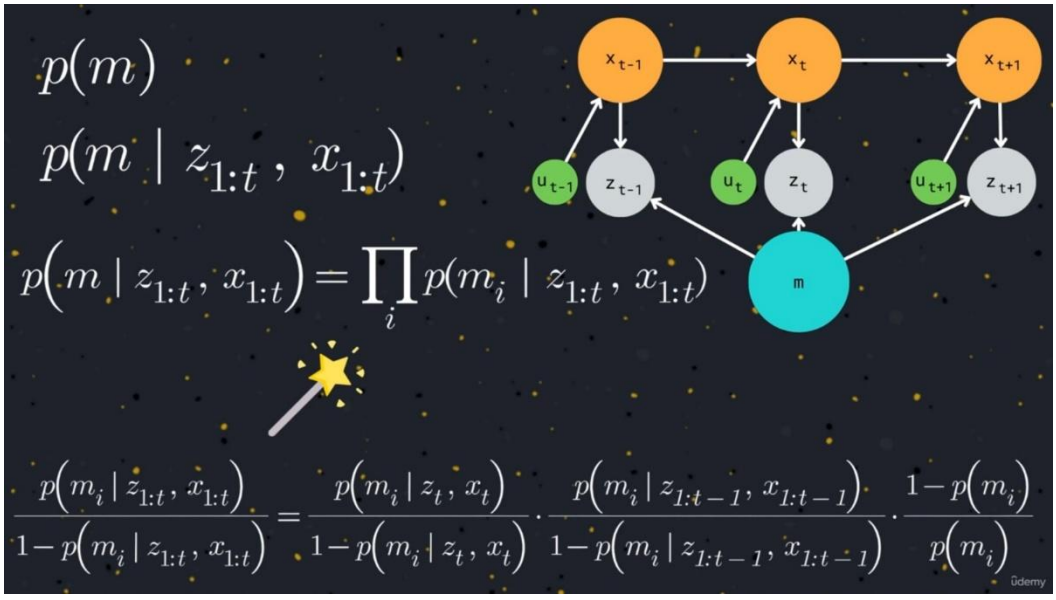


FIGURE 148

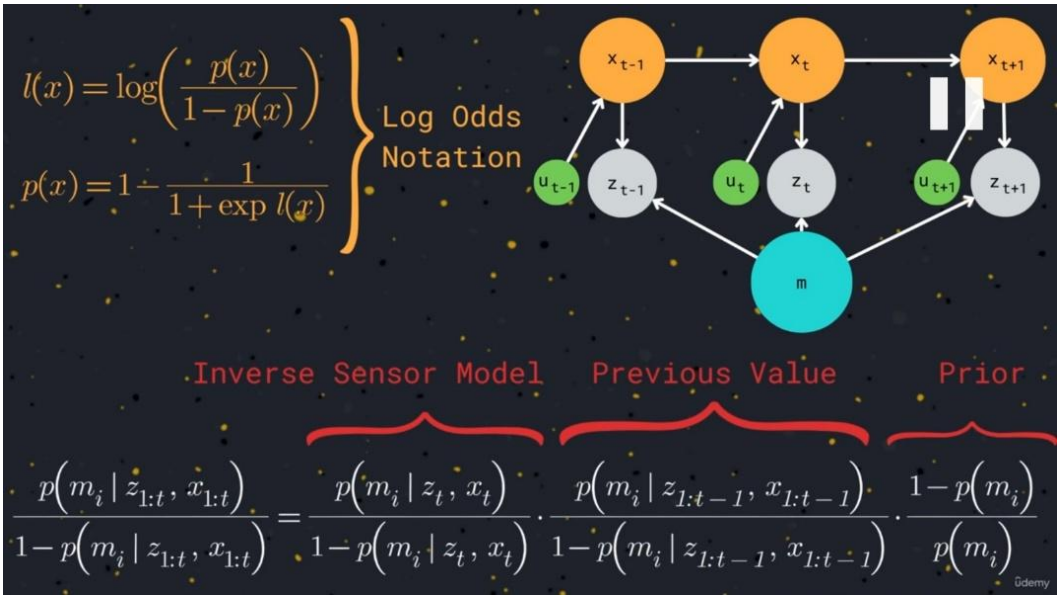


FIGURE 149

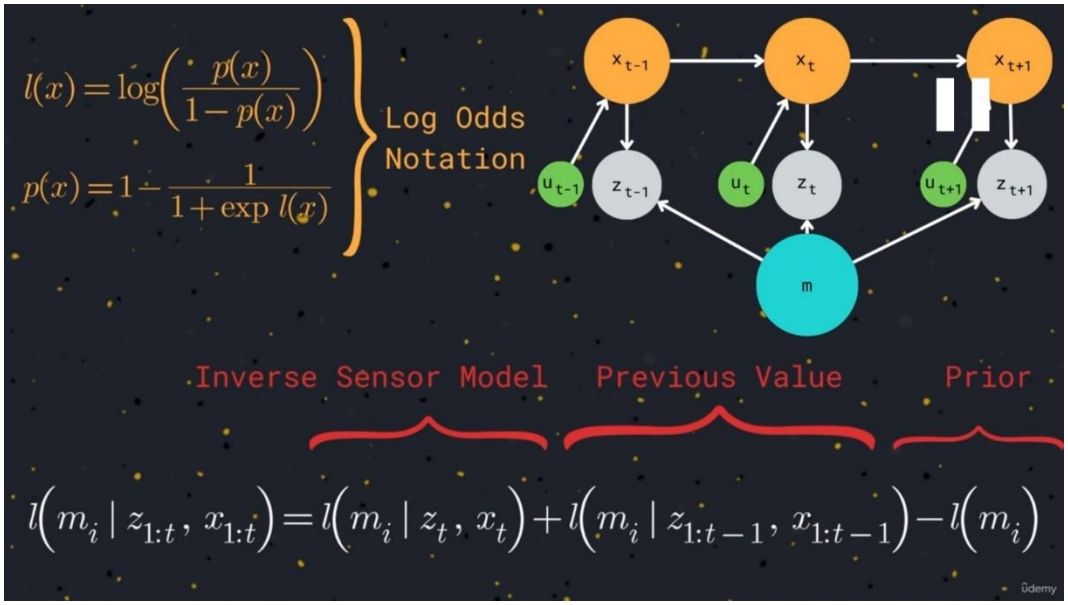


FIGURE 150

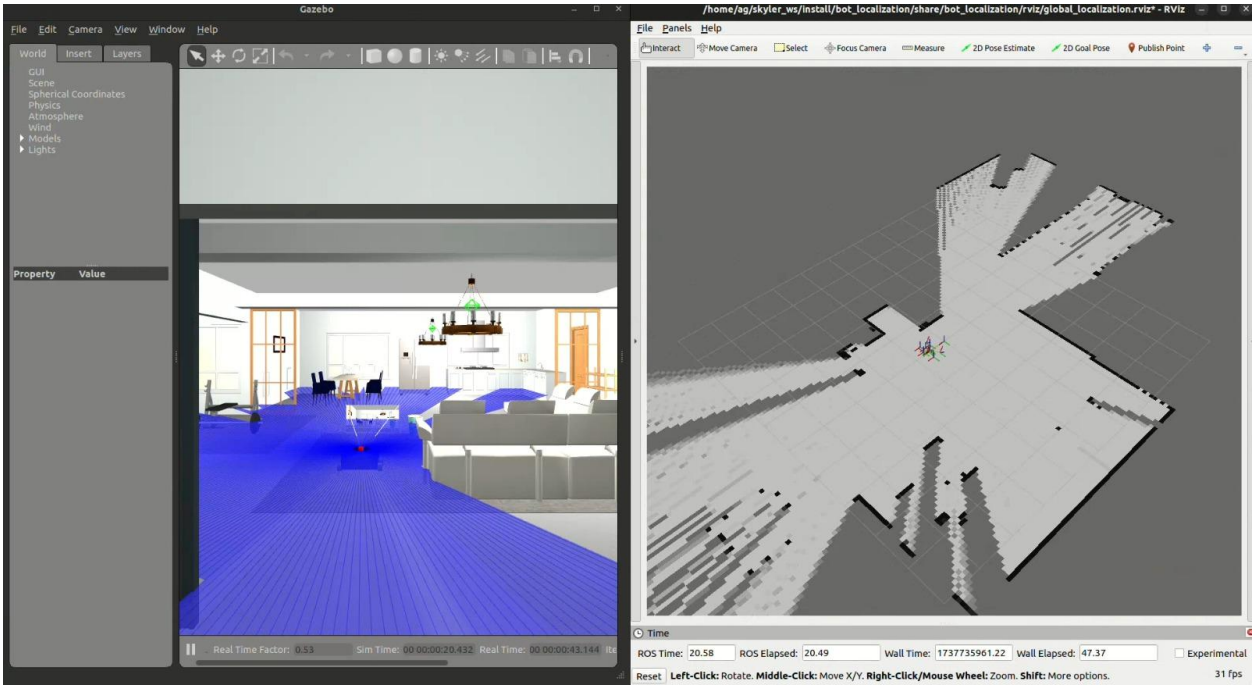


FIGURE 151 LAB MAPPING WITH KNOWN POSITION

4.31.2.10 Localization with Known Map:

Localization with a known map involves using an already available map of the environment to determine the robot's current position. The robot utilizes the map, along with its sensors (e.g., LiDAR, cameras, IMU), to match its sensor data with features on the map. This approach is often used in situations where the map of the environment is already pre-existing or built beforehand.

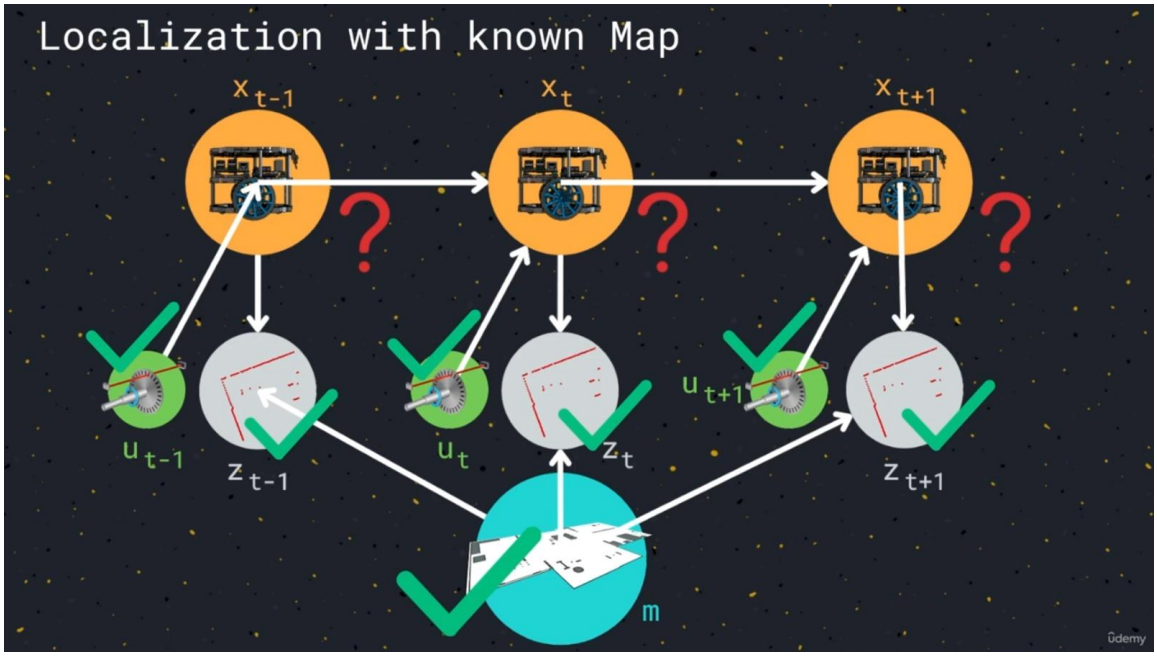


FIGURE 152

How It Works:

The robot compares its sensor measurements (such as distances to walls or obstacles) to a pre-existing map and computes its position within the map. Techniques like ICP (Iterative Closest Point) or scan matching can be used to refine the robot's position.

Applications: Indoor environments (e.g., warehouses, buildings) where the map is relatively static and well-known.

4.31.2.11 Map-based Localization:

Map-based localization refers to using a map of the environment to estimate the robot's position and orientation. The map can be either pre-built or created dynamically using techniques such as SLAM (Simultaneous Localization and Mapping) .

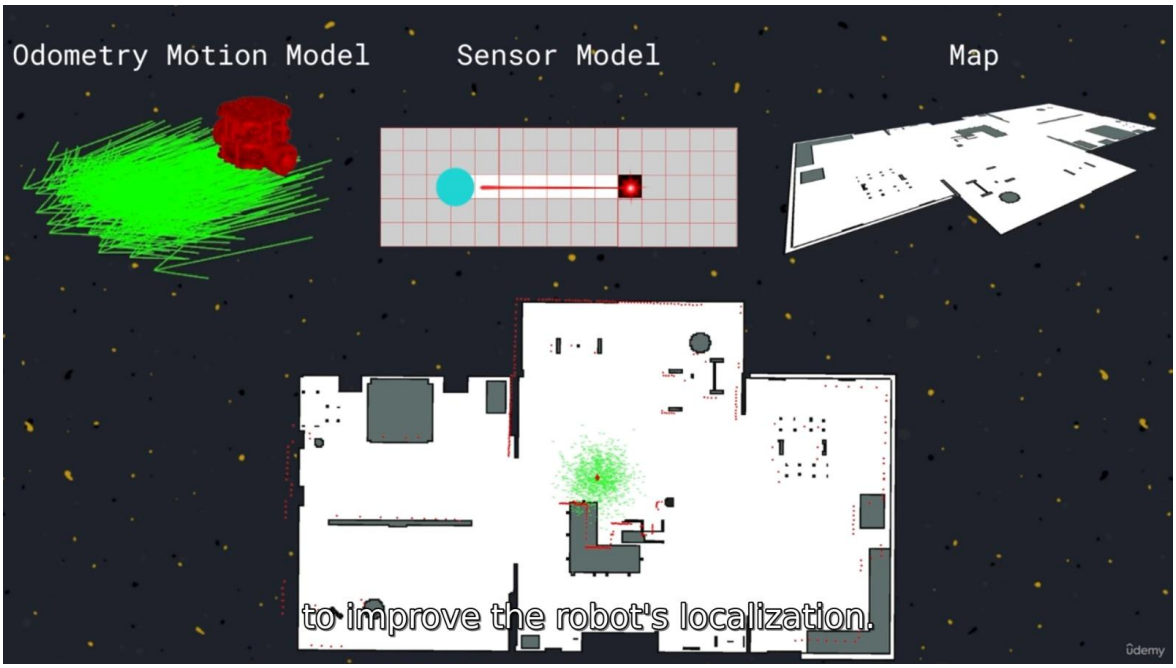


FIGURE 153 MAP-BASED LOCALIZATION

How It Works: The robot uses sensor data (such as from laser scanners or cameras) to match its observations with features in the map. For example, it may use known landmarks or obstacles in the environment to estimate its position relative to the map.

Techniques:

- **Matching sensor data to the map:** This could involve techniques like scan matching, where the robot compares the sensor's current view of the environment with the map.
- **Global localization:** This estimates the robot's position in the whole map.
- **Local localization:** This focuses on tracking the robot's position relative to its previous position.

Applications: Path planning, autonomous navigation, and robotic assistance in structured environments.

4.31.2.12 Single and Multiple Hypothesis Localization:

- **Single Hypothesis Localization:**
  - In single hypothesis localization, the robot assumes a single possible position in the map at any given time. It is typically simpler but assumes that the robot's position can be uniquely determined.
  - **How It Works:** After each sensor measurement, the robot updates its belief about its position, refining its location by eliminating impossible positions based on the current sensor data.
- **Multiple Hypothesis Localization:**
  - In multiple hypothesis localization, the robot maintains multiple possible positions (hypotheses) simultaneously. This is useful when the robot's position is uncertain, and it's unclear where exactly it is in the map.
  - **How It Works**: The robot maintains a set of hypotheses, each representing a possible position. Over time, as the robot gets more sensor data, some hypotheses are confirmed, and others are discarded based on the data.

Applications: Environments with ambiguous or noisy sensor data, such as environments with poor visibility or where the robot may be near areas with similar features.

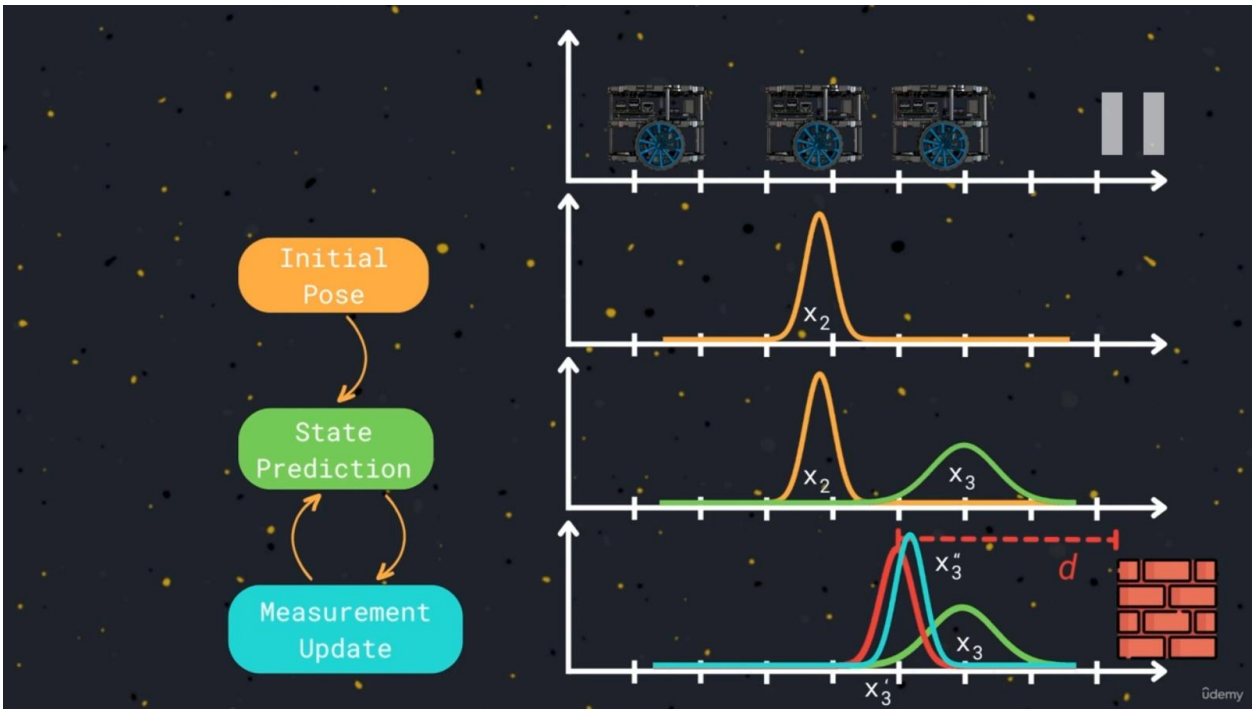


FIGURE 154 SINGLE AND MULTIPLE HYPOTHESIS LOCALIZATION:

4.31.2.13 Markov Localization:

Markov localization is a probabilistic method for localization where the robot's position is represented by a probability distribution over the map.

How It Works: The robot's belief about its position is updated using Bayesian updates. The robot combines its previous belief (position estimate) with sensor data to update its current belief about where it is. The Markov assumption simplifies the process by assuming that the current belief only depends on the previous belief and the current sensor data, not the full history of measurements.

Mathematical Basis: The localization problem is framed as a Markov decision process (MDP), where each position has an associated probability. The update rule involves combining the prior belief (a probability distribution) with a likelihood function based on sensor readings.

Applications: Navigation in uncertain environments, localization with noisy sensors.

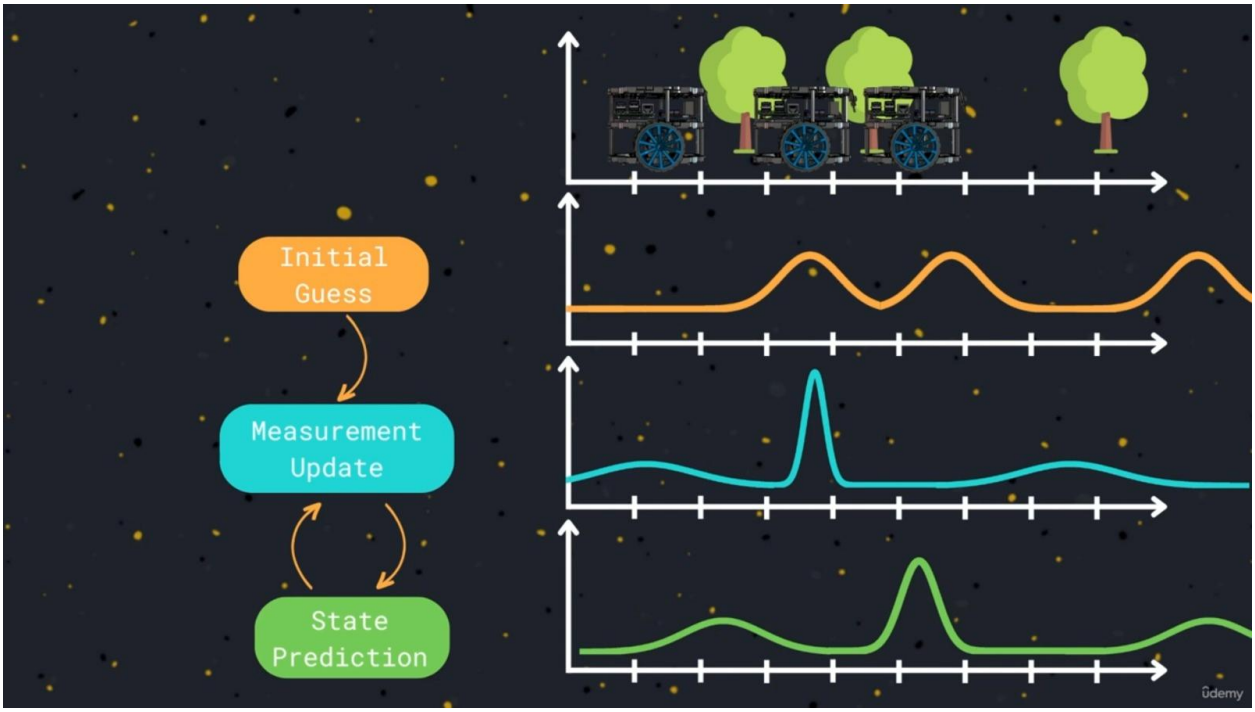


FIGURE 155 MARKOV LOCALIZATION

4.31.2.14 Randomized Sampling:

Randomized sampling techniques are used to estimate the robot's position by sampling from possible locations on the map.

