Mohamed Khider University of Biskra

Faculty of Science and Technology

Department of Electrical Engineering

MATER'S THESIS

Science and Technology

Telecommunications

Networks and telecommunications

Presented and supported by:

BEZZIOU ABDENNACER

Submitted on: 10/06/2025

ROS-Powered Autonomous Robot Simulation

Before The Board of Juries:

M.OUAFI ABDELKARIM	Pr	University of Biskra	President
M.BEKHOUCHE KHALED	MCA	University of Biskra	Supervisor
M.BENAKCHA ABDELHAMID	Pr	University of Biskra	Examiner

Academic year: 2024 - 2025

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to **Mr. Bekhouche Khaled**, my supervisor, for his valuable guidance, continuous support, and encouragement throughout the preparation of this thesis. His insights and expertise were essential to the successful completion of this work.

I am also grateful to **Mr. Ouafi Abdelkarim**, President of the jury, and **Mr. Benakcha Abdelhamid**, Examiner, for accepting to evaluate my work and for their constructive comments and suggestions.

A special thanks to the staff of the **Department of Electrical Engineering** at **Mohamed Khider University of Biskra** for providing the academic environment and resources needed for this research.

I would like to express my heartfelt appreciation to my family, especially my parents, for their unconditional love, patience, and unwavering support during all stages of my studies.

Finally, I extend my thanks to my friends and colleagues for their encouragement, help, and the many discussions that enriched my experience during this journey.

DEDICATION

To my parents

To the persons I owe everything to. To the persons whom I stood tongueless to express how grateful I am for everything they did for me and for making me who I am today.

May God Bless You Beyond Any Known Measure

To my brothers and sisters

To the strongest bond of my life. To my source of joy, happiness, strength and will. To the persons who stood for me and been my shield whenever I needed them.

May The Lord Protect you

To my classmates

To my friends whom I shared every moment of joy and sorrow. To my friends whom I spent my journey with.

May God Be Before You To Guide You

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

List of figures

List of figures

List of figures

List of abbreviations

- ADC: Analog to Digital Converter
- AMCL: Adaptive Monte Carlo Localization
- AMR: Autonomous Mobile Robot
- ASIC: Application-Specific Integrated Circuit
- AUAV: Autonomous Unmanned Aerial Vehicle
- CCD: Charge Coupled Device
- CMOS: Complementary Metal-Oxide Semiconductor
- DAC: Digital to Analog Converter
- DDS: Data Distribution Service
- DOF: Degrees of Freedom
- FMCW: Frequency-Modulated Continuous-Wave
- FOG: Fiber-Optic Gyroscope
- GL: Gray Level
- GNSS: Global Navigation Satellite System
- GPS: Global Positioning System
- I2C: Inter-Integrated Circuit
- ICC: Instantaneous Center of Curvature
- IMU: Inertial Measurement Unit
- IR: Infrared
- LED: Light Emitting Diode
- LiDAR: Light Detection and Ranging
- MEMS: Microelectromechanical Systems
- NIR: Near-Infrared
- OPA: Optical Phased Array
- PGM: Portable Gray Map
- PID: Proportional-Integral-Derivative
- RCL: ROS Client Library
- RGB: Red, Green, Blue
- RGB-D: Red, Green, Blue, and Depth
- ROS: Robot Operating System
- RTOS: Real-Time Operating System
- RRT: Rapidly-exploring Random Tree

List of abbreviations

- SL: Structured Light
- SLAM: Simultaneous Localization and Mapping
- TDOA: Time Difference of Arrival
- TF: Transform
- TOF: Time of Flight
- UAV: Unmanned Aerial Vehicle
- UHF: Ultra High Frequency
- UWB: Ultra-wideband
- ADC: Analog to Digital Converter
- AMCL: Adaptive Monte Carlo Localization
- AMR: Autonomous Mobile Robot
- CCD: Charge Coupled Device
- CMOS: Complementary Metal-Oxide Semiconductor
- COLLADA: Collaborative Design Activity
- DDS: Data Distribution Service
- DOF: Degrees of Freedom
- FPS: Frames Per Second
- Gazebo: 3D Robot Simulator
- GNSS: Global Navigation Satellite System
- HSV: Hue, Saturation, Value
- IMU: Inertial Measurement Unit
- LIDAR: Light Detection and Ranging
- Nav2: Navigation2 Stack
- PGM: Portable Gray Map
- PID: Proportional-Integral-Derivative
- QoS: Quality of Service
- RCL: ROS Client Library
- REP: ROS Enhancement Proposal
- ROS2: Robot Operating System 2
- RTF: Real Time Factor
- RViz2: ROS Visualization Tool 2
- SDF: Simulation Description Format