How It Works: The robot generates a set of random samples or hypotheses about its position, then refines those hypotheses over time by comparing the sensor data with the expected data from those hypotheses. This approach doesn't require a deterministic method and can be useful in highly uncertain environments.

Applications: Used in Monte Carlo Localization and other probabilistic localization techniques.

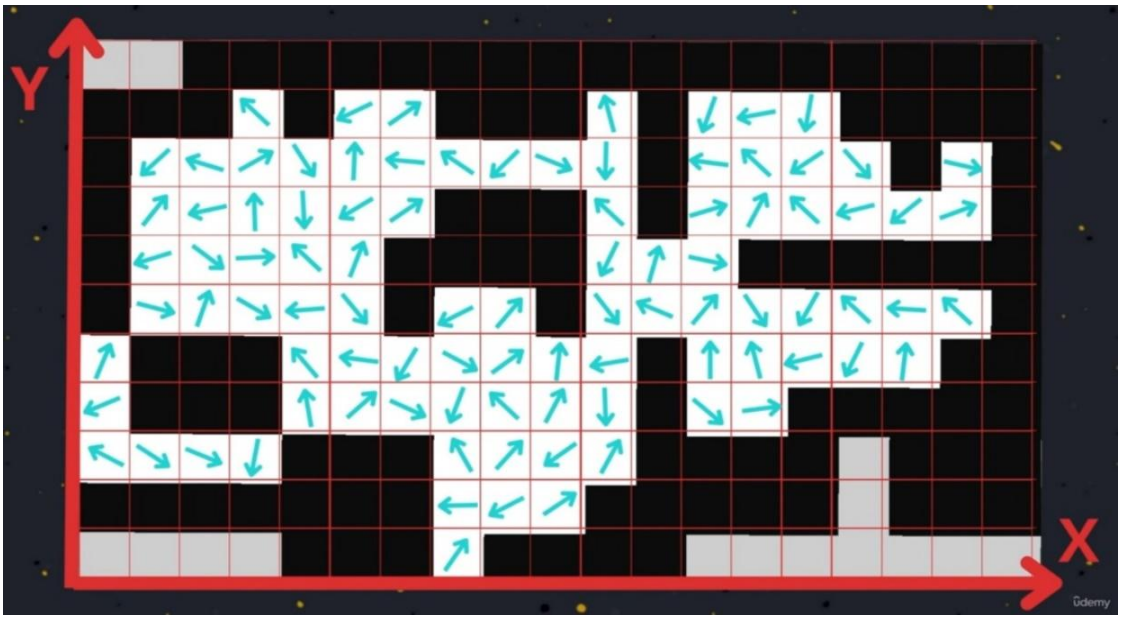


FIGURE 156 RANDOMIZED SAMPLING:

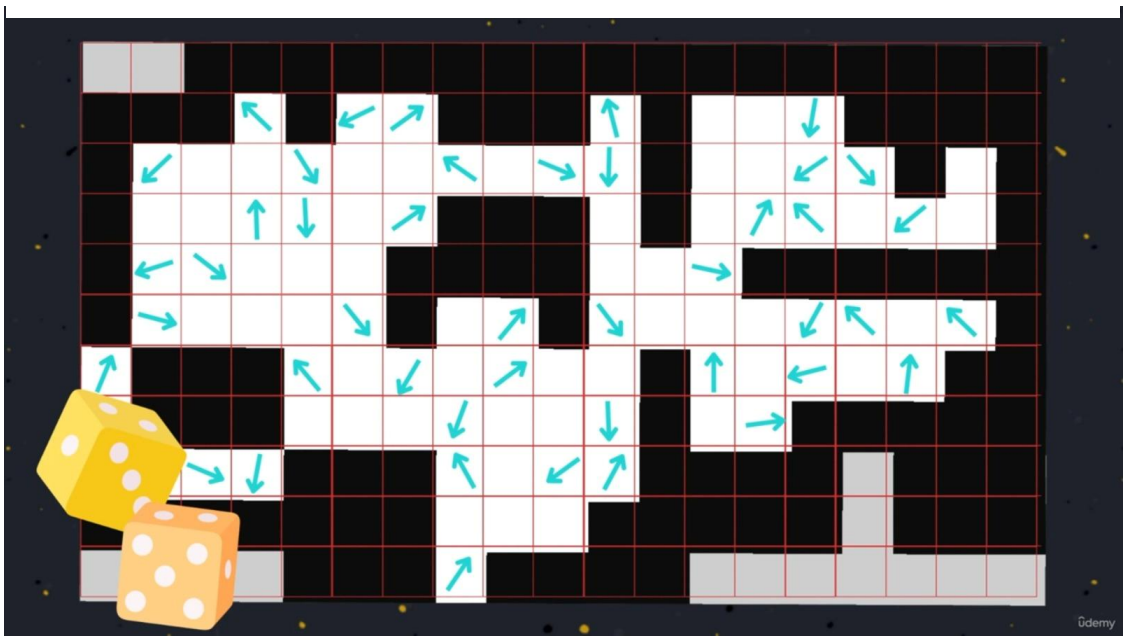


FIGURE 157

4.31.2.15 Monte Carlo Localization (MCL):

Monte Carlo Localization is a probabilistic localization technique that uses a set of random samples (particles) to represent the robot's belief about its position.

How It Works:

- The robot represents its belief by a set of particles, each representing a hypothesis of the robot's position in the map.
- The particles are propagated based on motion updates, and the likelihood of each particle is updated using sensor measurements.
- Over time, particles are resampled according to their likelihood (more likely particles are kept, less likely particles are discarded).

Key Steps:

1. **Initialization:** Generate a set of particles that represent possible positions of the robot.
2. **Prediction:** Move each particle based on the robot's motion (e.g., odometry data).
3. **Correction:** Update the likelihood of each particle based on sensor data (e.g., distance to obstacles).
4. **Resampling:** Resample particles based on their likelihood to focus on the more likely positions.

Applications: Widely used in mobile robots for global localization (when the robot doesn't know its initial position) and localization in dynamic environments.

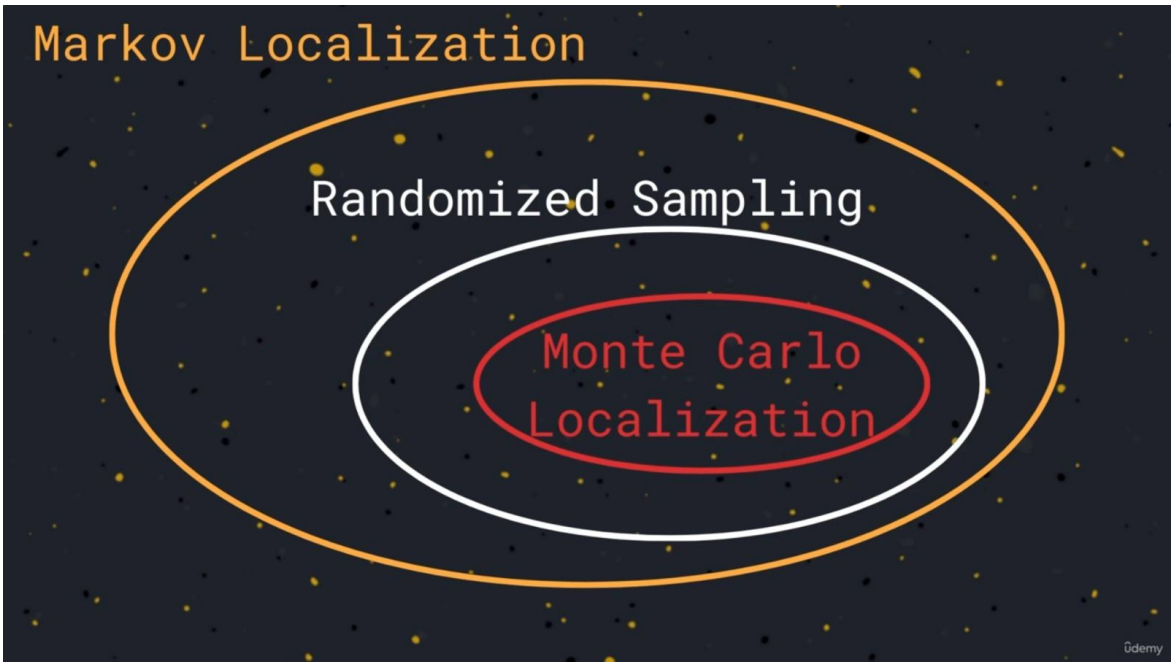


FIGURE 158

4.31.2.16 Nav2 AMCL (Adaptive Monte Carlo Localization):

AMCL is a specific implementation of Monte Carlo Localization (MCL) for ROS 2, used to estimate a robot's pose in a known map by maintaining a set of particles that represent possible robot locations.

How It Works:

- AMCL operates by using a *particle filter* (based on MCL) to localize the robot within a known map.
- It continually updates the particles based on odometry and sensor data (e.g., laser scans) to refine the robot's position.
- AMCL adapts to the robot's environment by adjusting the weight of the particles based on how well they match the actual sensor readings, improving the accuracy of the localization.

Key Features:

- Efficient resampling: AMCL adjusts the particle filter over time to focus on the most likely positions.
- Handles uncertainty: It's robust in environments with noisy sensors and ambiguous positions.

Applications: AMCL is a core component in ROS 2 for enabling precise localization and navigation in mobile robots, particularly in indoor environments.

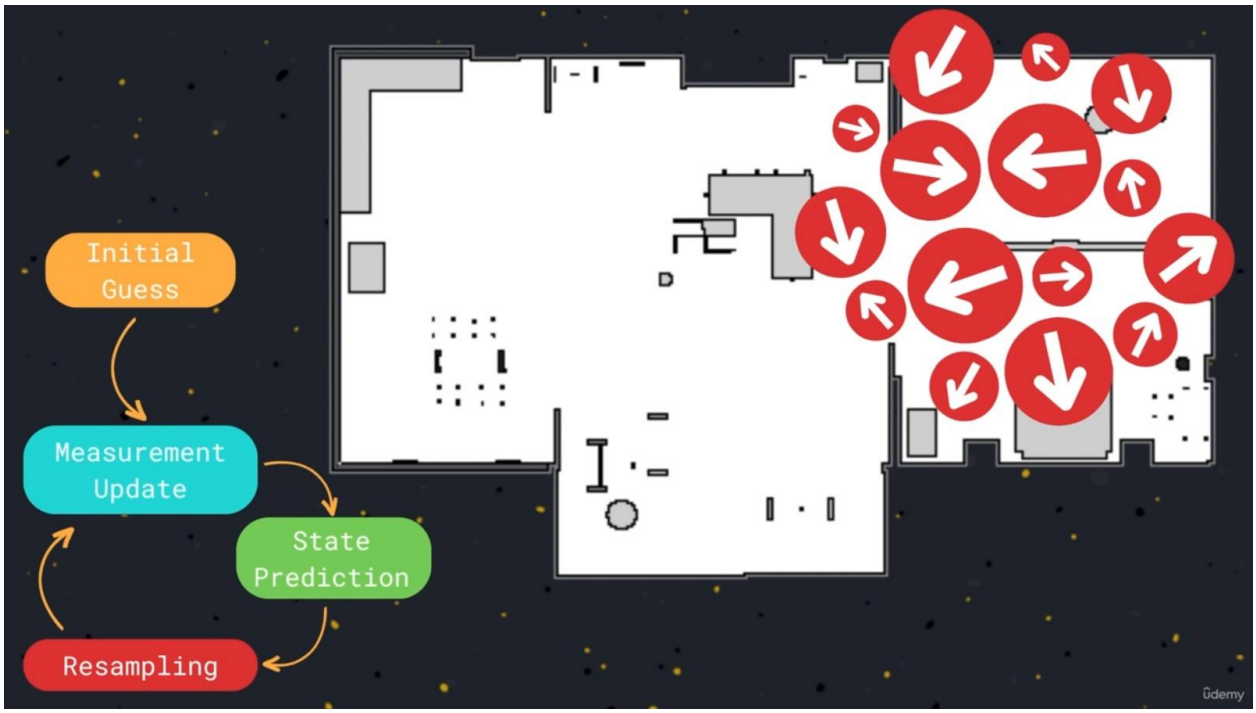


FIGURE 159

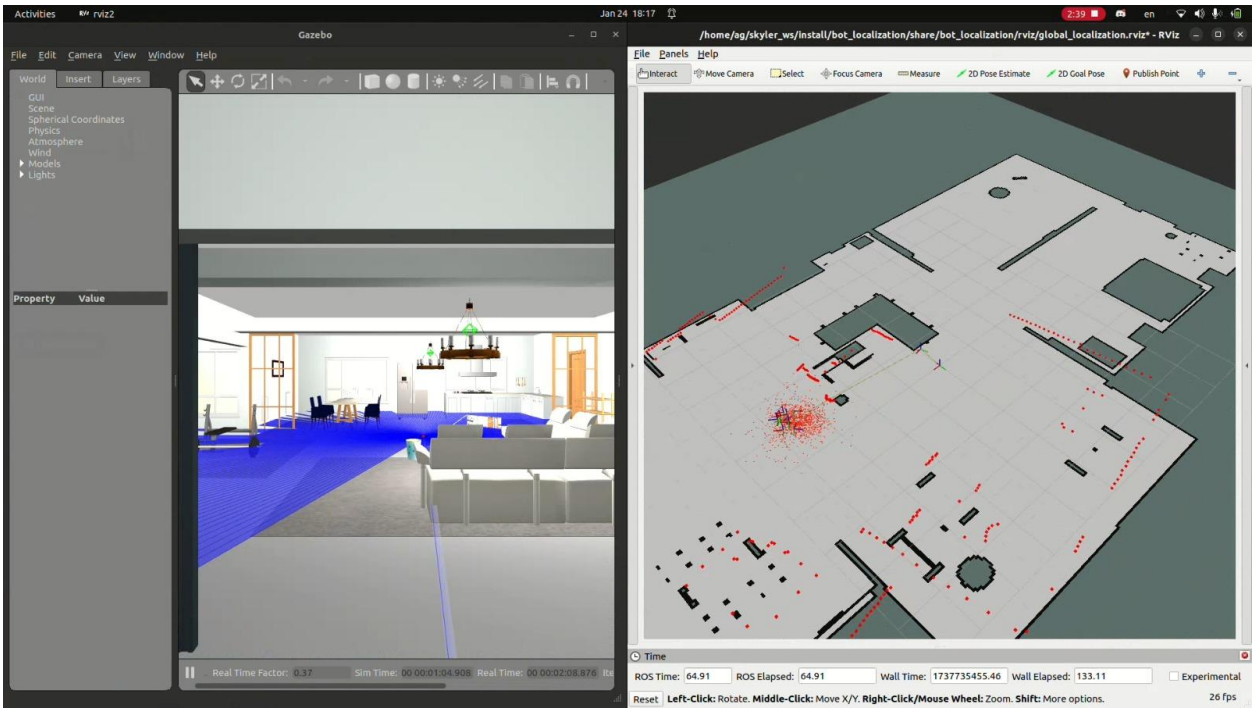


FIGURE 160 LAB LOCALIZATION

4.31.3 SLAM

Simultaneous Localization and Mapping (SLAM) is one of the foundational concepts in robotics, especially for autonomous navigation in unknown or partially known environments. It refers to the process where a robot builds a map of its surroundings while simultaneously determining its location within that map. The ability to perform SLAM is critical for mobile robots that operate autonomously without relying on GPS or an already mapped environment.

Why Another Mapping Approach?

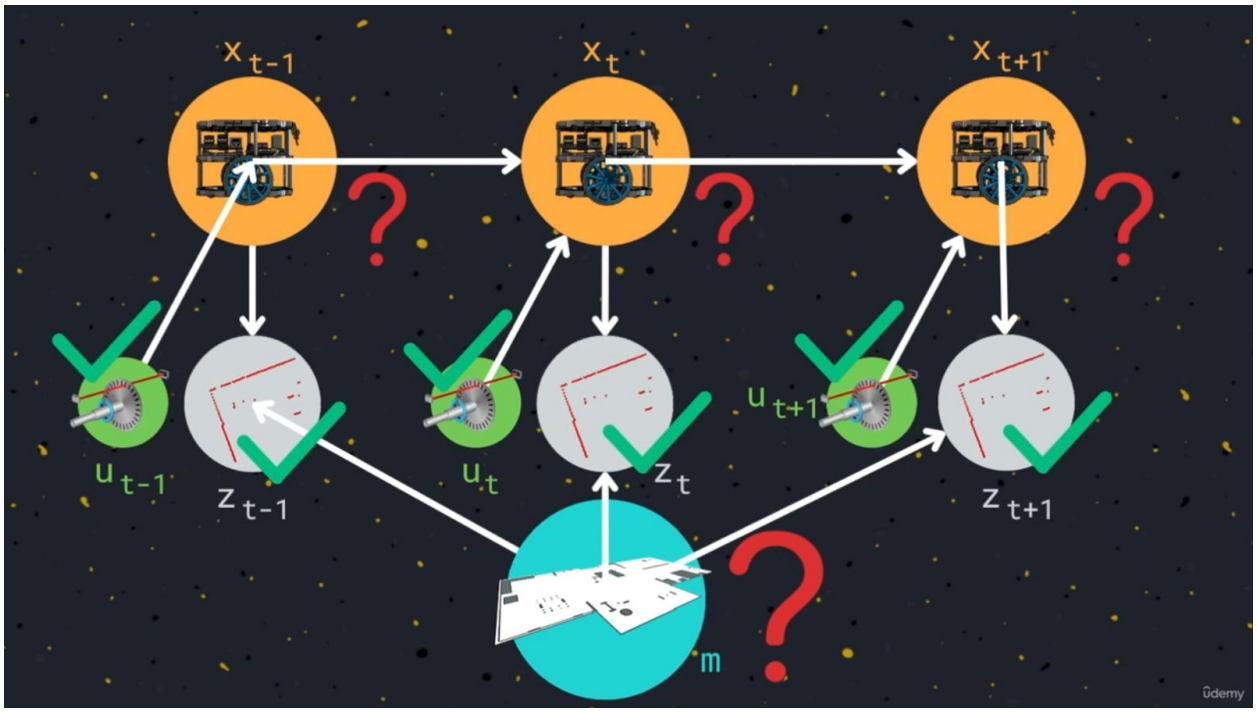


FIGURE 161

Traditional mapping methods rely on pre-existing maps, where the robot simply locates itself within a known map. However, this approach is impractical in dynamic or unexplored environments. SLAM is needed in these cases to allow a robot to both map the environment and localize itself simultaneously.

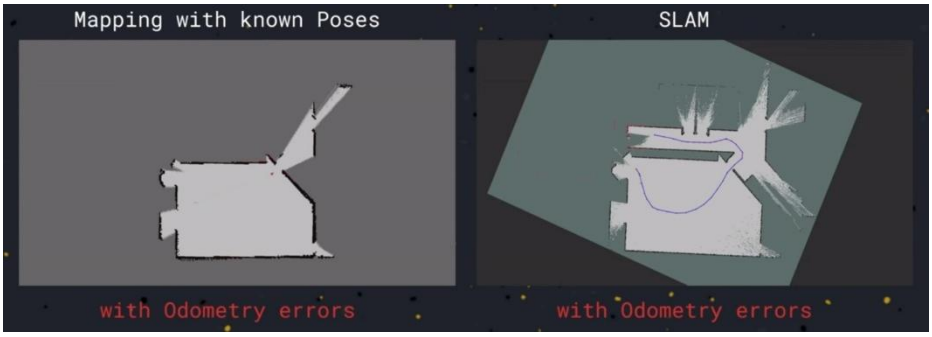


FIGURE 162

Challenges:

- **Dynamic environments**: Environments change over time, so the map needs to be updated continuously.
- **Uncertainty**: Sensors are often noisy, so both localization and mapping must account for error and uncertainty.
- **Scalability**: As robots explore larger areas, maintaining accurate maps and positions becomes increasingly difficult.

SLAM allows robots to overcome these challenges by creating a map while simultaneously keeping track of where it is on that map, even when there are no pre-existing references.