List of abbreviations

- SLAM: Simultaneous Localization and Mapping
- TF: Transform
- URDF: Unified Robot Description Format
- YAML: YAML Ain’t Markup Language
- VSG: Vibrating Structure Gyroscope

الملخص

تستعرض هذه المذكرة تكامل الأنظمة المدمجة والروبوتات، مع التركيز على دورها في تمكين الأتمتة المتقدمة والأنظمة الذكية .تُعد الأنظمة المدمجة، التي تتميز بقيود الوقت الحقيقي وكفاءة استخدام الموارد، العمود الفقري الحسابي للروبوتات، حيث تتيح التحكم الدقيق والتفاعل مع البيئات الديناميكية .يتناول هذا البحث المكونات المادية والبرمجية للأنظمة المدمجة، وتكاملها مع الروبوتات، وتصميم روبوت بعجلتين يعمل بالدفع التفاضلي كمثال تطبيقي .كما يتم تحليل تقنيات الاستشعار الحيوية لإدراك الروبوتات المتنقلة، والملاحة، والتفاعل، مع التركيز على أجهزة الاستشعار القريبة، وأجهزة القياس، وأجهزة الاستشعار المعتمدة على الرؤية .ويُناقش إطار عمل نظام تشغيل الروبوتات 2 (ROS2)من حيث بنيته الموزعة، وتقنيات التوطين، وقدرات الملاحة، وذلك بدعم من أدوات مثل Gazebo و Rviz.ومن خلال دراسات حالة، يسلط المشروع الضوء على التطبيقات الواقعية لهذه التقنيات في الروبوتات المتنقلة، والطائرات بدون طيار، والمركبات الذاتية القيادة، مبرزًا أثرها في تطور علم الروبوتات الحديث.

ABSTRACT

This thesis explores the integration of embedded systems and robotics, focusing on their role in enabling advanced automation and intelligent systems. Embedded systems, characterized by real-time constraints and resource efficiency, form the computational backbone of robotics, facilitating precise control and interaction with dynamic environments. The study delves into the hardware and software components of embedded systems, their synergy with robotics, and the design of a two-wheeled differential drive robot as a practical example. It further examines sensor technologies critical for mobile robot perception, navigation, and interaction, covering proximity, ranging, and vision-based sensors. The Robot Operating System 2 (ROS2) framework is analyzed for its distributed architecture, localization techniques, and navigation capabilities, supported by tools like Gazebo and Rviz. Through case studies, the document highlights real-world applications of these technologies in mobile robots, UAVs, and autonomous vehicles, demonstrating their impact on modern robotics

INTRODUCTION

Over the past few decades, the world has witnessed a remarkable revolution in the field of robotics, driven by the growing interest of students, educators, scientists, and researchers alike. Robots have demonstrated their effectiveness in diverse domains such as manufacturing, space exploration, home automation, and medical surgeries, motivating researchers to further expand their applications to new areas. Among the various categories of robots, wheeled autonomous mobile robots stand out as one of the most widely used types due to their simplicity, efficiency, and adaptability. The central aim of this thesis is to design and implement an autonomous robot capable of performing one of the most essential tasks for true autonomy: autonomous mobile robot navigation.

The first chapter introduces embedded systems and their critical role in robotics, describing them as specialized computing platforms characterized by real-time constraints and efficient resource usage. It covers their hardware components (memory, microcontrollers, and peripherals) and software elements (firmware and Real-Time Operating Systems), while also discussing their classification and integration with robotics through applications such as industrial assembly, waste management, and surgery. A prototype of a two-wheeled differential drive robot is also presented, with an explanation of its hardware (chassis, DC motors, IMU) and software, including differential drive kinematics.

The second chapter focuses on sensor technologies essential for mobile robot perception and navigation. Sensors are categorized by excitation signal, measurement domain, and type, including proximity sensors, encoders, GNSS, IMUs, cameras, and LiDAR. The chapter also discusses specifications such as accuracy and resolution, highlighting their role in enabling robots to perceive, localize, and navigate within dynamic environments.

The third chapter presents the Robot Operating System 2 (ROS 2), with emphasis on its fundamentals, localization, and navigation capabilities. It outlines its distributed architecture (DDS, RCL), core concepts (nodes, topics, services), and navigation frameworks such as AMCL and the Navigation2 (Nav2) stack, supported by tools like Gazebo and RViz. Practical examples of their applications in mobile robots are also included.

The fourth chapter describes the simulation of the TurtleBot3 Waffle, a differential-drive mobile robot, within the ROS 2 Humble environment using Gazebo. It explains the

setup of Ubuntu and ROS 2, workspace configuration, development of Python/C++ nodes for communication, teleoperation, and map generation using SLAM (Cartographer). Furthermore, it evaluates simulation performance, particularly in terms of localization accuracy and navigation efficiency, while also discussing future improvements to enhance the robot's reliability and reduce uncertainty.