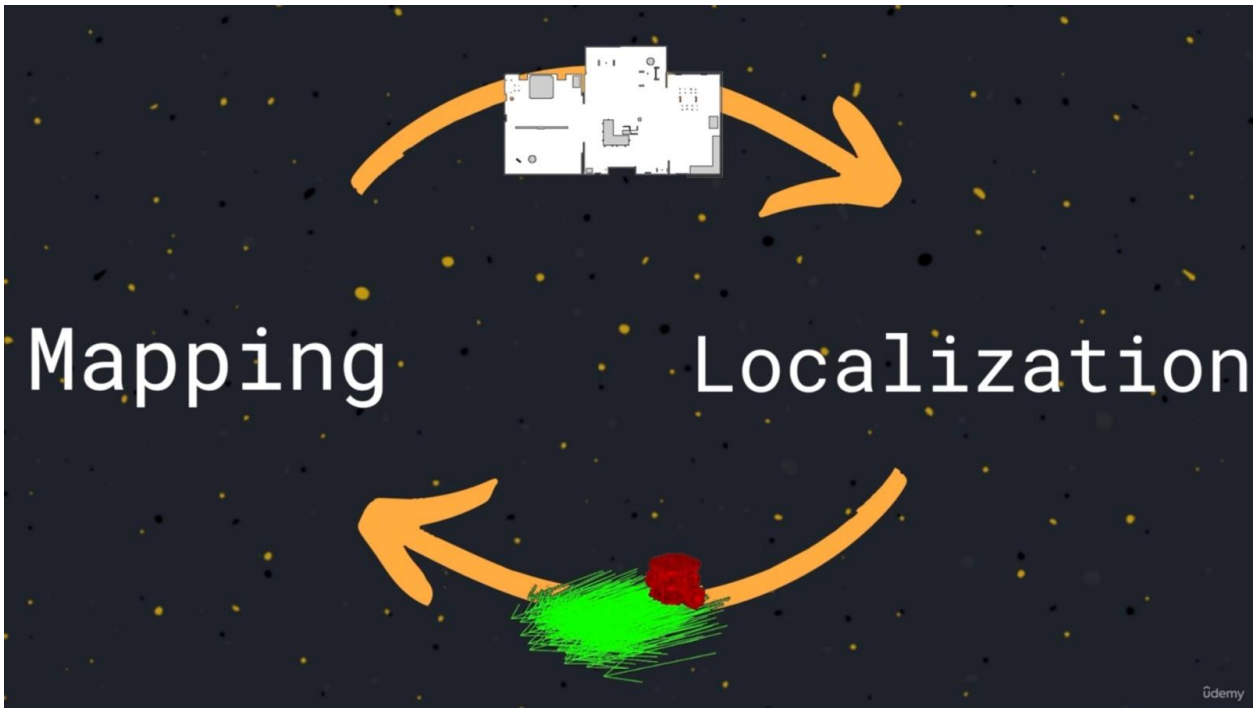


FIGURE 163

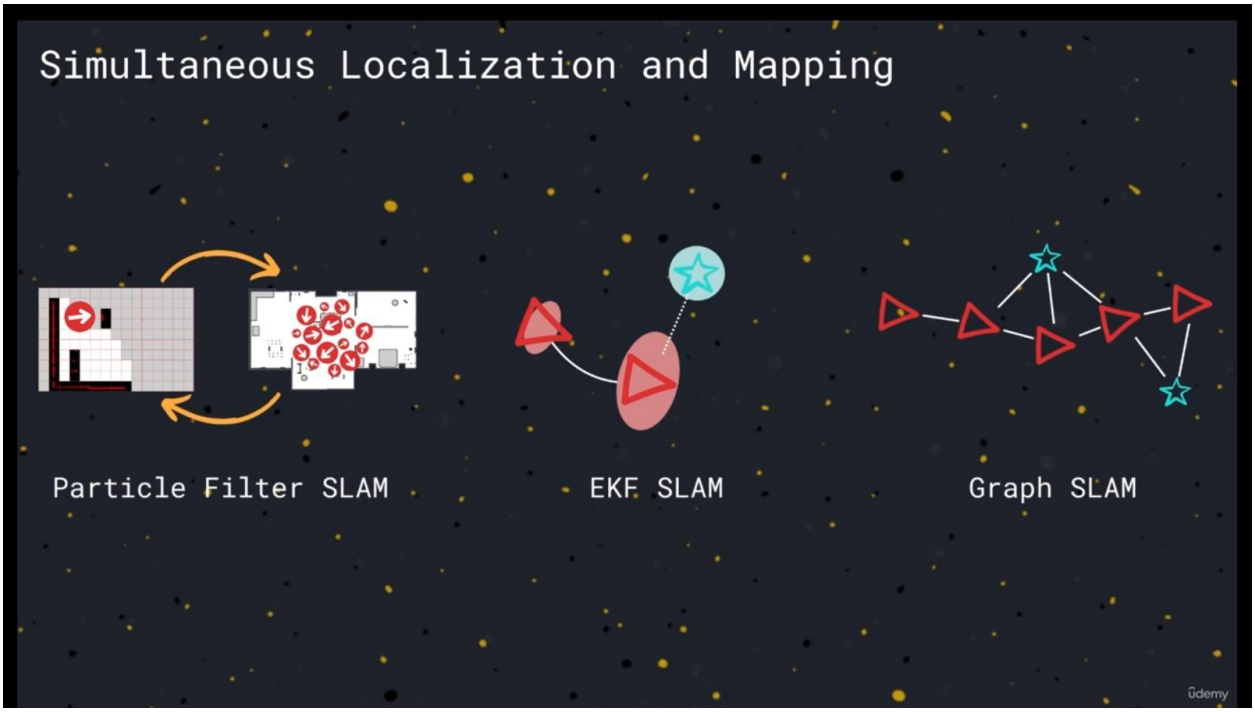


FIGURE 164

4.31.3.1 Particle Filter SLAM:

The Particle Filter (also known as Monte Carlo Localization) SLAM is a probabilistic method used for both localization and mapping. It is particularly useful in environments with uncertainty and when precise measurements are difficult to obtain.

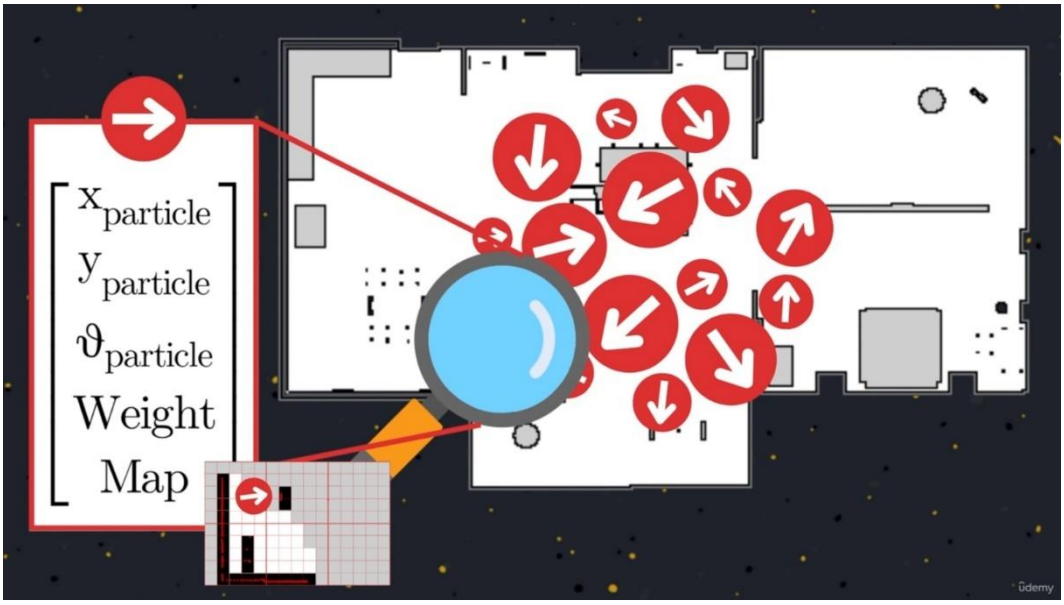


FIGURE 165

How it Works:

- A set of particles (each representing a possible hypothesis of the robot's pose and map) is maintained.
- The robot uses sensor data to update the likelihood of each particle based on its current observations.
- After each update, a process called resampling occurs, where particles with higher likelihoods are kept, and less likely particles are discarded.
- This process allows the robot to gradually refine its position and the map as it explores.

Advantages:

1. Non-linear systems: Particle filters can handle non-linear systems and non-Gaussian noise, making them very flexible.
2. Robust to multi-modal distributions: Particle filters work well in environments where multiple possible positions or maps exist (e.g., loop closures).

Applications: Often used in small robots, drones, and autonomous vehicles for localizing and mapping dynamic environments.

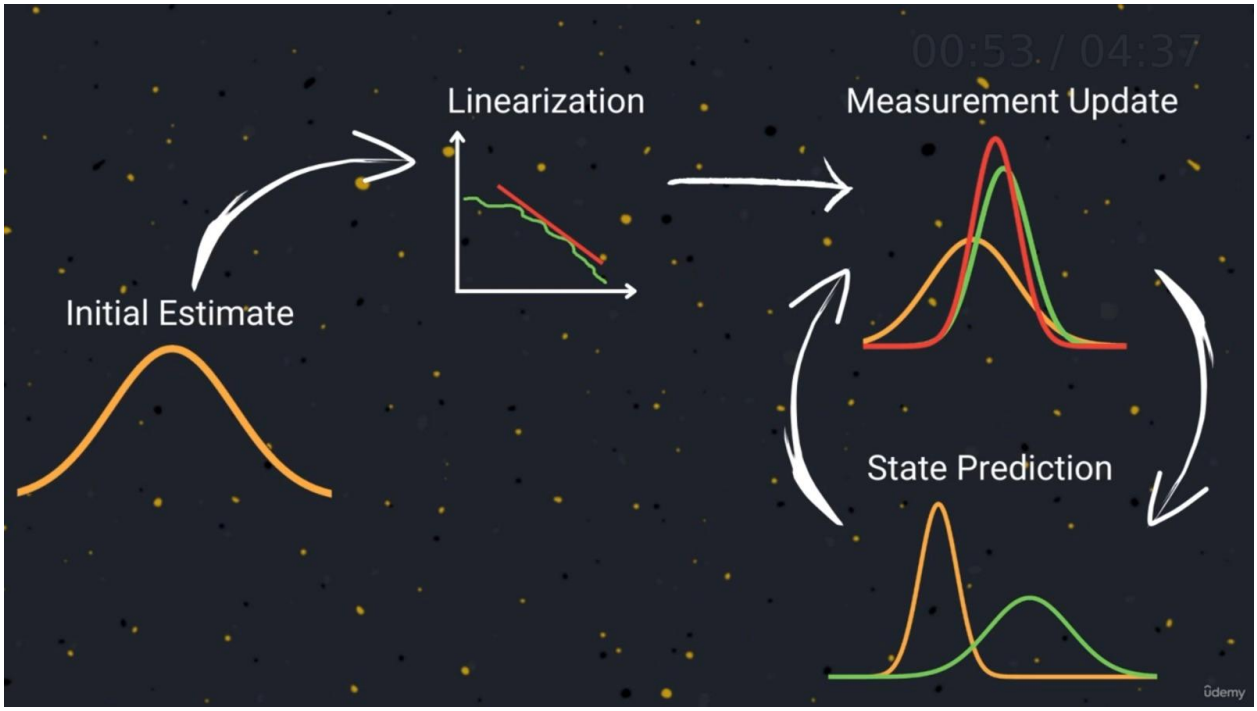


FIGURE 166

4.31.3.2 EKF SLAM (Extended Kalman Filter SLAM):

The Extended Kalman Filter SLAM is another popular probabilistic method that estimates both the robot's location and the map simultaneously.

How it Works:

- **State Representation:** EKF SLAM represents the state as a vector that includes both the robot's pose and the positions of landmarks in the environment.
- **Prediction Step:** The robot's movement is predicted based on motion models (like odometry).
- **Update Step:** Sensor measurements are incorporated to update the state estimate and the map using a Kalman filter, which is a recursive algorithm that updates predictions with new measurements.

Advantages:

- **Efficient:** EKF SLAM is relatively computationally efficient when the number of landmarks is small.
- **Well-established:** It is a well-understood technique with a solid mathematical foundation.

Challenges:

- Non-linearities: EKF SLAM assumes linear motion and sensor models. Non-linearities (such as large turns or sensor errors) can lead to errors.
- Computational Complexity: As the map size increases, EKF SLAM can become computationally expensive because of the need to maintain and update a large covariance matrix.

Applications: Often used in industrial robots and autonomous vehicles that operate in relatively controlled environments with small or moderate map sizes.

4.31.3.3 Graph SLAM:

Graph SLAM is another approach that builds a graph structure where each node represents either the robot's pose at a particular time or a landmark. Edges between nodes represent the spatial constraints based on sensor measurements and robot motion.

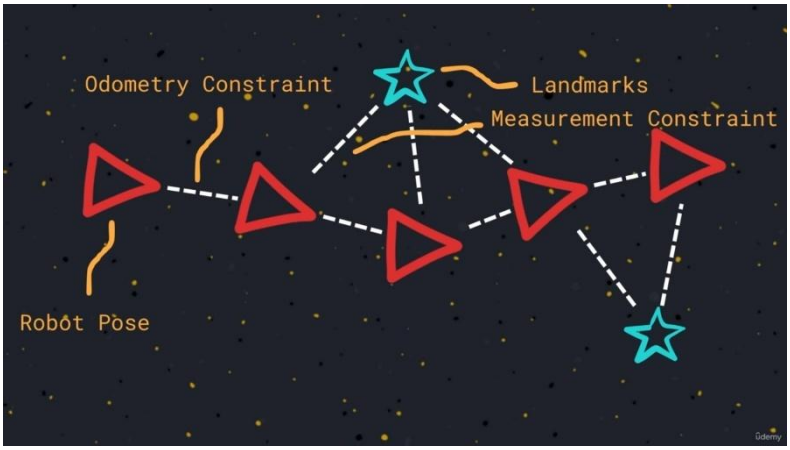


FIGURE 167 GRAPH SLAM

How it Works:

- The graph consists of nodes (representing poses and landmarks) and edges (representing relative constraints between poses or between poses and landmarks).
- As the robot moves, new nodes are added to the graph, and edges are formed based on sensor data.

Optimization: The robot's position and the map are optimized by adjusting the graph to minimize the error across all the edges. This is done using non-linear least squares optimization techniques, such as Gauss-Newton or Levenberg-Marquardt.

Advantages:

- Global Consistency: Graph SLAM handles the entire trajectory of the robot, providing more globally consistent maps and localization estimates.
- Flexible: Can be used with various sensors (LiDAR, cameras, etc.) and is more robust to large amounts of data.

Challenges:

- Computational Complexity: The optimization process can become computationally expensive, especially in large-scale environments.
- Loop Closure: Handling loop closures (when the robot revisits previously explored areas) is a challenge, as it can lead to large graph adjustments.

Applications: Used in autonomous vehicles, large-scale robot mapping, and in applications requiring highly accurate, globally consistent maps.

4.31.3.4 Graph SLAM - Front-End and Back-End:

Graph SLAM can be divided into two components: Front-End and Back-End.

Front-End:

- The front-end is responsible for data association, where the robot determines which landmarks correspond to previously observed features, and pose estimation, where it estimates the robot's position.
- The front-end works by processing raw sensor data and converting it into useful information, like robot poses and measurements of the environment.

Back-End:

- The back end is responsible for optimizing the graph formed by the front-end. It adjusts the nodes and edges in the graph to minimize the overall error in the system, usually using optimization algorithms.
- The back end ensures that the global consistency of the map is maintained and that loop closures are properly handled.

Applications: Separating front-end and back-end allows for parallel processing and better scalability, which is especially useful in large or complex environments.

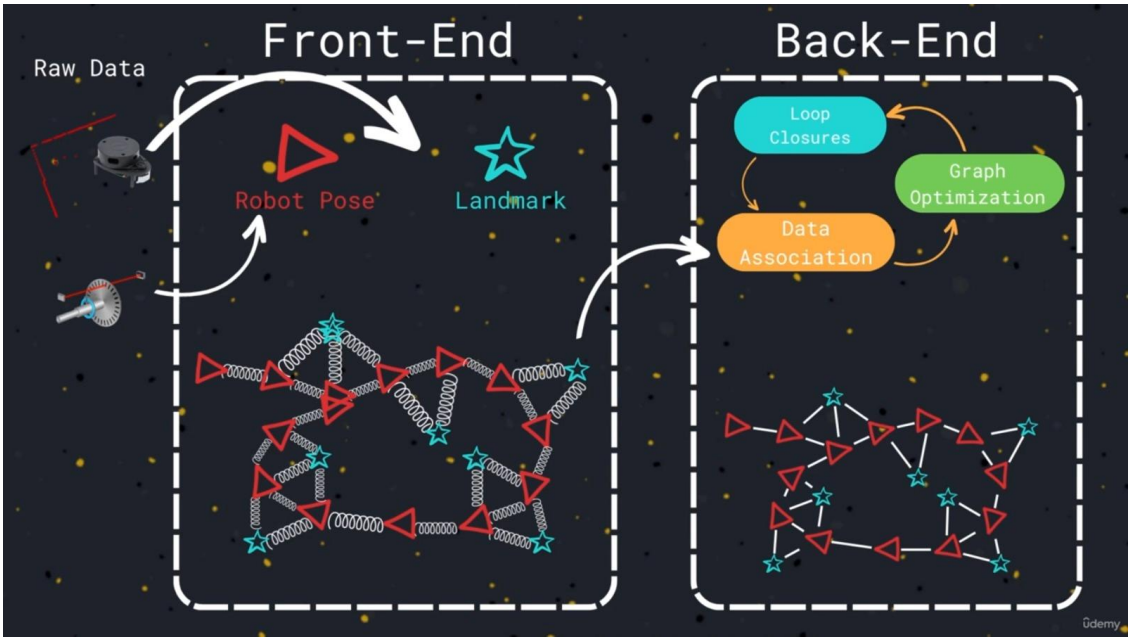


FIGURE 168

4.31.3.5 SLAM Toolbox:

Slam Toolbox is a ROS-based package used for performing SLAM in real-time. It implements several SLAM algorithms, including Graph SLAM, 2D mapping, and other related techniques, making it a flexible tool for autonomous robots.

Features:

- **Real-time SLAM:** Supports real-time mapping and localization, making it suitable for both small and large robots.
- **2D and 3D Support:** Can be used for 2D mapping (e.g., indoor environments) or 3D mapping (e.g., larger or outdoor environments).
- **Loop Closure:** Includes features to detect and handle loop closures, which is crucial for improving the accuracy and consistency of maps.
- **Flexible Optimization:** Provides options for different optimization techniques to ensure accurate maps.

Benefits:

- **Integration with ROS:** Works seamlessly with the ROS ecosystem, providing easy integration with other ROS packages.
- **Efficient:** It's designed to work efficiently, even in real-time scenarios with complex environments.

Multi-Robot Support: Can be used for multi-robot SLAM, which is useful for collaborative mapping.

Applications:

- Ideal for robotic systems using ROS 2 that require real-time SLAM capabilities, especially in dynamic or large environments.
- slam toolbox is a ROS-based tool for performing real-time SLAM, offering efficient and scalable solutions for autonomous mapping and localization.
- Each of these SLAM approaches has its strengths and is suitable for different robotic applications depending on the environment, computational resources, and real-time requirements.

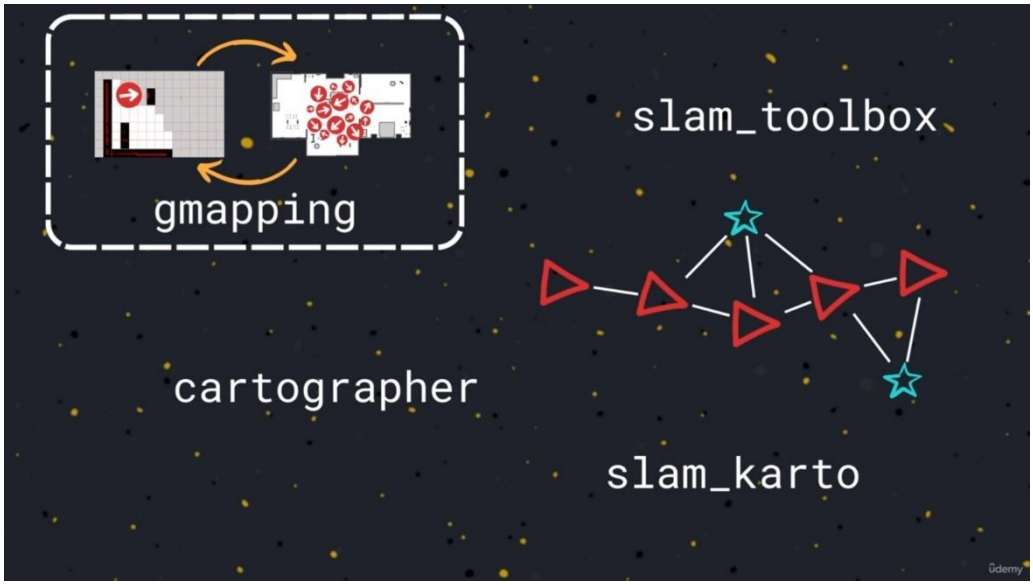


FIGURE 169

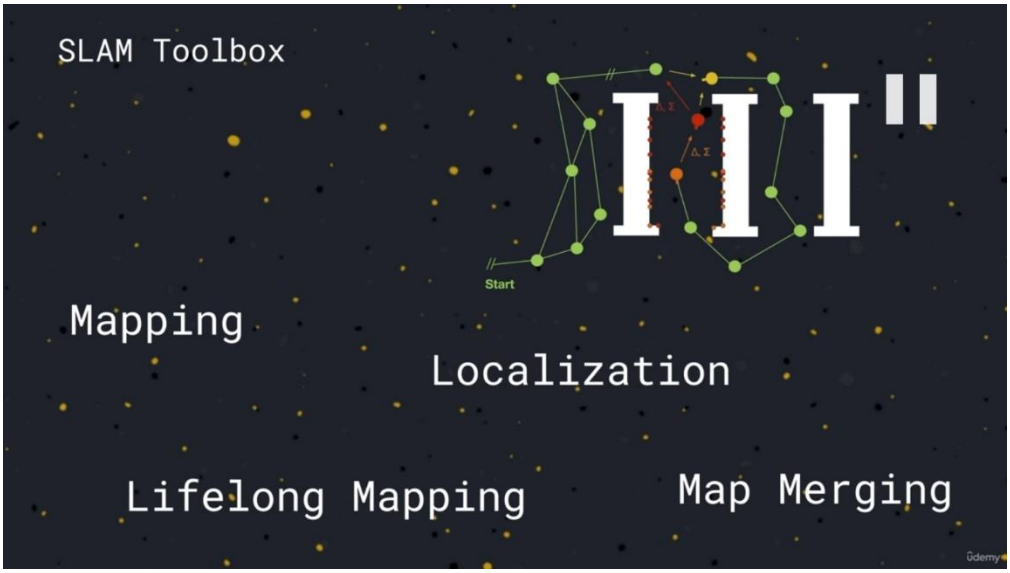


FIGURE 170

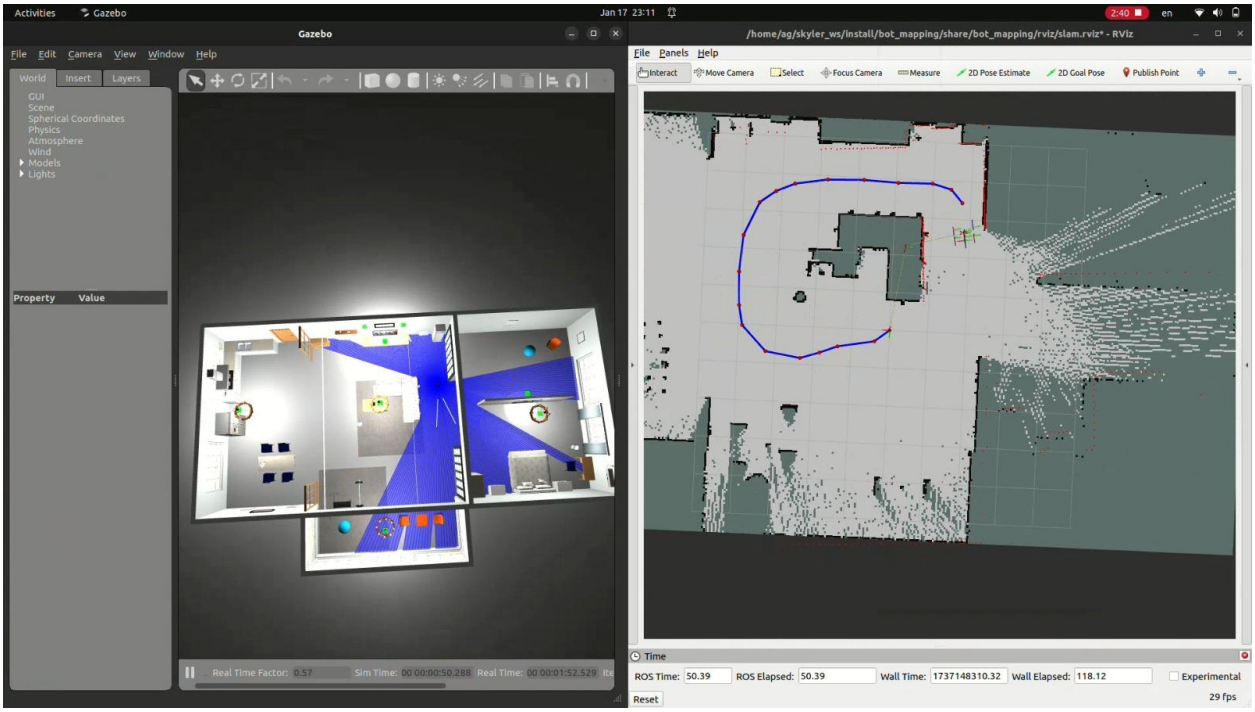


FIGURE 171 LAB SLAM

4.31.4 ROS2 Navigation

Steps for Navigation:

1. Start Your Robot:

~ ros2 launch am gazebo.launch.py world:=/home/AG/skyler_ws/src/bot_describtion/worlds/ world.world

2. Start the Main Navigation2 Launch File:

~ ros2 launch nav2_bringup bringup_launch.py map:=home/map.yaml

Add use_sim_time:=True if using Gazebo.

3. Start RViz:

~ ros2 run rviz2 rviz2

4. Send Navigation Commands:

- Use the “2D Pose Estimate” button to set the initial pose.
- Use the “Nav2 Goal” button to send navigation goals.

After giving the 2d pose estimate:

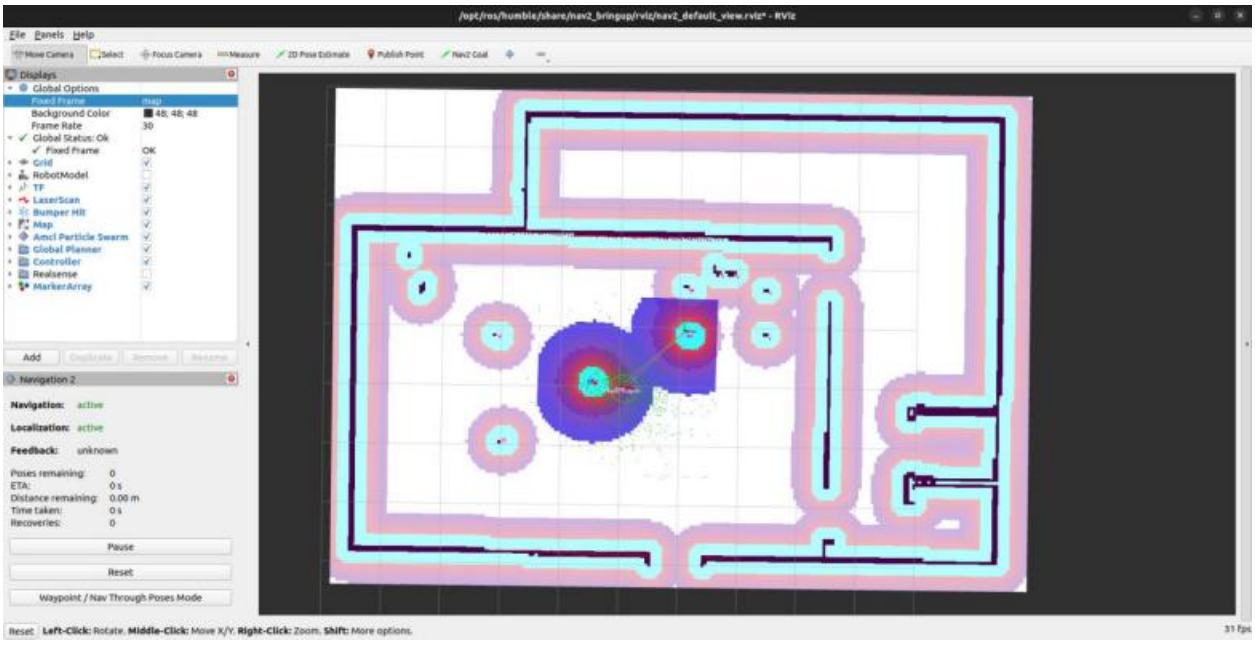


FIGURE 172

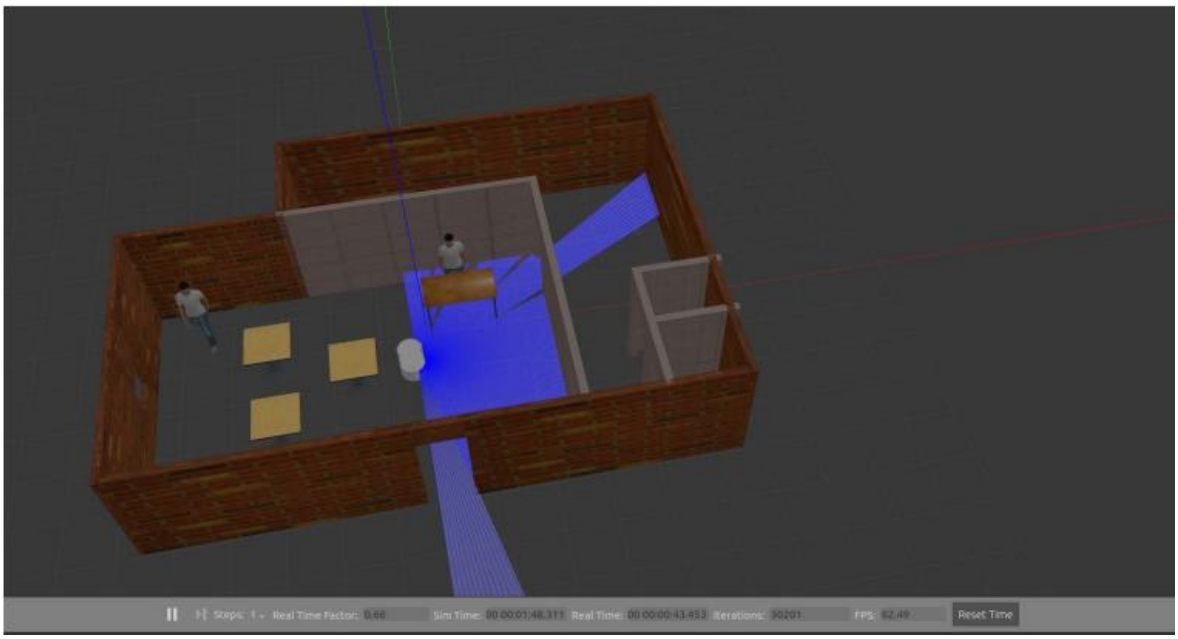


FIGURE 173

giving the robot nav goal with orientation:

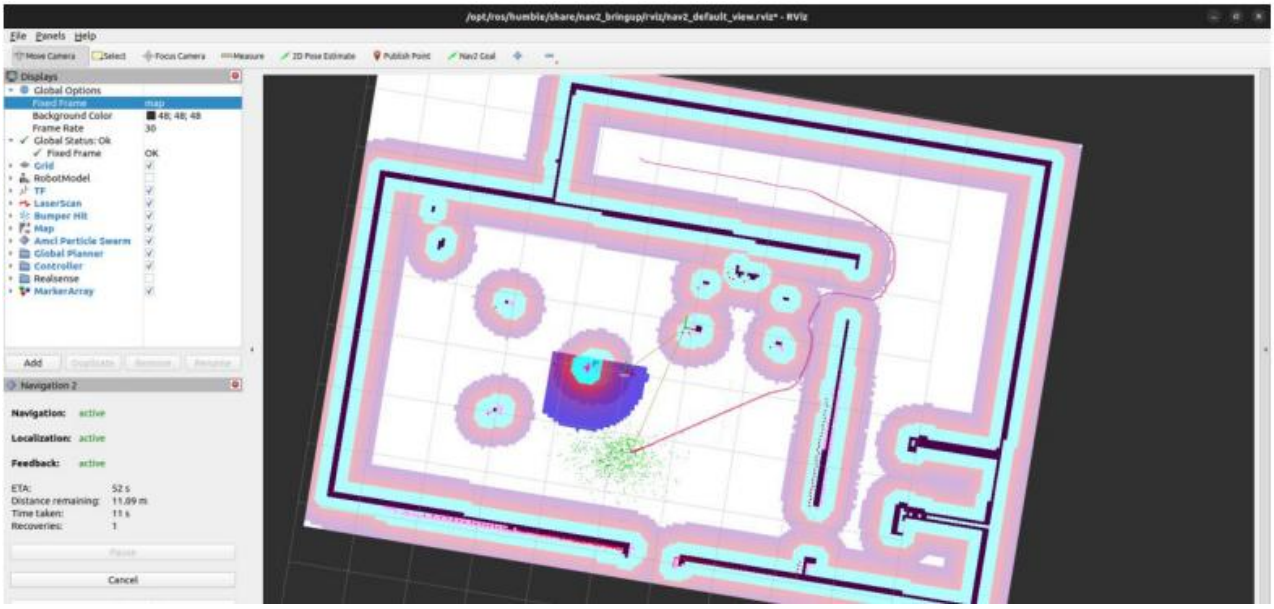
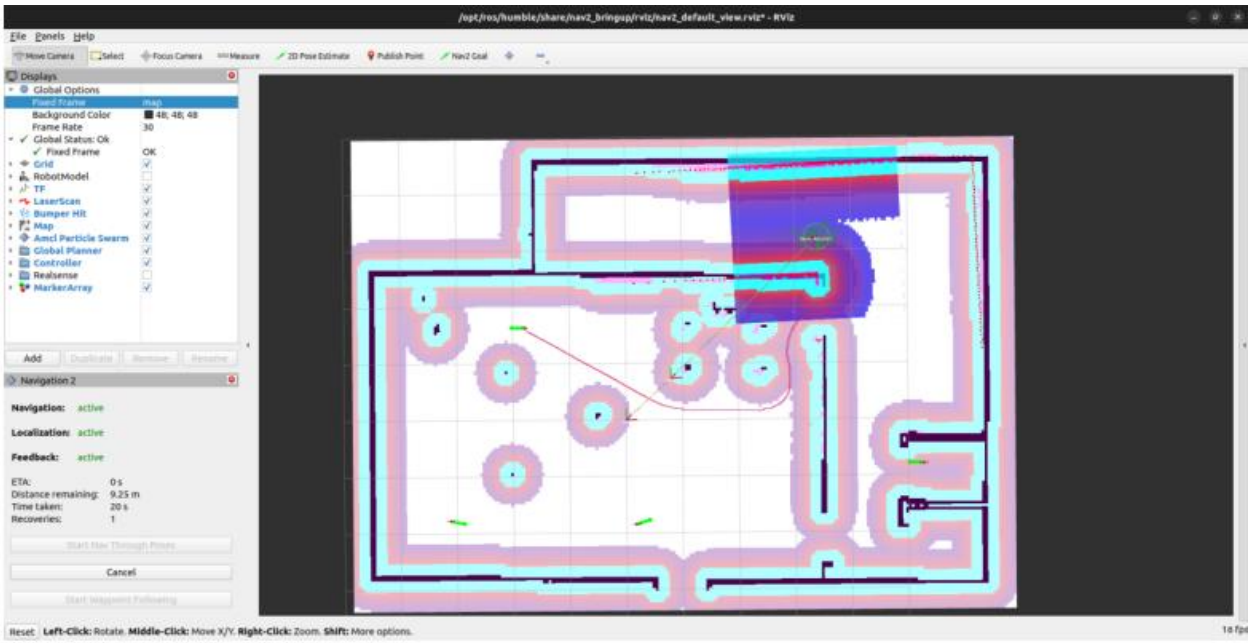


FIGURE 174

giving the robot multiple goals:



4.31.4.1 Understanding Global and Local Planners in Robotic Navigation

Introduction In this explanation, I focus on demonstrating the practical application of global and local planners in robotics, choosing to illustrate their functions through a live demo instead of lengthy theoretical descriptions. Overview of the Demonstration I start by initiating a data module and launching a navigation tool, highlighting the interplay between the global and local planners in robotic navigation. Global Planner Explained Path Calculation: The global planner creates an efficient route (shown as a pink line) using the entire map. It leverages a 'cost map' where each pixel's cost is determined by its closeness to obstacles. Update Frequency: The path generated by the global planner is periodically updated, with the frequency dependent on the system's settings.

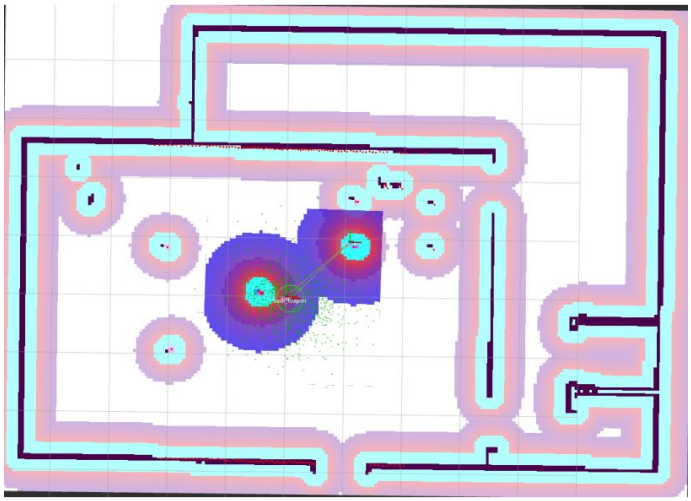


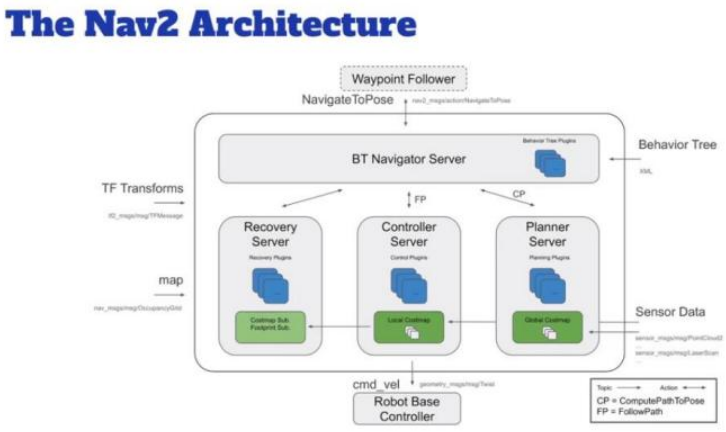
FIGURE 175

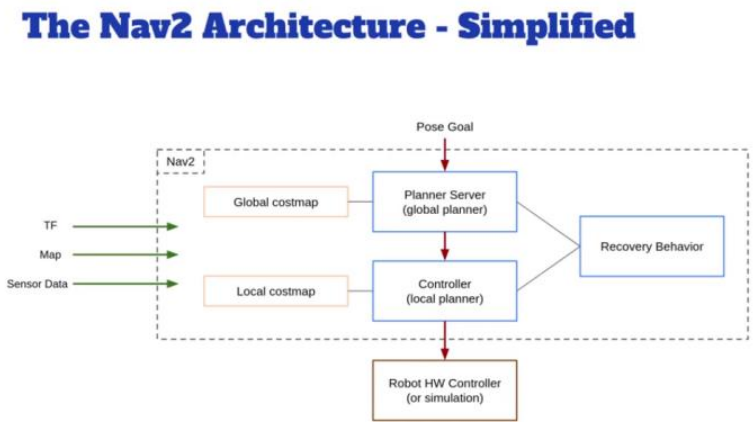
4.31.4.2 Local Planner (Controller) Details

- **Real-Time Guidance:** Known as the controller, the local planner is tasked with navigating in real time. It follows the path determined by the global planner, making adjustments as necessary.
- **Cost Map Application:** It uses its own cost map, focusing on the immediate vicinity of the robot, to guide its movements.
- **Higher Update Rate:** The local planner updates more frequently than the global planner, allowing for quick responses to the surrounding environment.
- **Analogy for Better Understanding:** I compare the global planner to a car's GPS system, which sets out the route and updates it occasionally. The local planner is akin to the driver, who follows the GPS but also makes real-time decisions, like taking detours or avoiding sudden obstacles.

4.31.4.3 Comprehensive Overview of the Navigation 2 Stack in Robotics

Introduction Having explored the primary components of the Navigation 2 (Nav2) stack, it's now time to integrate them into a cohesive architecture. This will provide a holistic understanding of the stack's operation. Although the official Nav2 architecture may seem complex, especially for beginners, I'll simplify it for better comprehension.





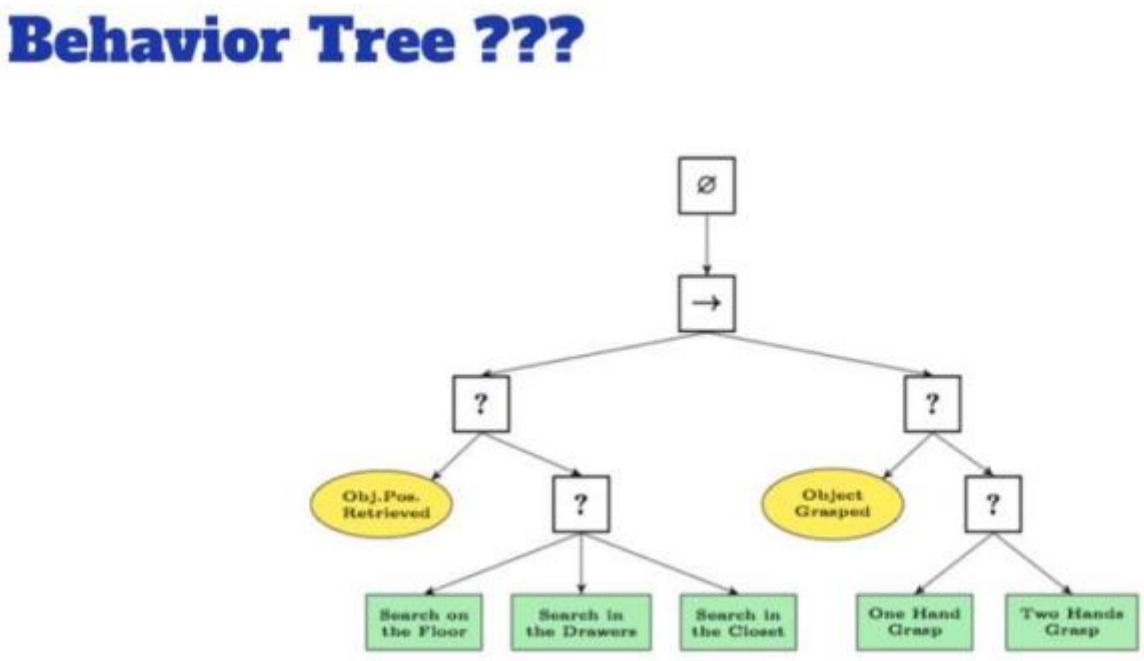
1. **Navigation 2 Box:** This is the core of the navigation stack.
2. **Required Inputs:**
   - Transforms (TFs): Essential for navigation, as seen previously.
   - Map Generation: Before navigation, a map is generated using SLAM (Simultaneous Localization and Mapping), which is then provided to the Nav2 stack.
   - Sensor Data: Inputs like laser scans from a LiDAR, camera streams, or 3D scanners are crucial for the stack's functionality.
3. **Navigation Goal Processing:**
   - Goal Pose: A navigation goal includes a position and orientation for the robot.
   - Global Planner (Planner Server): Responsible for finding a valid path using the map and the global cost map.
   - Local Planner (Controller): Ensures the robot follows the path to reach its destination, utilizing the local cost map and sensor data. It sends commands to the robot controller.
   - Robot Controller: A custom component that translates velocity commands into actionable commands for the robot's motors.
4. **Recovery Behavior:** Activated when the robot faces difficulties in path following, it includes actions like clearing the cost map or making spatial adjustments.

Official Nav2 Architecture

- **Like Simplified Version:** The official architecture includes most components from the simplified version.
- **Behavior Tree Navigator Server (BT):** This is an additional component not detailed in the simplified version. A behavior tree is a model for representing behavior sequencing. For instance, in a grasping task, the behavior tree defines the sequence of searching and grasping actions.

4.31.4.4 Behavior Tree in Navigation

Operational Flow: When a goal pose is received, the planner server first finds a path. The controller then helps the robot follow this path. In case of navigation issues, the recovery server attempts a recovery behavior.



The behavior tree helps the robot navigate to a goal, and if it faces problems, it tries to recover by clearing obstacles or adjusting its behavior.

Root Node: Starting point.

- Navigate With Replanning: A sequence of actions.
- Compute Path: Calculate the path to the goal.
- Follow Path: Make the robot follow the path.
- Goal Checker: Check if the goal is reached.

If there are issues, it goes into **recovery**:

- Navigate Recovery: Initiates recovery behaviors.
- Clear Costmaps: Clear obstacles from maps.
- Spin and Wait: Rotate or pause to resolve issues.

Chapter 5
Experimental Work

Chapter 5
Experimental Work

5.1 Climbing Robots Design and Manufacturing Considerations

Climbing robots require an effective design and manufacturing approach that ensures not only lightweight construction but also a high payload capacity and good maneuverability. In this work, the design of the robot differs slightly from the initial concept. The adhesion force is achieved using a suction motor of the "AC universal" motor type, rather than a BLDC motor, as the locally available BLDC motors do not provide sufficient adhesion force. This motor change also leads to the use of cables for power connections instead of a battery

5.1.1 Fabrication of the Structure and Components

The fabrication process began with a detailed 3D design using SolidWorks, ensuring precise alignment of dimensions and accurate assembly. The design focused on balancing durability and lightweight characteristics to meet the robot's operational requirements on vertical surfaces. The structure was divided into key components to simplify assembly and maintenance, while securely accommodating motors and sensors.

Materials were carefully selected to optimize performance. Acrylic was used for the outer frame due to its lightweight properties and excellent stress resistance, while metal was chosen for motor mounts and supporting components to provide additional strength and stability during operation. To achieve the highest level of precision, CNC tools were employed for cutting and assembling the components. This ensured parts were perfectly aligned with the theoretical design, facilitating error-free assembly.

During the assembly phase, high-quality screws and fasteners were used to ensure structural stability and to withstand the stresses caused by the robot's weight and suction forces. Additionally, preliminary tests were conducted after each assembly step to verify that the dimensions and fittings aligned with the planned design.

To ensure structural balance, weight distribution was strategically optimized. Heavy components, such as motors, were placed in the lower part of the robot to achieve a low center of gravity, enhancing stability during vertical operations. The robot’s base was designed to withstand the suction forces generated by the closed impeller system without deformation or bending.

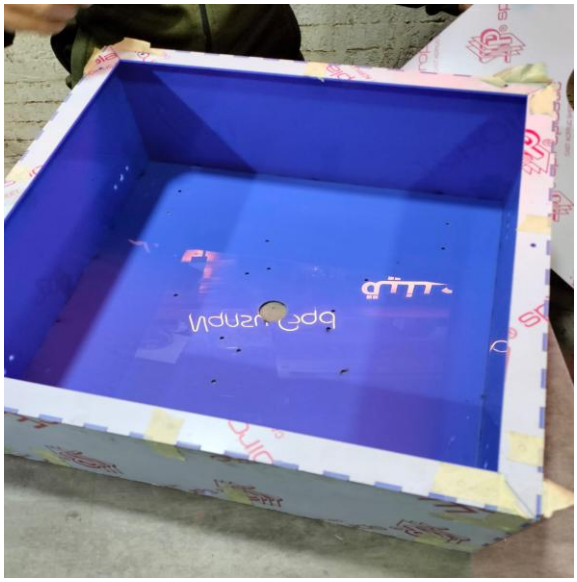


FIGURE 176 BODY

5.1.2 Base Plate Manufacture and Testing

The base plate of the robot is fabricated from acrylic, chosen for its light weight, cost-effectiveness, and ease of machining. Acrylic plates are lighter than aluminum plates for the same stress, and their thickness is smaller compared to aluminum. The base plate has a thickness of 3 mm. Experimental tests show that the robot's base plate can support over 600 N. These tests were conducted by evacuating air from a chamber and measuring the internal pressure, which reached 90 kPa (absolute)

The robot utilizes an AC Universal motor with a power rating of 1000 Watts, 30,000 RPM, and 4 amps for its operation. The module incorporates one sealing method, using an inflated tube seal controlled by an ON-OFF switch to operate the AC motor. A second sealing type, the skirt bristle seal, is used along with a lifting mechanism for the inflated tube. Some modifications to the manufacturing process are required to ensure optimal performance

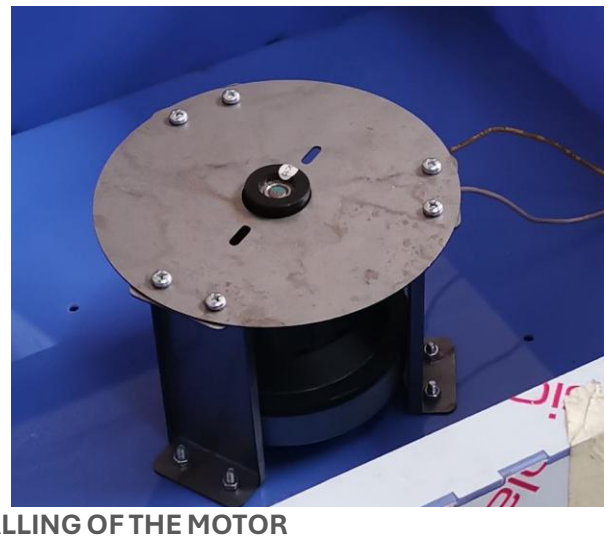


FIGURE 177 INSTALLING OF THE MOTOR

5.2 Installation of the Motion System

The installation of the motion system is a critical step in ensuring the robot's efficiency and precision while operating on vertical surfaces. After completing the fabrication of the main structure, the focus shifted to assembling the motion components to seamlessly integrate with the design and meet operational requirements.

5.2.1 Motor Mounting:

The brushless DC motors (BLDC) were securely mounted to the rear wheels using custom-designed metal motor holders. These holders were crafted from lightweight yet durable materials to provide robust support and withstand the stresses generated during the robot's movement. The motor holders ensured precise alignment of the motors, minimizing vibrations and slippage during operation.

The positioning of the motors was carefully planned to achieve even weight distribution, enhancing the robot’s stability on vertical surfaces. The motors were mounted at precise angles to align with the wheel axes, reducing energy loss and ensuring smooth power transmission.

FIGURE 178 MOTOR MOUNTING

5.2.2 Wheel and Timing Belt Installation:

The rear wheels were directly attached to the motor shafts, utilizing high-quality wheels with a 10 cm diameter to balance traction force and movement speed. To ensure synchronized and precise motion between the wheels, timing belts were incorporated as a key component of the motion system.

The timing belts were selected for their durability and flexibility, capable of withstanding the forces generated during operation without stretching or slipping. The belts were tensioned with precision using adjustable pulleys to eliminate slack and ensure consistent performance under varying loads.

5.2.3 Steering Mechanism for the Front Wheel:

The front wheel was connected to a steering mechanism controlled by a shaft and a timing belt. This setup allowed the robot to steer accurately, enabling smooth directional changes and easy maneuverability during operation. The front wheel was mounted in a way that ensured free movement with minimal friction, resulting in stable and efficient performance.

5.2.4 Initial Performance Testing:

Following the installation of the motion system, preliminary tests were conducted to evaluate its performance under different operating conditions. These tests involved running the robot on various vertical glass surfaces and simulating obstacles such as window frames. The results demonstrated high efficiency in transferring motion between the wheels, with the robot moving smoothly and stably even during directional changes or sudden stops.

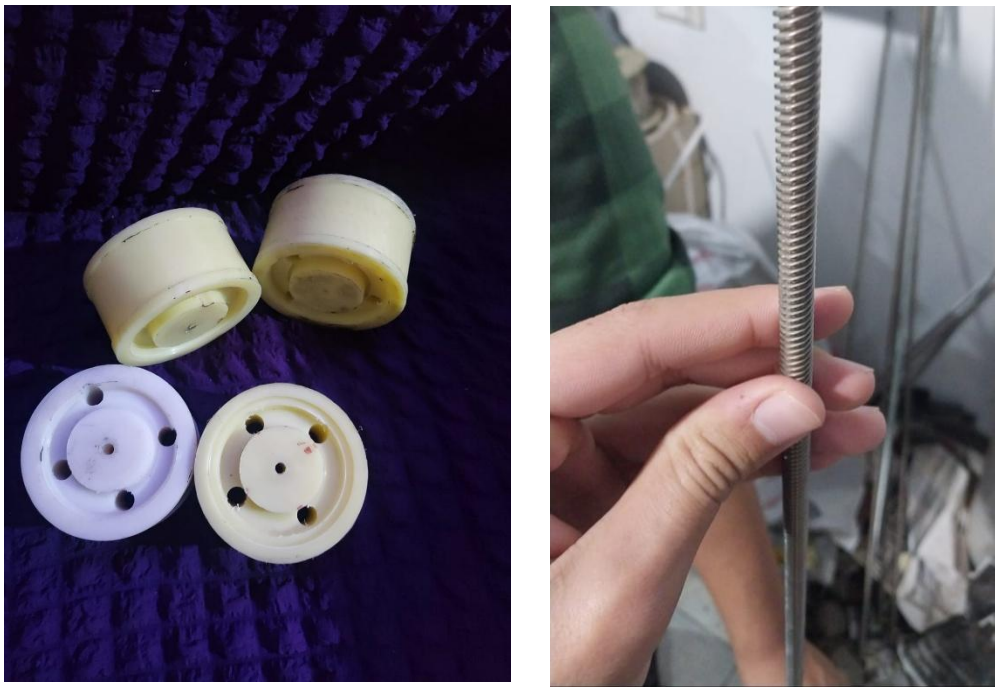


FIGURE 179 WHEEL AND AXE

5.2.5. System Optimization:

Based on the initial tests, adjustments were made to optimize the motion system. This included fine-tuning the tension of the timing belts, reinforcing motor mounts, and verifying alignment between the wheels and shafts to reduce vibrations and improve overall performance.

The installation of the motion system successfully achieved a balance between power and precision, enabling the robot to move steadily and smoothly on vertical surfaces. This system is a cornerstone of the robot's performance, allowing it to tackle practical challenges such as obstacles and uneven surfaces with ease.

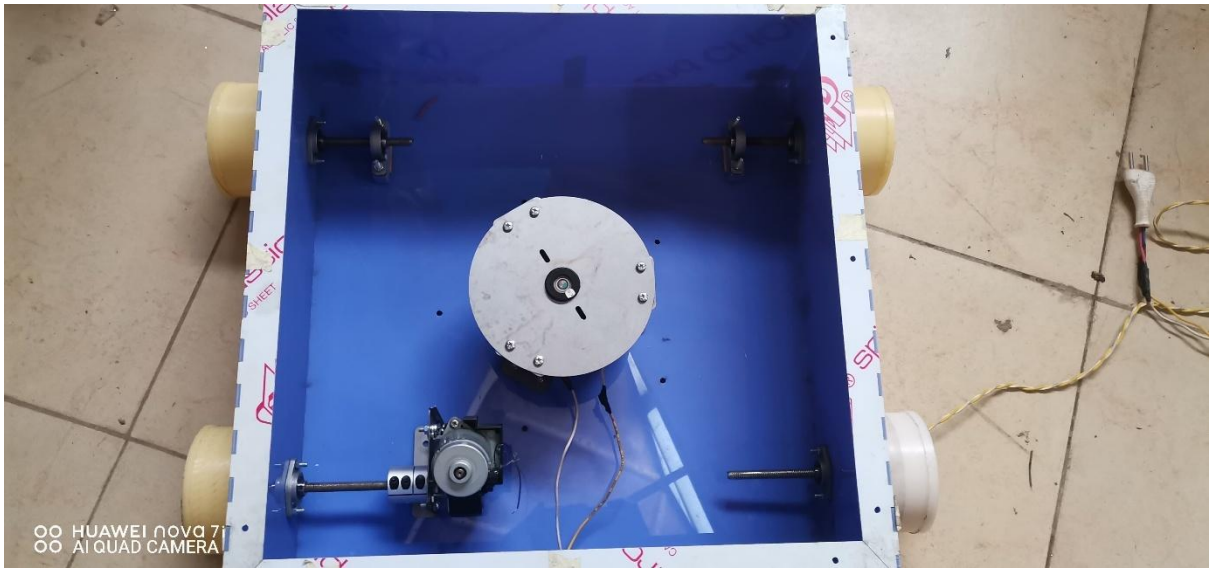


FIGURE 180

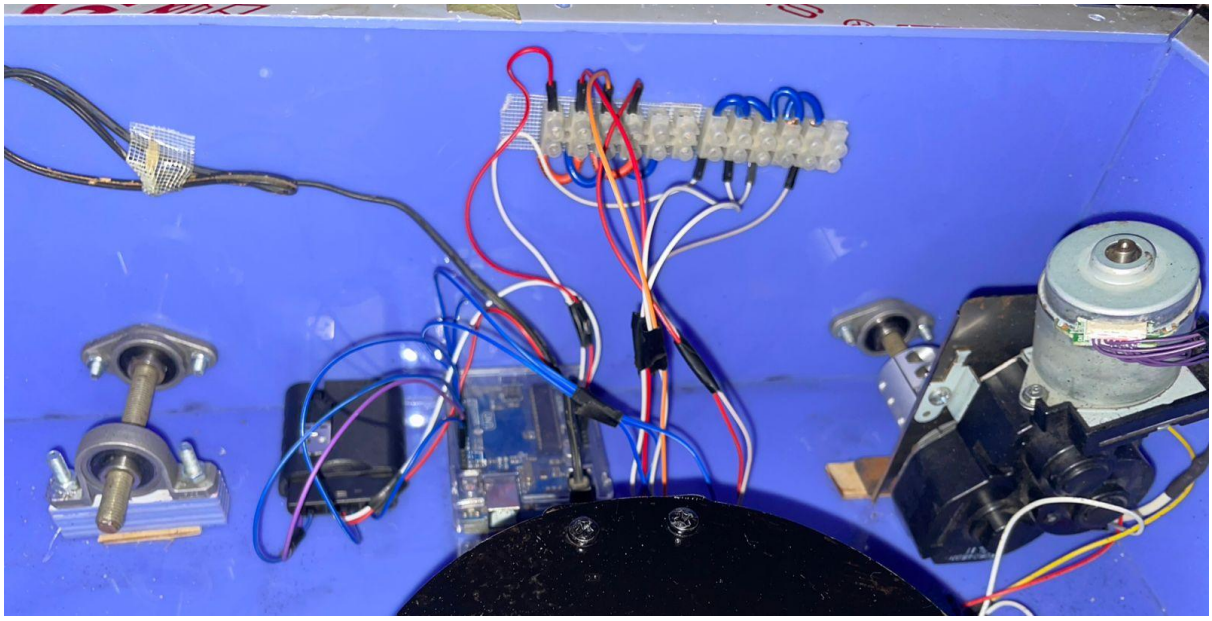


FIGURE 181

5.3 Installation of the Suction System and Sealing

The suction system is one of the most critical components in the window cleaning robot, as it ensures the robot's stability and secure attachment to vertical glass surfaces during operation. Its installation involved meticulous planning, precise testing, and the selection of high-quality materials to achieve reliable performance and minimize energy loss.

5.3.1 Testing and Installation of the Closed Impeller System

At the heart of the suction system is the closed impeller, which generates the suction force required to firmly adhere the robot to the glass surface. The impeller was carefully tested under various operational conditions to evaluate its performance in terms of airflow, suction power, and efficiency. These tests allowed us to fine-tune the impeller's rotational speed and optimize its integration with the robot's overall design. The impeller system was then installed in a dedicated compartment within the robot's body, ensuring it was securely mounted to withstand vibrations and operational stresses. Special attention was given to aligning the impeller with the suction chamber to maximize the pressure difference and create a consistent vacuum.

5.3.2 Sealing System Design and Material Selection:

To achieve an effective vacuum, the sealing system was designed to create an airtight environment between the robot and the glass surface. The sealing system was engineered using soft, flexible materials, such as high-grade silicone or rubber, which ensured proper contact with the glass, even on uneven surfaces.

The sealing material was selected for its durability and resistance to wear and tear, as it would be subjected to continuous pressure and friction during operation. The height and thickness of the seal were calculated (e.g., 1.2 cm height) to provide a balance between flexibility and stability, allowing the seal to adapt to surface irregularities while maintaining its structural integrity.

FIGURE 182

5.3.3 Assembly of the Suction Chamber and Sealing System:

The suction chamber was designed with precision to ensure efficient airflow and minimize turbulence, which could reduce suction power. The sealing material was attached around the perimeter of the suction chamber, creating a continuous airtight barrier.

The attachment process involved securing the sealing material with adhesives and mechanical fasteners to prevent detachment during operation. Additionally, the edges of the sealing material were finished to eliminate micro-gaps that could lead to air leakage.

5.3.4 Minimizing Air Leakage:

A significant focus during the installation was on minimizing air leakage, as even small gaps can greatly affect suction performance. To address this, the sealing system was tested for uniformity and continuity along its perimeter. Any imperfections detected were corrected through additional sealing or reinforcement.

To further enhance the system's efficiency, the sealing perimeter was optimized to align with the pressure distribution created by the impeller. This ensured that the suction force was evenly distributed, maximizing the robot's adhesion to the glass surface.

FIGURE 183 SEALING

5.3.5 Performance Testing and Optimization:

After completing the installation, the entire suction system, including the impeller and sealing components, was subjected to rigorous testing. These tests simulated real-world conditions, such as varying glass surface textures, angles, and environmental factors like wind. The system successfully demonstrated its ability to maintain a stable vacuum, securely holding the robot in place while it performed its cleaning tasks.

Based on the results, further refinements were made to improve efficiency, such as adjusting the sealing height, optimizing the impeller speed, and reinforcing areas prone to air leakage.

5.3.6 Final Integration and Results:

The fully installed suction and sealing system proved to be highly effective, providing the necessary adhesion force to support the robot's weight (8 kg) and additional forces during movement and cleaning. The combination of the high-performance impeller and durable sealing material ensured consistent suction power, even during prolonged operation.

This well-engineered suction system, combined with the robust sealing design, forms a cornerstone of the robot's capability to operate reliably on vertical surfaces, ensuring stability and efficiency throughout its cleaning tasks.

FIGURE 184

5.4 Integration with Electronic Systems

The integration of electronic systems was a crucial step in transforming the robot into an intelligent system capable of interacting with its environment accurately and efficiently. The robot was equipped with advanced sensors and control systems, including a dedicated circuit for controlling the suction motor's speed, which significantly enhanced its overall performance.

5.4.1 Installation of Navigation Sensors:

A high-precision LIDAR sensor was mounted on the robot to create detailed 3D maps of its surroundings. This sensor provides accurate data on distances and obstacles, enabling the robot to navigate intelligently on vertical surfaces and avoid collisions during operation.

5.4.2 Installation of Proximity Sensors:

In addition to the LIDAR, proximity sensors using ultrasonic or infrared technologies were installed. These sensors detect nearby obstacles with high accuracy, allowing the robot to make immediate decisions when encountering barriers.

FIGURE 185 LIDAR

5.4.3. Suction Motor Speed Control Circuit (SCR Dimmer):

Given the critical role of the suction system in the robot, we developed a custom control circuit based on an SCR Dimmer to regulate the speed of the suction motor (AC). This circuit allowed precise control over the closed impeller's rotation speed, optimizing suction performance for different operational needs.

- **Design Efforts:**
    - We initially designed the control circuit using Proteus software to simulate and validate the theoretical performance of the system.
    - After successful simulation, we manufactured a custom PCB for the circuit, incorporating design refinements to ensure stability and efficiency during practical application.
    - Several iterations of the design were tested and optimized, focusing on ensuring reliability under varying operational conditions.
- **Functions of the SCR Dimmer Circuit:**
    - Regulates the voltage supplied to the motor to control the impeller speed.
    - Improves energy efficiency by adjusting the speed based on the suction requirements.
    - Protects the motor from sudden load changes or voltage fluctuations.

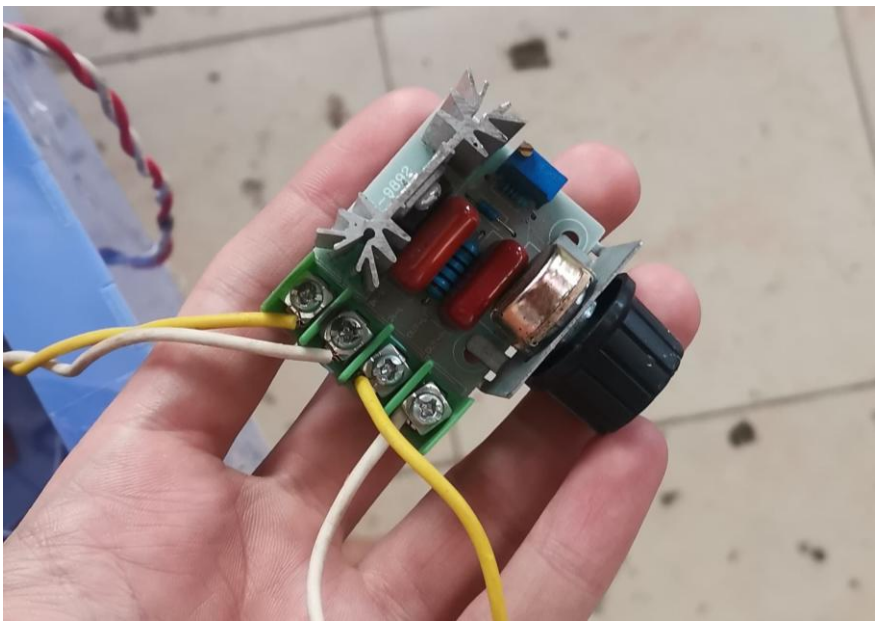


FIGURE 186 SCR

Design:

Main Components of the Digital AC Dimmer

TRIAC

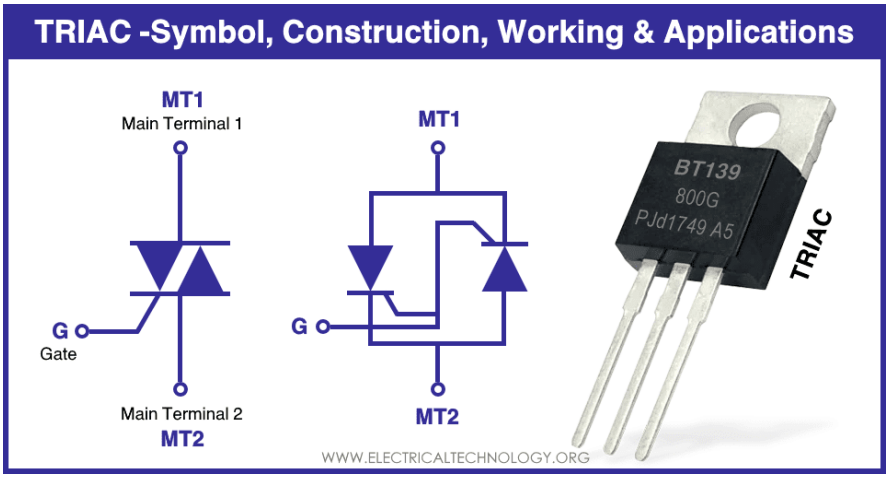


FIGURE 187 TRIAC

The TRIAC is the core switching device in the dimmer circuit, allowing bidirectional control of AC power. By adjusting the firing angle of the TRIAC, we control the amount of power delivered to the load, enabling precise dimming or speed control of AC devices.

MOC3021 Optocoupler (Triac Driver)

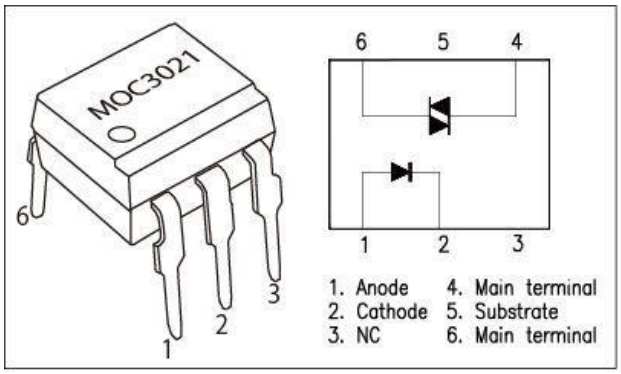


FIGURE 188 MOC3021 OPTOCOUPLER

It ensures reliable triggering of the TRIAC by providing consistent voltage breakdown characteristics. This helps stabilize the TRIAC's operation, particularly at lower power levels, and ensures smooth control without flickering or noise also the optocoupler is included to electrically isolate the high-voltage AC circuit from the low-voltage control circuit. This separation is essential for protecting sensitive components like the microcontroller and ensuring operator safety during use.

Zero-Crossing Detector

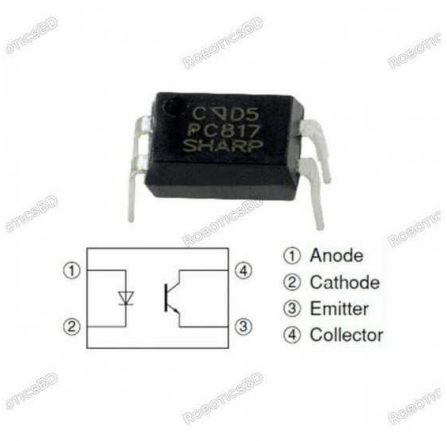


FIGURE 189 ZERO-CROSSING DETECTOR

The zero-crossing detector detects the point at which the AC voltage crosses zero. This information allows precise timing of the TRIAC firing, reducing electromagnetic interference (EMI) and ensuring efficient and smooth dimming.

Snubber Circuit

A snubber circuit, typically composed of a resistor and capacitor in series, is connected across the TRIAC. This circuit suppresses voltage spikes caused by inductive loads, protecting the TRIAC from damage and improving the overall stability and reliability of the dimmer. The snubber circuit ensures that the dimmer operates smoothly even under varying load conditions.

Microcontroller

The microcontroller serves as the intelligent control unit of the dimmer. It adjusts the firing angle of the TRIAC based on programmed inputs or user commands. This enables dynamic and precise control over dimming levels, providing versatility and ease of use.

Resistors

Resistors are used to limit and control the current throughout the circuit. They play a crucial role in protecting sensitive components. 5 resistors were used.

1. **Resistor for Bridge & Zero-Crossing**

   Voltage Drop across Bridge 2v | Voltage Drop across PC817's LED side 1.5v

   Peak Voltage = 311 - 2 – 1.5 = 307v | Peak Current 0.4A

   Bridge Resistor = 307 / 0.04 = 7700Ω = 10kΩ

2. **External Pullup Resistor 10kΩ**

3. **R for Driver and Arduino limiting current**

   Arduino 5v | moc3021 Led size rated current = 60mA, Voltage drop = 1.5v

   Resistor = (5-1.5)/0.05 = 70 Ω = 100 Ω

4. **R for Triac**

   Needs to handle power above 5W

Simulation:

Before trying the circuit, we had to validate the design first, so we used a simulation software called proteus 8 pro, placing all the components and uploading the Arduino code and the result were great, here is the circuit design:

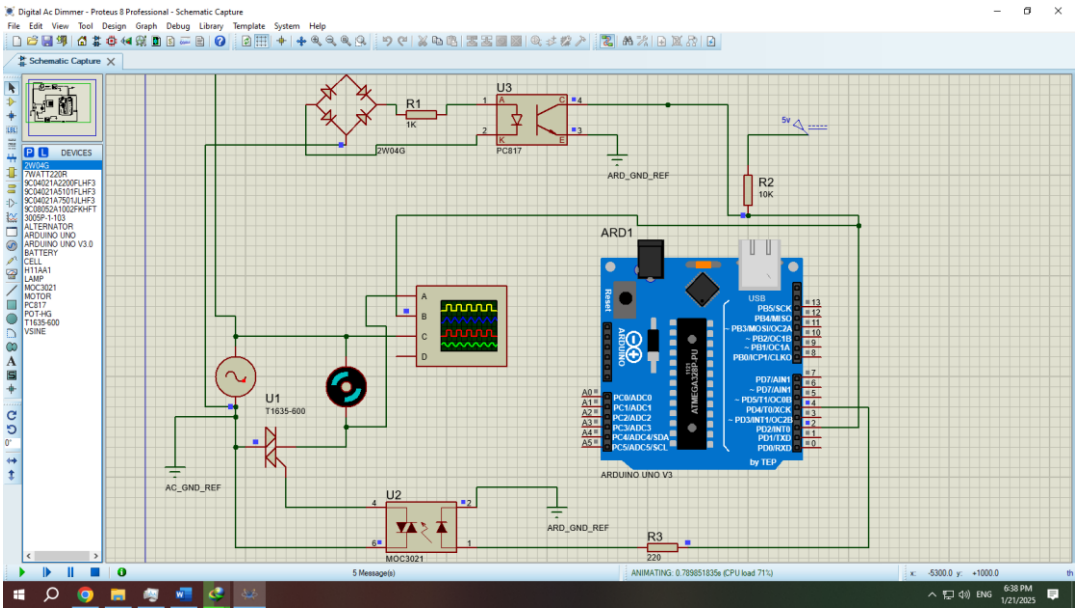


FIGURE 190 DESIGN OF CIRCUIT

And here is the circuit's output and input compared using the digital oscilloscope:

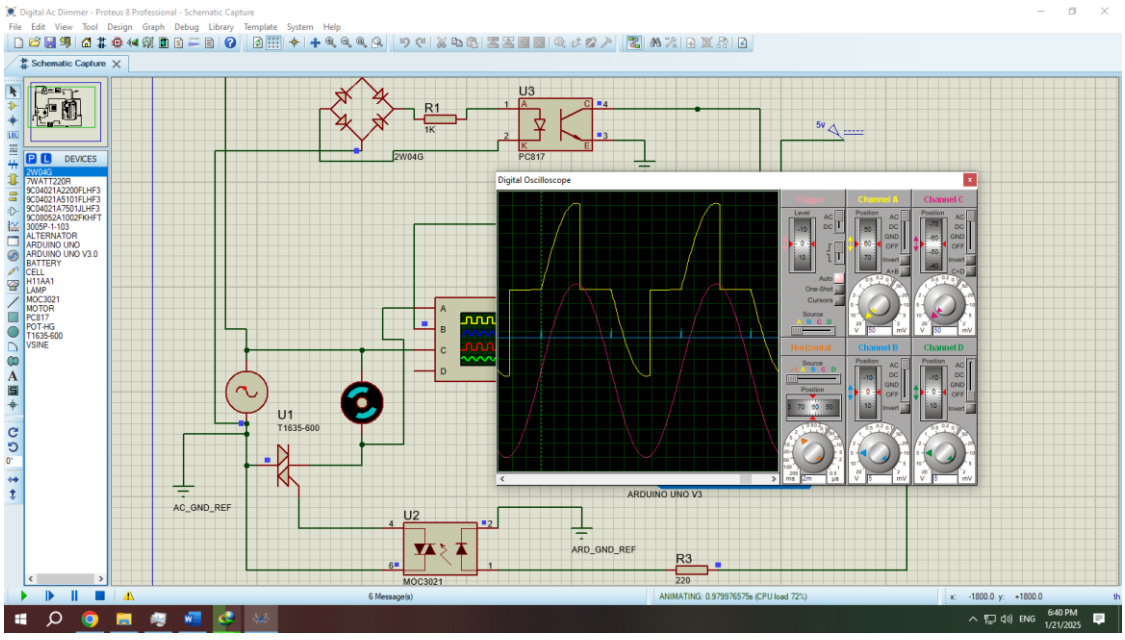


FIGURE 191 CIRCUIT'S OUTPUT AND INPUT COMPARED USING THE DIGITAL OSCILLOSCOPE

Implementation:

After designing the circuit, we had to make sure it was working by testing it on a low power device first (e.g. Lamp),

The circuit was put on a breadboard connecting all the components and after a few tests, it worked beautifully, and we could control the lamp's light intensity digitally with the code

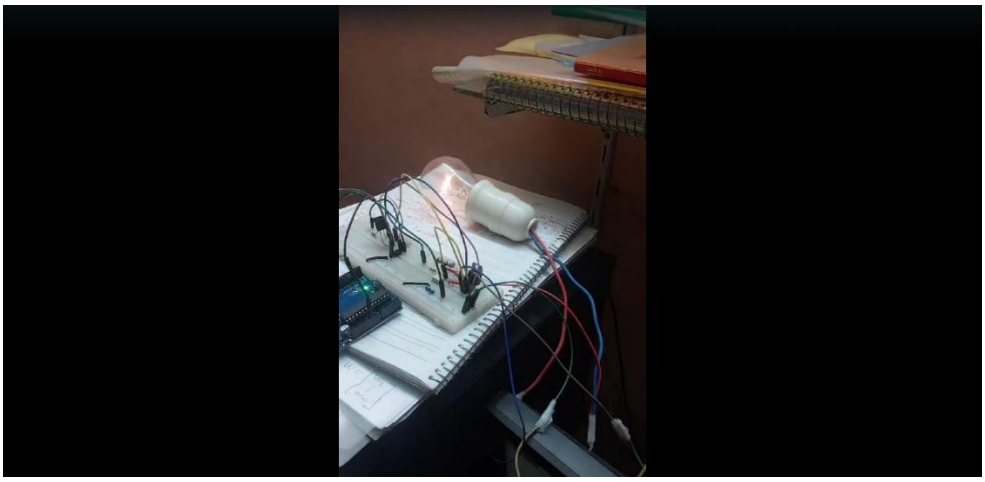


FIGURE 192

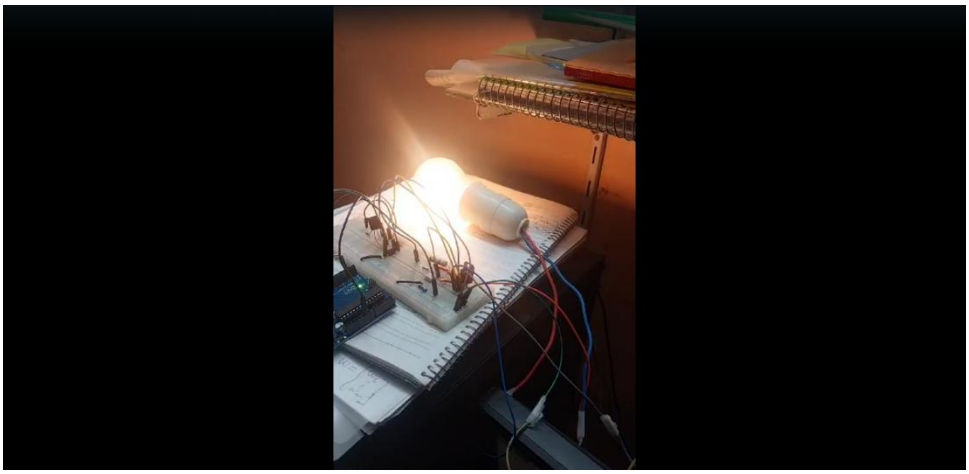


FIGURE 193

But when we tried to connect the motor (220v AC / 5A) it was too much for the breadboard to handle as it can't handle more than 5A, and the motor has a high startup current that causes the breadboard fuses to snap.

So, we moved to design the circuit as a PCB

PCB Design:

When designing the PCB for our circuit, we had to make sure that the current flowing through had big enough routes to go through without raising the board's temperature which was done through the following tables:

Temp Rise	10℃			20℃			30℃		
Copper	1/2oz.	1oz.	2oz.	1/2oz.	1oz.	2oz.	1/2oz.	1oz.	2oz.
Trace Width (:inch)	Maximum Current Amps								
0.01	0.5	1	1.4	0.6	1.2	1.6	0.7	1.5	2.2
0.015	0.7	1.2	1.6	0.8	1.3	2.4	1	1.6	3
0.02	0.7	1.3	2.1	1	1.7	3	1.2	2.4	3.6
0.025	0.9	1.7	2.5	1.2	2.2	3.3	1.5	2.8	4
0.03	1.1	1.9	3	1.4	2.5	4	1.7	3.2	5
0.05	1.5	2.6	4	2	3.6	6	2.6	4.4	7.3
0.075	2	3.5	5.7	2.8	4.5	7.8	3.5	6	10
0.1	2.6	4.2	6.9	3.5	6	9.9	4.3	7.5	12.5
0.2	4.2	7	11.5	6	10	11	7.5	13	20.5
0.25	5	8.3	12.3	7.2	12.3	20	9	15	24

FIGURE 194 TABLE 1 CARRYING CAPACITY PER MIL STD 275

As such for each route the appropriate trace width was chosen, design was cut into 2 layers,

Here is the top layer:

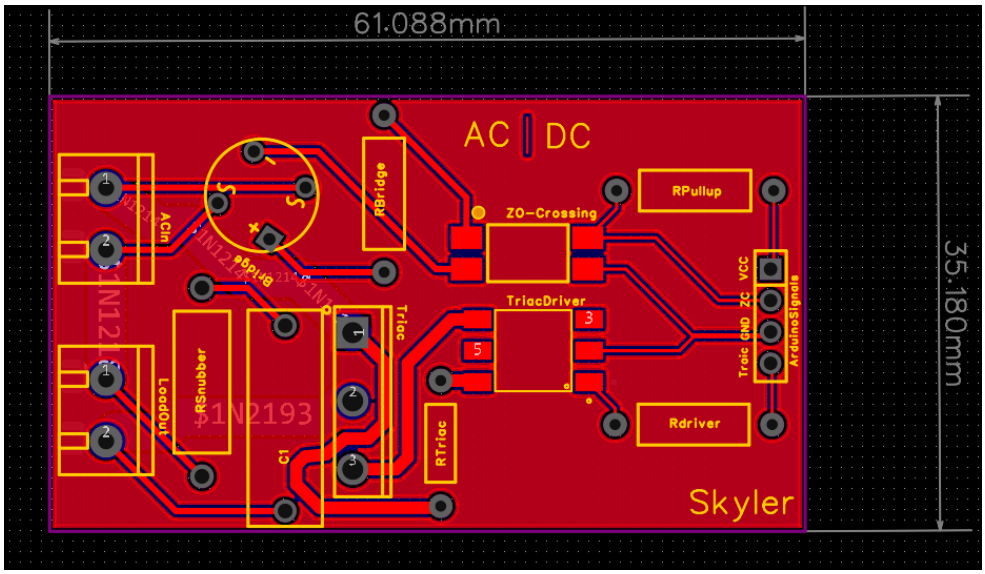


FIGURE 195 TOP LAYER

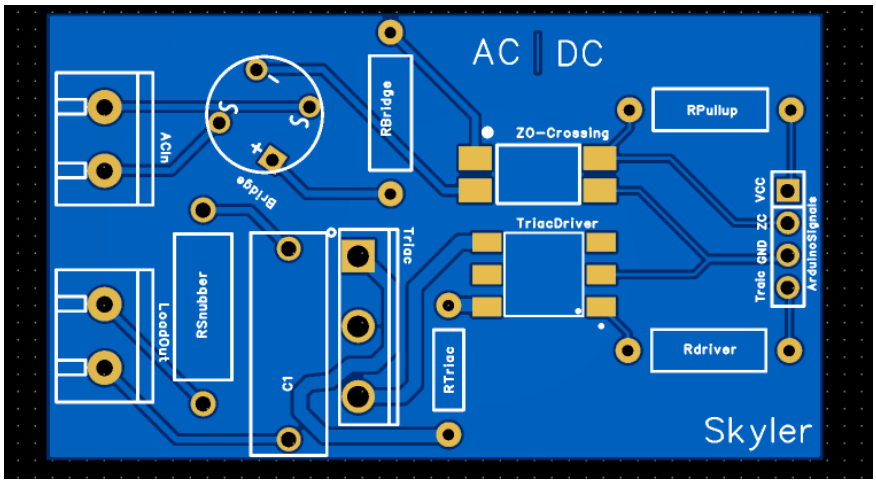


FIGURE 196

And here is the bottom layer:

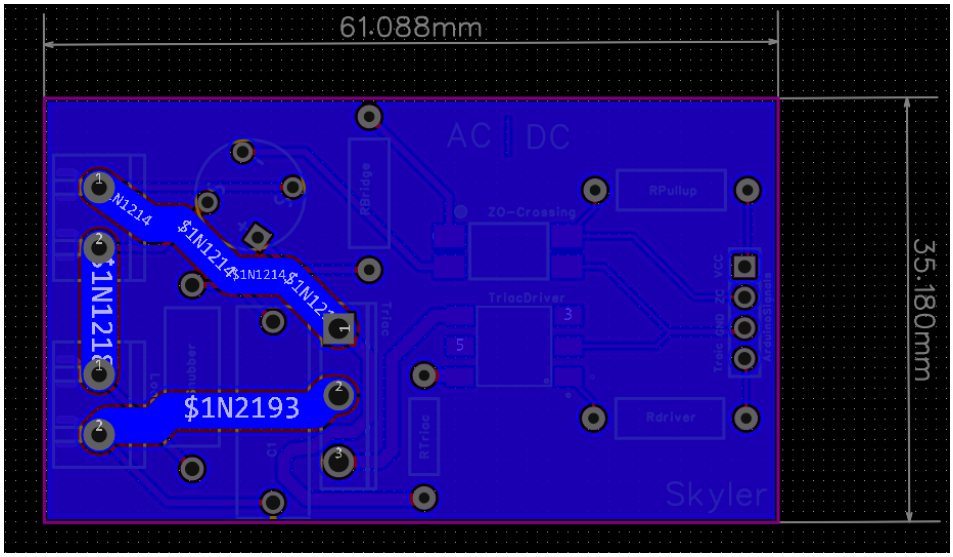


FIGURE 197

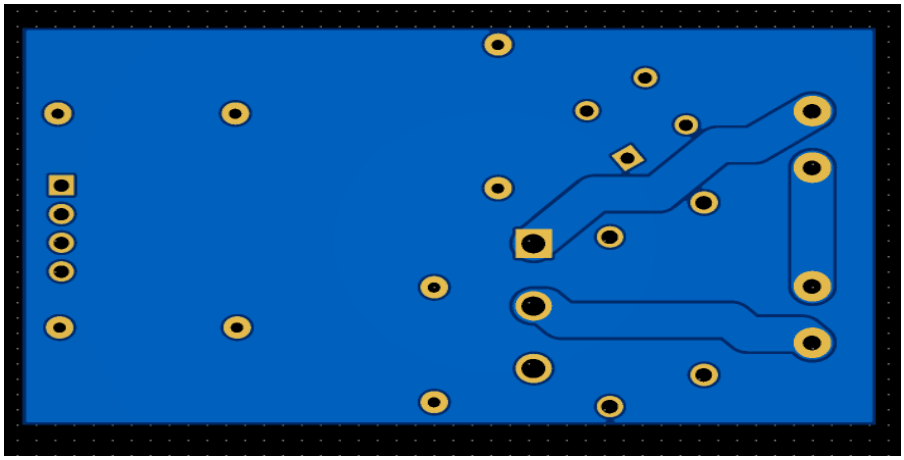


FIGURE 198

3D VIEW:

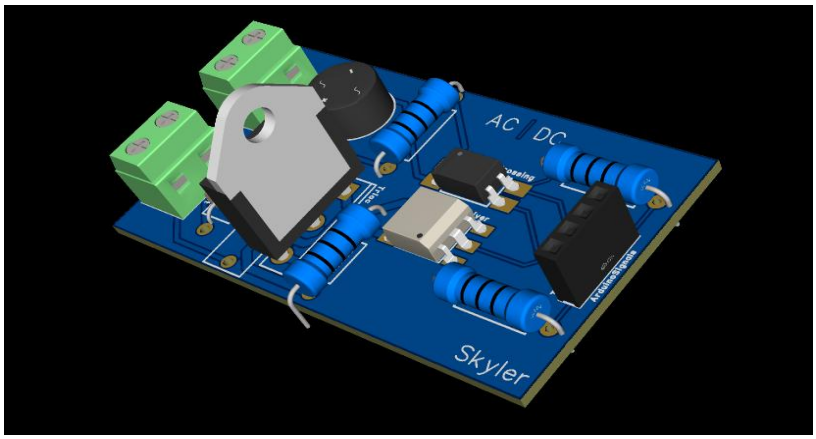


FIGURE 199

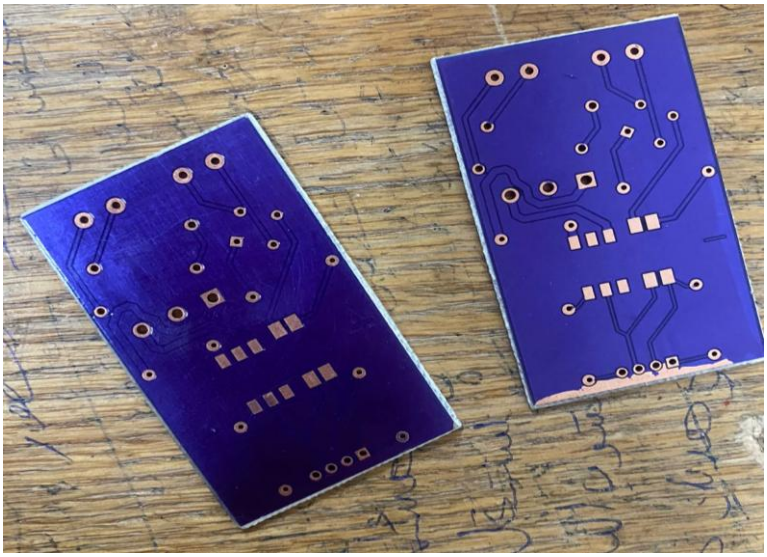


FIGURE 200

5.4.4 Integration with the Central Control System:

The SCR Dimmer circuit was integrated with the robot's central controller (such as an Arduino or Raspberry Pi), enabling the robot to automatically adjust suction speed based on sensor input.
Additionally, all sensors and motors were connected to a central processing unit that handled data input and issued appropriate commands for navigation and movement.

This dimmer was used to lower the supplied voltage to our motor, in turn lowering the speed it was a handy circuit with low price, but the only problem was that it was analog and there was no way to connect it and interface it digitally with the Arduino without risking damaging the Arduino components.

5.4.5 Performance Testing:

After completing the electronic integration, comprehensive tests were conducted to evaluate the effectiveness of the integrated systems, including:

- The robot's response to its environment using LIDAR and proximity sensors.
- The efficiency of the suction motor speed control using the SCR Dimmer circuit and its impact on energy consumption and overall performance.
- The performance of the electronic system in diverse environments and operational conditions.

FIGURE 201

5.4.6 Results:

The integrated electronic system successfully provided precise and complete control over the robot's navigation and suction speed. The SCR Dimmer circuit proved highly effective in optimizing suction performance while minimizing energy consumption. The custom-manufactured PCB was stable and reliable during practical application.

This electronic integration greatly enhanced the robot's intelligence and efficiency, enabling it to operate autonomously and effectively during cleaning tasks.

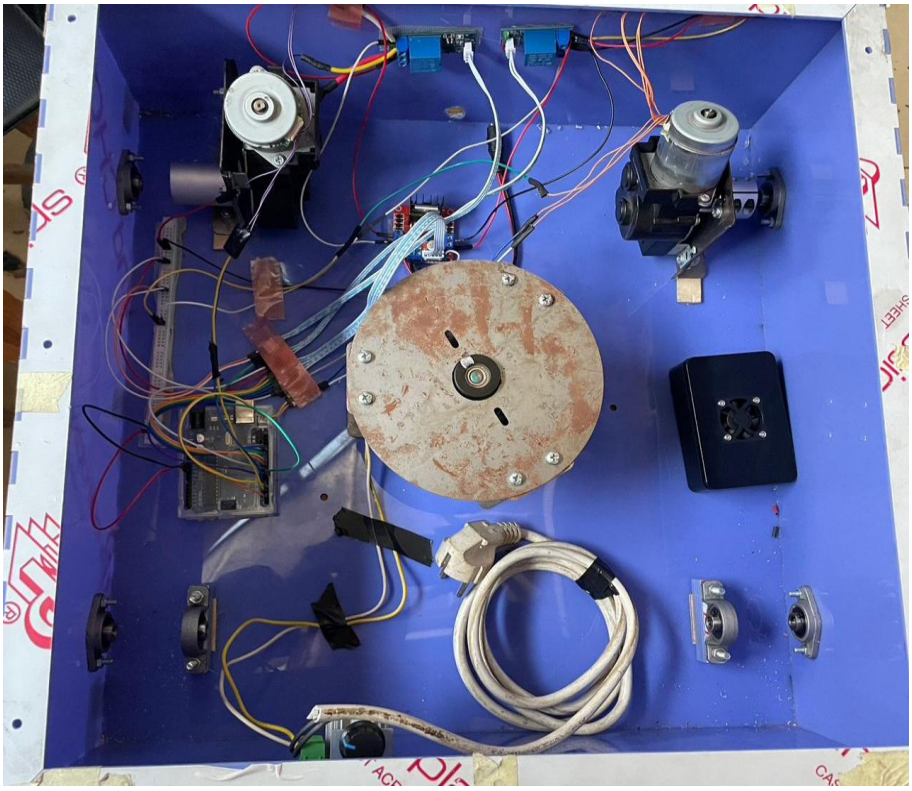


FIGURE 202

FIGURE 203

5.5 System Testing and Validation

The testing phase was a critical step in validating the performance of the window cleaning robot and ensuring that all its systems worked cohesively under real-world conditions. Comprehensive tests were conducted on the prototype in various environments to evaluate its functionality, stability, and overall efficiency. These tests were designed to simulate actual operating conditions and identify any areas requiring refinement or optimization.

5.5.1 Adhesion Performance Testing:

The robot's suction system was rigorously tested on different types of glass surfaces, including smooth, textured, and slightly uneven surfaces. The tests aimed to assess the strength and reliability of the adhesion force generated by the suction system.

- The robot demonstrated consistent and stable adhesion on all tested surfaces, maintaining a secure grip even during prolonged operation.
- Additional tests under inclined surfaces and varying environmental conditions (e.g., wind or dust) confirmed the effectiveness of the sealing system and the impeller in maintaining vacuum pressure.

5.5.2 Energy Consumption and Motor Efficiency:

The robot's energy consumption was monitored during operation to assess the efficiency of its motors and electronic systems.

- The SCR Dimmer circuit controlling the suction motor demonstrated excellent performance in adjusting the impeller speed based on operational requirements, reducing unnecessary power consumption.
- The brushless DC motors used for movement operated efficiently, delivering high torque with minimal power usage.

5.5.3 Stress Testing and Long-Term Operation:

To ensure durability, the robot was subjected to extended operation over several hours. These stress tests simulated prolonged cleaning tasks in commercial applications.

- The system maintained stable performance throughout the tests, with no significant overheating or wear observed in the motors, suction system, or structural components.
- The robot's lightweight design and balanced weight distribution contributed to its stability and durability during extended use.

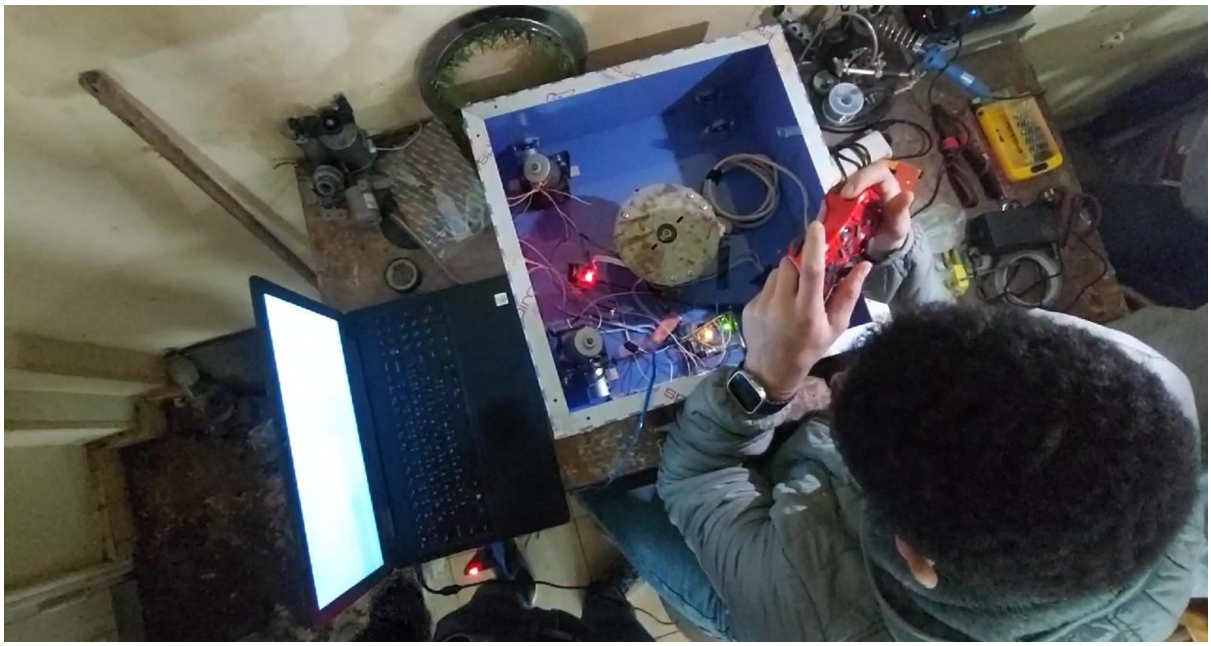


FIGURE 204

FIGURE 205

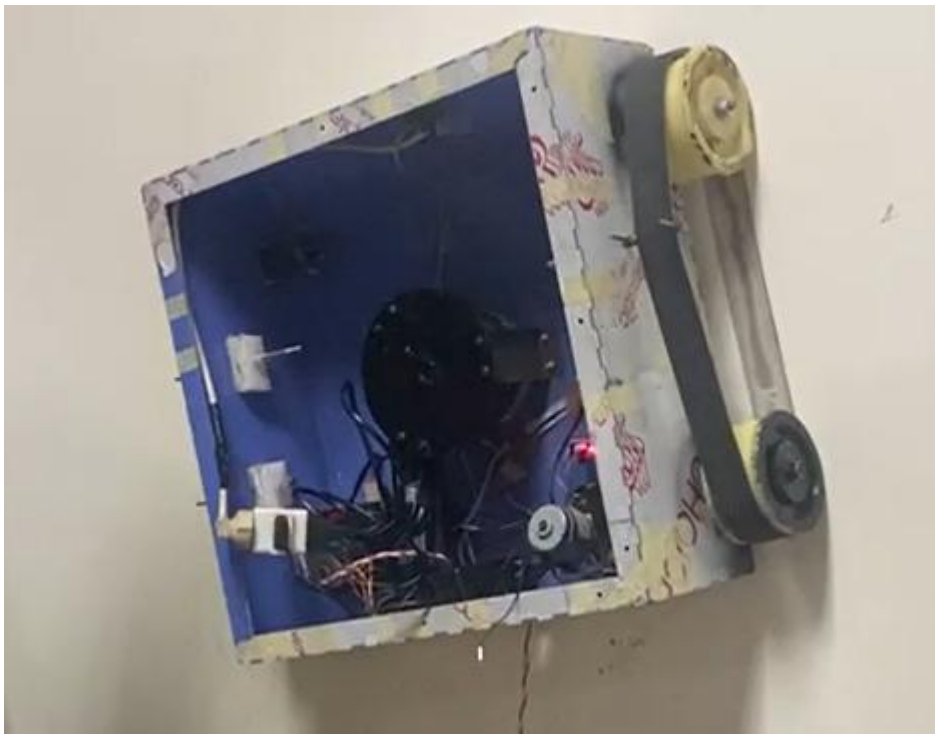


FIGURE 206

References

- *Nishi, A.; Wakasugi, Y.; Watanabe, K. Design of a robot capable of moving on a vertical wall. Adv. Robot. 1986, 1, 33–45.*
- *Schmidt, D.; Berns, K. Climbing robots for maintenance and inspections of vertical structures—A survey of design aspects and technologies. Robot. Auton. Syst. 2013, 61, 1288–1305.*
- *Zhang, L.; Sun, J.; Yin, G.; Zhao, J.; Han, Q. A cross structured light sensor and stripe segmentation method for visual tracking of a wall climbing robot. Sensors 2015, 15, 13725–13751.*
- *Huang, H.; Li, D.; Xue, Z.; Chen, X.; Liu, S.; Leng, J.; Wei, Y. Design and performance analysis of a tracked wall-climbing robot for ship inspection in shipbuilding. Ocean Eng. 2017, 131, 224–230.*
- *Kermorgant, O. A magnetic climbing robot to perform autonomous welding in the shipbuilding industry. Robot. Comput.-Integr. Manuf. 2018, 53, 178–186.*
- *Chen, X.; Wu, Y.; Hao, H.; Shi, H.; Huang, H. Tracked wall-climbing robot for calibration of large vertical metal tanks. Appl. Sci. 2019, 9, 2671.*
- *Fukui, R.; Yamada, Y.; Mitsudome, K.; Sano, K.; Warisawa, S. Hangrawler: Large-payload and high-speed ceiling mobile robot using crawler. IEEE Trans. Robot. 2020, 36, 1053–1066.*
- *Xin, L.; Pengli, L.; Yang, L.; Wang, C. Back-Stepping Fuzzy Adaptive Sliding Mode Trajectory Tracking Control for Wall-Climbing Robot. JCP 2019, 14, 662–679.*
- *Fierro, R.; Lewis, F.L. Control of a nonholonomic mobile robot using neural networks. IEEE Trans. Neural Netw. 1998, 9, 589–600.*
- *Katsuki, Y.; Yamamoto, M.; Ikeda, T. A trajectory tracking control of a mobile robot for verticalwalls. In 2010 World Automation Congress; IEEE: Piscataway, NJ, USA, 2010; pp. 1–6.*
- *Hong,S.; Choi, J.-S.; Kim, H.-W.; Won, M.-C.; Shin, S.-C.; Rhee, J.-S.; Park, H.-U. A path tracking control algorithm for underwater mining vehicles. J. Mech. Sci. Technol. 2009, 23, 2030–2037.*
- *Wang,S.; Zhang, S.; Ma, R.; Jin, E.; Liu, X.; Tian, H.; Yang, R. Remote control system based on the Internet and machine vision for tracked vehicles. J. Mech. Sci. Technol. 2018, 32, 1317–1331.*
- *Zou,T.; Angeles, J.; Hassani, F. Dynamic modeling and trajectory tracking control of unmanned tracked vehicles. Robot. Auton. Syst. 2018, 110, 102–111.*
- *Zhou,L.; Wang, G.; Sun, K.; Li, X. Trajectory Tracking Study of Track Vehicles Based on Model Predictive Control. Stroj. Vestn. J. Mech. Eng. 2019, 65, 329–342.*
- *Zhao, Z.; Liu, H.; Chen, H.; Xu, S.; Liang, W. Tracking control of unmanned tracked vehicle in off-road conditions with large curvature. In Proceedings of the 2019 IEEE Intelligent Transportation Systems Conference (ITSC), Auckland, New Zealand, 27–30 October 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 3867–3873.*

- *Sebastian, B.; Ben-Tzvi, P. Active disturbance rejection control for handling slip in tracked vehicle locomotion. J. Mech. Robot. 2019, 11, 021003.*
- *Sabiha, A.D.; Kamel, M.A.; Said, E.; Hussein, W.M. ROS-based trajectory tracking control for autonomous tracked vehicle using optimized backstep** and sliding mode control. Robot. Auton. Syst. 2022, 152, 104058.*
- *Sun,H.; Wang, H.; Xu, Y.; Yang, J.; Ji, Y.; Cheng, L.; Liu, C.; Zhang, B. Trajectory tracking control strategy of the gantry welding robot under the influence of uncertain factors. Meas. Control 2023, 56, 442–455.*
- *Kitano, M.; Kuma, M. An analysis of horizontal plane motion of tracked vehicles. J. Terramech. 1977, 14, 211–225.*
- *Kitano, M.; Jyozaki, H. A theoretical analysis of steerability of tracked vehicles. J. Terramech. 1976, 13, 241–258.*
- *Wong,J.; Chiang, C. Ageneral theory for skid steering of tracked vehicles on firm ground. Proc. Inst. Mech. Eng. Part D J. Automob. Eng. 2001, 215, 343–355.*
- *Kar, M.K. Prediction of track forces in skid-steering of military tracked vehicles. J. Terramech. 1987, 24, 75–84.*
- *Cui, D.; Wang, G.; Zhao, H.; Wang, S. Research on a Path-Tracking Control System for Articulated Tracked Vehicles. Stroj. Vestn. J. Mech. Eng. 2020, 66, 311–324.*
- *Wu,H.;Su, W.; Liu, Z. PID controllers: Design and tuning methods. In Proceedings of the 2014 9th IEEE Conference on Industrial Electronics and Applications 2014, Hangzhou, China, 9–11 June 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 808–813.*
- *Ang, K.H.; Chong, G.; Li, Y. PID control system analysis, design, and technology. IEEE Trans. Control Syst. Technol. 2005, 13, 559–576.*
- *https://surtrtech.com/2018/01/27/step-by-step-on-how-to-use-the-l298n-dual-h-bridge-driver-with-arduino/*
- *https://howtomechatronics.com/tutorials/arduino/stepper-motors-and-arduino-the-ultimate-guide/*
- *https://www.electronicwings.com/arduino/mpu6050-interfacing-with-arduino-uno*

Appendix

MATLAB Function

Part 1: Simulation

1: DIFFERENTIAL DRIVE CONTINUOUS SIMULATION

```
%% Define vehicle
R = 0.1;                % Wheel radius [m]
L = 0.5;                % Wheelbase [m]
dd = DifferentialDrive(R,L);

%% Run a continuous simulation using ODE45
initPose = [0 0 pi/4];  % Initial pose (x y theta)
tspan = [0 10];
[t,pose] =
ode45(@(t,y)diffDriveDynamics(t,y,dd),tspan,initPose);
pose = pose';

%% Display results
close all
figure
hold on
plot(pose(1,1),pose(2,1),'ro', ...
     pose(1,end),pose(2,end),'go', ...
     pose(1,:),pose(2,:),'b-');
axis equal
title('Vehicle Trajectory');
xlabel('X [m]')
ylabel('Y [m]')
legend('Start','End','Trajectory')


%% Continuous dynamics
function dy = diffDriveDynamics(t,y,vehicle)

    % Set desired velocities and solve inverse kinematics
    vDes = 0.2;
    if t < 5
       wDes = -0.5;
    else
       wDes = 0.5;
    end
    [wL,wR] = vehicle.inverseKinematics(vDes,wDes);

    % Calculate forward kinematics and convert the speeds to
global
    % coordinates
```

```
    [v,w] = vehicle.forwardKinematics(wL,wR);
    velB = [v;0;w];             % Body velocities [vx;vy;w]
    dy = bodyToWorld(velB,y);   % Convert from body to world

end
```

2: DIFFERENTIAL DRIVE DISCRETE SIMULATION

```
%% Define Robot
R = 0.1;                % Wheel radius [m]
L = 0.5;                % Wheelbase [m]
dd = DifferentialDrive(R,L);

%% Simulation parameters
sampleTime = 0.01;              % Sample time [s]
initPose = [0;0;pi/4];          % Initial pose (x y theta)

% Initialize time, input, and pose arrays
tVec = 0:sampleTime:10;         % Time array
vRef = 0.2*ones(size(tVec));    % Reference linear speed
wRef = zeros(size(tVec));       % Reference angular speed
wRef(tVec < 5) = -0.5;
wRef(tVec >=5) = 0.5;
pose = zeros(3,numel(tVec));    % Pose matrix
pose(:,1) = initPose;

%% Simulation loop
for idx = 2:numel(tVec)
    % Solve inverse kinematics to find wheel speeds
    [wL,wR] = inverseKinematics(dd,vRef(idx-1),wRef(idx-1));

    % Compute the velocities
    [v,w] = forwardKinematics(dd,wL,wR);
    velB = [v;0;w]; % Body velocities [vx;vy;w]
    vel = bodyToWorld(velB,pose(:,idx-1));  % Convert from
body to world

    % Perform forward discrete integration step
    pose(:,idx) = pose(:,idx-1) + vel*sampleTime;
end

%% Display results
close all
figure
hold on
plot(pose(1,1),pose(2,1),'ro', ...
     pose(1,end),pose(2,end),'go', ...
     pose(1,:),pose(2,:),'b-');
```

```
axis equal
title('Vehicle Trajectory');
xlabel('X [m]')
ylabel('Y [m]')
legend('Start','End','Trajectory')
```

3: SIMULATE THE TRACKED WHEEL MOBILE ROBOT WITH TIME T

```
% Set up the figure
figure;
axis equal;
axis([0 25 0 7]);
hold on;

% Define start and end positions
start_pos = 2; % Start position
end_pos = 15;  % End position

% Define the number of frames for the animation
num_frames = 100;

% Animation loop
for t = linspace(start_pos, end_pos, num_frames)

    % Clear the previous figure
    clf;

    % Draw the robot body: front rectangle (red)
    rectangle('Position', [t, 3, 3, 2], 'Curvature', 0.1,
'FaceColor', [1 0 0]);

    % Draw the robot body: rear rectangle (red)
    rectangle('Position', [t + 5, 3, 3, 2], 'Curvature', 0.1,
'FaceColor', [1 0 0]);

    % Draw the linkages between the two bodies
    line([t + 3 t + 5], [4 4], 'LineWidth', 2, 'Color', 'k',
'LineStyle', '--'); % Upper linkage
    line([t + 3 t + 5], [3.5 3.5], 'LineWidth', 2, 'Color',
'k', 'LineStyle', '--'); % Lower linkage

    % Draw the front wheels (blue)
    rectangle('Position', [t, 2.5, 3, 0.5], 'Curvature', [0,
0], 'FaceColor', [0 0 1]); % Left
    rectangle('Position', [t, 4.5, 3, 0.5], 'Curvature', [0,
0], 'FaceColor', [0 0 1]); % Right

    % Draw the rear wheels (blue)
```

```
    rectangle('Position', [t + 5, 2.5, 3, 0.5],
'Curvature', [0, 0], 'FaceColor', [0 0 1]); % Left
    rectangle('Position', [t + 5, 4.5, 3, 0.5], 'Curvature',
[0, 0], 'FaceColor', [0 0 1]); % Right

    % Add a star in the middle of the front body (yellow)
    star_shape_front = create_star(t + 1.5, 3.75, 0.6, 5); %
Center at (t + 1.5, 3.75), radius 0.6
    patch(star_shape_front(:, 1), star_shape_front(:, 2), [1 1
0]); % Yellow star

    % Add a star in the middle of the rear body (yellow)
    star_shape_rear = create_star(t + 6.5, 3.75, 0.6, 5); %
Center at (t + 6.5, 3.75), radius 0.6
    patch(star_shape_rear(:, 1), star_shape_rear(:, 2), [1 1
0]); % Yellow star

    % Adjust the axes to fit the drawing
    axis equal;
    axis([0 25 0 7]);

    % Add a title to the plot
    title('Two Tracked Wheel Mobile Robot Moving');

    % Add text at the start position
    text(start_pos, 5.5, 'Start', 'FontSize', 12, 'Color',
'g', 'FontWeight', 'bold');

    % Add text at the end position
    text(21.5, 5.5, 'End', 'FontSize', 12, 'Color', 'r',
'FontWeight', 'bold');

    % Pause to display the movement
    pause(0.05);
end

% Function to create a star shape
function star_shape = create_star(x_center, y_center, radius,
num_points)
    theta = linspace(0, 2*pi, num_points+1);
    outer_radius = radius;
    inner_radius = radius / 2.5;

    star_shape = zeros(2 * num_points, 2);
    for i = 1:num_points
        angle_outer = theta(i);
        angle_inner = theta(i) + pi/num_points;

        star_shape(2*i-1, :) = [x_center + outer_radius *
cos(angle_outer), y_center + outer_radius * sin(angle_outer)];
```

```
        star_shape(2*i, :) = [x_center + inner_radius *
cos(angle_inner), y_center + inner_radius * sin(angle_inner)];
    end
end
```

Part 2: Computed Torque Control (CTC)

```
Parameter of mobile robot (m.file)
%%
% Written By AG
clear
clc

%% Parameters
% The robot is assumed to be a regular cuboid and the wheels
are assumed to
% be regular cylinders.
L = 0.25;
R = 0.05;
d = 0.20;
mc = 5.00;
mw = 0.10;
Ic = (2/3)*mc*L^2;
Iw = 0.5*mw*R^2;
m = mc + 2*mw;
I = Ic + mc*d^2 + 2*mw*L^2;

%% Simulation
Data = sim('CTC', 20*pi);

% Right Wheel
f = figure('Name','Right Wheel','NumberTitle','off');
f.Position = [50 0 1500 800];
subplot(2, 3, 1)
plot(Data.position_1.Time, Data.position_1.Data)
grid
xlabel('Time(sec)')
ylabel('Position')
legend('Actual Angle','Desired Angle')
subplot(2, 3, 2)
plot(Data.position_1.Time, Data.velocity_1.Data)
grid
xlabel('Time(sec)')
ylabel('Velocity')
legend('Desired Velocity','Actual Velocity')
subplot(2, 3, 3)
plot(Data.position_1.Time, Data.acceleration_1.Data)
grid
```

```
xlabel('Time(sec)')
ylabel('Acceleration')
legend('Desired Acceleration','Actual Acceleration')
subplot(2, 3, 4)
plot(Data.position_1.Time, Data.position_1.Data(:,1)-
Data.position_1.Data(:,2))
grid
xlabel('Time(sec)')
ylabel('Position Error')
subplot(2, 3, 5)
plot(Data.position_1.Time, Data.velocity_1.Data(:,1)-
Data.velocity_1.Data(:,2))
grid
xlabel('Time(sec)')
ylabel('Velocity Error')
subplot(2, 3, 6)
plot(Data.position_1.Time, Data.acceleration_1.Data(:,1)-
Data.acceleration_1.Data(:,2))
grid
xlabel('Time(sec)')
ylabel('Acceleration Error')

% Left Wheel
f = figure('Name','Left Wheel','NumberTitle','off');
f.Position = [50 0 1500 800];
subplot(2, 3, 1)
plot(Data.position_2.Time, Data.position_2.Data)
grid
xlabel('Time(sec)')
ylabel('Position')
legend('Actual Angle','Desired Angle')
subplot (2, 3, 2)
plot (Data.position_2.Time, Data.velocity_2.Data)
grid
xlabel('Time(sec)')
ylabel('Velocity')
legend('Desired Velocity','Actual Velocity')
subplot (2, 3, 3)
plot (Data.position_2.Time, Data.acceleration_2.Data)
grid
xlabel('Time(sec)')
ylabel('Acceleration')
legend('Desired Acceleration','Actual Acceleration')
subplot(2, 3, 4)
plot(Data.position_2.Time, Data.position_2.Data(:,1)-
Data.position_2.Data(:,2))
grid
xlabel('Time(sec)')
ylabel('Position Error')
subplot (2, 3, 5)
```

```
plot(Data.position_2.Time, Data.velocity_2.Data(:,1)-
Data.velocity_2.Data(:,2))
grid
xlabel('Time(sec)')
ylabel('Velocity Error')
subplot (2, 3, 6)
plot(Data.position_2.Time, Data.acceleration_2.Data(:,1)-
Data.acceleration_2.Data(:,2))
grid
xlabel('Time(sec)')
ylabel('Acceleration Error')

%X-Y Visualization
t = Data.X.Time;
dx = Data.X.Data(:,1);
dy = Data.Y.Data(:,1);
dt = Data.Theta.Data(:,1);
xa = Data.X.Data(:,2);
ya = Data.Y.Data(:,2);
ta = Data.Theta.Data(:,2);
f = figure('Name','Robot Path','NumberTitle','off');
f.Position = [220 0 1150 800];
xlabel('X-Position')
ylabel('Y-Position')
axis([-4,4,-3,3])
hold on
hh = rectangle('Position',[xa(1) - 0.05,ya(1) -
0.05,0.1,0.1],'Curvature',1,...
    'FaceColor','#A2142F','EdgeColor','k','LineWidth',1);
o = hh;
h = hh;
k = hh;
kk = hh;

kkt = fix(length(t)/20);
kte = length(t);

for ii = 1:kkt
    delete(kk)
    delete(hh)
    kk = plot(xa(1:ii),ya(1:ii),'r-.','LineWidth',1.6);
    hh = rectangle('Position',[xa(1) - 0.05,ya(1) -
0.05,0.1,0.1],...
        'FaceColor','#A2142F','EdgeColor','k','LineWidth',1);
    legend('Desired Path')
    hold on
    drawnow
end

for ii = 1:kte
    delete(kk)
```

```
    delete(hh)
    delete(h)
    delete(k)
    delete(o)
    % Draw actual path
    if kkt + ii < kte
        kk = plot(xa(1:ii + kkt),ya(1:ii + kkt),'r-
.','LineWidth',1.6);
    else
        kk = plot(xa(1:end),ya(1:end),'r-.','LineWidth',1.6);
    end
    % Draw actual path
    k = plot(xa(1:ii),ya(1:ii),'g--','LineWidth',1.6);
    hh = rectangle('Position',[xa(1) - 0.05,ya(1) -
0.05,0.1,0.1],...
        'FaceColor','#A2142F','EdgeColor','k','LineWidth',1);
    % Draw robot
    h = rectangle('Position',[xa(ii) - 0.2,ya(ii) -
0.2,0.4,0.4],...

'Curvature',1,'FaceColor','w','EdgeColor','b','LineWidth',2.2)
;
    o = plot([xa(ii),xa(ii) - 0.35*cos(ta(ii))],[ya(ii),ya(ii)
- 0.35*sin(ta(ii))],...
        'k-','LineWidth',2.5);
    legend('Desired Path','Actual Path','Direction')
    hold on
    drawnow
end
```

FUNCTION 1: TRAJECTORY GENERATION

```
function [dX,dY,dTheta] = traject_gen(t)

    %TRAJECT_GEN Trajectory Generator
    %   TRAJECT_GEN(t) takes time samples as input argument
and return x
    %   position, y position, and the orientation for the
infinity path.

    dX = 3*cos(t/10)/(1 + sin(t/10)^2);                % X-
Position
    dY = 4*sin(t/10)*cos(t/10)/(1 + sin(t/10)^2);      % Y-
Position
    % Orientation
    dTheta = atan(-(12*sin(t/10)^2 -
4)/(3*sin(t/10)*(sin(t/10)^2 - 3))) - ~(t < 10*pi)*pi;

end
```

FUNCTION 2: INVERSE KINEMATICS

```
function [dqd1,dqd2] = inv_kin(dXd,dYd,dTheta_dot,dTheta,R,L)

    %INV_KIN Solve inverse kinematics
    %   INV_KIN(xd,yd,td,t) takes four input arguments
representing the
    %   robot velocities wrt the inertial frame, xd, yd, and
td. It also
    %   takes the robot orientation t, solve inverse
kinematics for these
    %   values, and then return the robot wheels velocities.

    forward_kinematics_matrix =
[R/2*cos(dTheta),R/2*cos(dTheta);

R/2*sin(dTheta),R/2*sin(dTheta);
                                    R/(2*L)        ,-R/(2*L)];
    inertial_frame_velocities = [dXd;dYd;dTheta_dot];

    wheels_velocities_matrix =
forward_kinematics_matrix\inertial_frame_velocities;

    dqd1 =  wheels_velocities_matrix(1);
    dqd2 =  wheels_velocities_matrix(2);
end
```

FUNCTION 3: INVERSE DYNAMICS

```
function [qdd1,qdd2] =
inverse_dynamics(D11,D22,D12,D21,C1,C2,T1,T2)

    %T_Matrix
    T = [T1;
         T2];

    %C_Matrix
    C = [C1;
         C2];

    %D_Matrix
    D = [D11,D12;
         D21,D22];

    %qdd_Matrix
    qdd = D \ (T - C);

    %Output
    qdd1 = qdd(1);
    qdd2 = qdd(2);

end
```

FUNCTION 4: FORWARD KINEMATICS

```
function [Xd,Yd,Theta_dot] = for_kin(qd1,qd2,Theta,R, L)

    %FOR_KIN Solve inverse kinematics
    %   FOR_KIN(xd,yd,td,t) takes three input arguments representing
the
    %   robot wheels velocities. It also takes the robot orientation
t,
    %   solve forward kinematics for these values, and then return
the
    %   robot velocities wrt to the inertial frame.

    forward_kinematics_matrix = [R/2*cos(Theta),R/2*cos(Theta);
                                 R/2*sin(Theta),R/2*sin(Theta);
                                 R/(2*L)       ,-R/(2*L)];
    wheels_velocities_matrix = [qd1;qd2];

    inertial_frame_velocities =
forward_kinematics_matrix*wheels_velocities_matrix;

    Xd = inertial_frame_velocities(1);
    Yd = inertial_frame_velocities(2);
    Theta_dot = inertial_frame_velocities(3);
end
```

Arduino Code

1. DC Motor

```
int in1 = 9;
int in2 = 8;
int ConA = 10;

void setup() {
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(ConA, OUTPUT);
}
//speed control range(0-255)
void speedA(){
  digitalWrite(in1, LOW);
  digitalWrite(in2, HIGH);
  analogWrite(ConA,50);
}
//changing speed and direction
void speedB(){
  digitalWrite(in1, HIGH);
  digitalWrite(in2, LOW);
  analogWrite(ConA,250);
}
//turning OFF
void turnOFF(){
  digitalWrite(in1, LOW);
  digitalWrite(in2, LOW);
  analogWrite(ConA,0);
}

void loop() {
  speedA();
  delay(2000);
  turnOFF();
  delay(2000);
  speedB();
  delay(2000);
  turnOFF();
  delay(2000);
}
```

2. Current Sensor

```
#include <Wire.h>
#include <Adafruit_INA219.h>
Adafruit_INA219 ina219;

void setup(void)
{
  Serial.begin(9600);
  while (!Serial)
  {
    delay(1);
  }

  // Initialize the INA219.
  if (! ina219.begin())
  {
    Serial.println("Failed to find INA219 chip");
    while (1)
    {
      delay(10);
    }
  }
  // To use a slightly lower 32V, 1A range (higher precision on amps):
  //ina219.setCalibration_32V_1A();
  // Or to use a lower 16V, 400mA range, call:
  //ina219.setCalibration_16V_400mA();
  Serial.println("Measuring voltage, current, and power with INA219 ...");
}

void loop(void)
{
  float shuntvoltage = 0;
  float busvoltage = 0;
  float current_mA = 0;
  float loadvoltage = 0;
  float power_mW = 0;

  shuntvoltage = ina219.getShuntVoltage_mV();
  busvoltage = ina219.getBusVoltage_V();
  current_mA = ina219.getCurrent_mA();
  power_mW = ina219.getPower_mW();
  loadvoltage = busvoltage + (shuntvoltage / 1000);

  Serial.print("Bus Voltage:   "); Serial.print(busvoltage); Serial.println("
V");
```

```
  Serial.print("Shunt Voltage: "); Serial.print(shuntvoltage);
Serial.println(" mV");
  Serial.print("Load Voltage:  "); Serial.print(loadvoltage); Serial.println("
V");
  Serial.print("Current:       "); Serial.print(current_mA); Serial.println("
mA");
  Serial.print("Power:         "); Serial.print(power_mW); Serial.println("
mW");
  Serial.println("");
  delay(1000);
}
```

---

3. IMU Sensor as Gyroscope

```
#include <Wire.h>
#include <MPU6050.h>
MPU6050 mpu;
// Timers
unsigned long timer = 0;
float timeStep = 0.01;
// Pitch, Roll and Yaw values
float pitch = 0;
float roll = 0;
float yaw = 0;
void setup()
{
  Serial.begin(115200);
  while(!mpu.begin(MPU6050_SCALE_2000DPS, MPU6050_RANGE_2G))
  {
    Serial.println("Could not find a valid MPU6050 sensor, check wiring!");
    delay(500);
  }
  // Calibrate gyroscope. The calibration must be at rest.
  // If you don't want calibrate, comment this line.
  mpu.calibrateGyro();
  // Set threshold sensivty. Default 3.
  // If you don't want use threshold, comment this line or set 0.
  // mpu.setThreshold(3);
}
void loop()
{
  timer = millis();
  // Read normalized values
  Vector norm = mpu.readNormalizeGyro();
  // Calculate Pitch, Roll and Yaw
  pitch = pitch + norm.YAxis * timeStep;
```

```
  roll = roll + norm.XAxis * timeStep;
  yaw = yaw + norm.ZAxis * timeStep;
  // Output raw
  Serial.print(" Pitch = ");
  Serial.print(pitch);
  Serial.print(" Roll = ");
  Serial.print(roll);
  Serial.print(" Yaw = ");
  Serial.println(yaw);
  // Wait to full timeStep period
  delay((timeStep*1000) - (millis() - timer));
}
```

4. IMU Sensor as Accelerometer

```
#include <Wire.h>
#include <MPU6050.h>

MPU6050 mpu;

void setup()
{
  Serial.begin(115200);

  Serial.println("Initialize MPU6050");

  while(!mpu.begin(MPU6050_SCALE_2000DPS, MPU6050_RANGE_2G))
  {
    Serial.println("Could not find a valid MPU6050 sensor, check wiring!");
    delay(500);
  }

  // If you want, you can set accelerometer offsets
  // mpu.setAccelOffsetX();
  // mpu.setAccelOffsetY();
  // mpu.setAccelOffsetZ();

  checkSettings();
}

void checkSettings()
{
  Serial.println();

  Serial.print(" * Sleep Mode:            ");
  Serial.println(mpu.getSleepEnabled() ? "Enabled" : "Disabled");
```

```
  Serial.print(" * Clock Source:          ");
  switch(mpu.getClockSource())
  {
    case MPU6050_CLOCK_KEEP_RESET:     Serial.println("Stops the clock and
keeps the timing generator in reset"); break;
    case MPU6050_CLOCK_EXTERNAL_19MHZ: Serial.println("PLL with external
19.2MHz reference"); break;
    case MPU6050_CLOCK_EXTERNAL_32KHZ: Serial.println("PLL with external
32.768kHz reference"); break;
    case MPU6050_CLOCK_PLL_ZGYRO:      Serial.println("PLL with Z axis
gyroscope reference"); break;
    case MPU6050_CLOCK_PLL_YGYRO:      Serial.println("PLL with Y axis
gyroscope reference"); break;
    case MPU6050_CLOCK_PLL_XGYRO:      Serial.println("PLL with X axis
gyroscope reference"); break;
    case MPU6050_CLOCK_INTERNAL_8MHZ:  Serial.println("Internal 8MHz
oscillator"); break;
  }

  Serial.print(" * Accelerometer:         ");
  switch(mpu.getRange())
  {
    case MPU6050_RANGE_16G:            Serial.println("+/- 16 g"); break;
    case MPU6050_RANGE_8G:             Serial.println("+/- 8 g"); break;
    case MPU6050_RANGE_4G:             Serial.println("+/- 4 g"); break;
    case MPU6050_RANGE_2G:             Serial.println("+/- 2 g"); break;
  }

  Serial.print(" * Accelerometer offsets: ");
  Serial.print(mpu.getAccelOffsetX());
  Serial.print(" / ");
  Serial.print(mpu.getAccelOffsetY());
  Serial.print(" / ");
  Serial.println(mpu.getAccelOffsetZ());

  Serial.println();
}

void loop()
{
  Vector rawAccel = mpu.readRawAccel();
  Vector normAccel = mpu.readNormalizeAccel();
  Serial.print(" Xraw = ");
  Serial.print(rawAccel.XAxis);
  Serial.print(" Yraw = ");
  Serial.print(rawAccel.YAxis);
  Serial.print(" Zraw = ");
  Serial.println(rawAccel.ZAxis);
```

```
  Serial.print(" Xnorm = ");
  Serial.print(normAccel.XAxis);
  Serial.print(" Ynorm = ");
  Serial.print(normAccel.YAxis);
  Serial.print(" Znorm = ");
  Serial.println(normAccel.ZAxis);
  delay(10);
}
```

5. DC Motor with encoder

```
volatile unsigned int temp, counter = 0; //This variable will increase or
decrease depending on the rotation of encoder
int in1 = 9;
int in2 = 8;
int ConA = 10;
void setup() {
  Serial.begin (9600);
  pinMode(2, INPUT_PULLUP); // internal pullup input pin 2
  pinMode(3, INPUT_PULLUP); // internalเป็นpullup input pin 3
//Setting up interrupt
  //A rising pulse from encodenren activated ai0(). AttachInterrupt 0 is
DigitalPin nr 2 on moust Arduino.
  attachInterrupt(0, ai0, RISING);
  //B rising pulse from encodenren activated ai1(). AttachInterrupt 1 is
DigitalPin nr 3 on moust Arduino.
  attachInterrupt(1, ai1, RISING);
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  pinMode(ConA, OUTPUT);
  }
  //speed control range(0-255)
void speedA(){
  digitalWrite(in1, LOW);
  digitalWrite(in2, HIGH);
  analogWrite(ConA,255);
}
//changing speed and direction
void speedB(){
  digitalWrite(in1, HIGH);
  digitalWrite(in2, LOW);
  analogWrite(ConA,250);
}
//turning OFF
void turnOFF(){
  digitalWrite(in1, LOW);
  digitalWrite(in2, LOW);
```

```
  analogWrite(ConA,0);
}
  void loop() {
  // Send the value of counter
  if( counter != temp ){
  Serial.println (counter);
  temp = counter;
  }
  speedA();
  }
  void ai0() {
  // ai0 is activated if DigitalPin nr 2 is going from LOW to HIGH
  // Check pin 3 to determine the direction
  if(digitalRead(3)==LOW) {
  counter++;
  }else{
  counter--;
  }
  }
  void ai1() {
  // ai0 is activated if DigitalPin nr 3 is going from LOW to HIGH
  // Check with pin 2 to determine the direction
  if(digitalRead(2)==LOW) {
  counter--;
  }else{
  counter++;
  }
  }
```

ROS 2 PACKAGE

https://github.com/AhmedGaberAG