Finally, the thesis concludes with a discussion of results and perspectives for future work, highlighting potential directions to improve accuracy, reliability, and adaptability, thereby reinforcing the foundations of autonomous mobile robotics.

Chapter I: Embedded Systems and Robotics

1.1 Introduction

Embedded systems and robotics represent two interconnected fields that drive innovation in automation, control, and intelligent systems. Embedded systems are specialized computing platforms designed for specific tasks, characterized by their integration into larger systems, real-time constraints, and resource efficiency. Robotics, on the other hand, focuses on the design and operation of robots—mechanical systems equipped with sensors, actuators, and embedded electronics to interact with their environment. The synergy between embedded systems and robotics enables the development of advanced robotic systems capable of performing complex tasks in diverse applications, from industrial automation to domestic assistance. This document explores the fundamentals of embedded systems, their role in robotics, and the design of a two-wheeled differential drive robot as a practical example.[1]

1.2 Embedded Systems Overview

Embedded systems are specialized computing systems designed to perform dedicated functions within a larger system. These systems are characterized by their real-time operations, resource constraints, and integration into hardware components. Here's an overview of the key elements in both **hardware** and **software** for embedded systems: .[2]

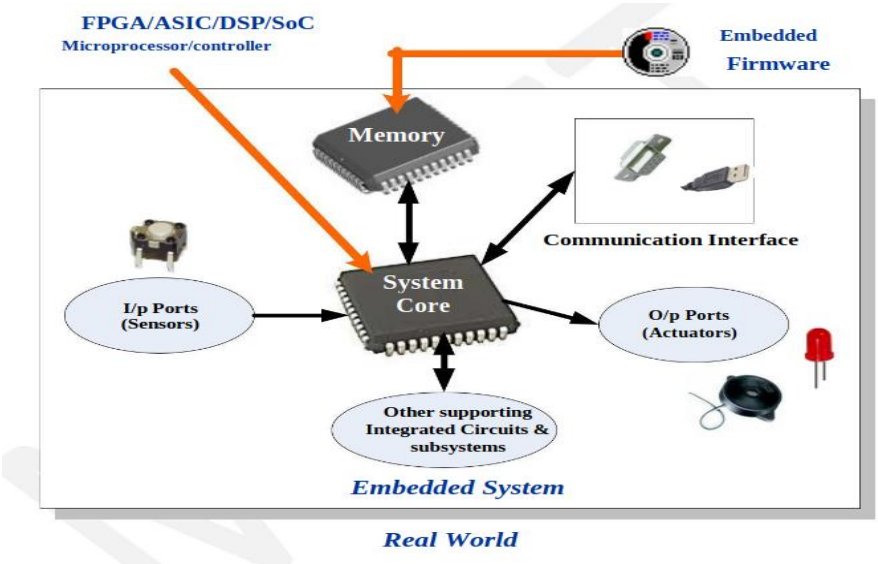


Figure1.1: Block Diagram of an Embedded System

1.3 embedded Systems Hardware

1.3.1 Memory:

- **RAM (Random Access Memory):** Temporary storage used to store data that the processor is currently working with.
- **ROM (Read-Only Memory):** Permanent storage for firmware and other crucial system information.
- **Flash Memory:** Non-volatile memory used for storing code and data that doesn't change often.

1.3.2 Computer:

- **Microprocessor:** The central unit of a computer system, responsible for executing instructions. It typically requires external components (RAM, I/O) to function.
- **Microcontroller:** A compact integrated circuit that combines a processor, memory, and peripherals, designed for embedded applications. Unlike microprocessors, microcontrollers are self-contained.
- **Microprocessor vs. Microcontroller:**
- **Microprocessor:** Used in general-purpose systems, needs external peripherals for full functionality.
- **Microcontroller:** Used in embedded systems, integrates CPU, memory, and peripherals in one chip, optimized for specific tasks.

. Peripherals:

- **Analog to Digital Converters (ADC):** Converts analog signals (e.g., from sensors) into a digital format for processing by the system.
- **Digital to Analog Converters (DAC):** Converts digital signals back to analog form, often used to control devices like motors or speakers.
- **I/O Devices:** Interfaces for communication between the embedded system and external devices (e.g., sensors, displays, keyboards).[2]

1.4 Embedded Systems Software

1.4.1 Application Software:

- These are software programs written to perform specific tasks or control the embedded system's hardware. Examples include software to monitor sensor data, control actuators, or manage communication between devices.

1.4.2 Middleware:

- Middleware provides software interfaces between application software and system hardware. It allows for communication between different hardware components and software layers, often hiding the complexity of system interactions.

1.4.3 Firmware:

- Firmware is low-level software that is permanently stored in the system's memory (e.g., in ROM or Flash memory). It controls the hardware and provides a basic operational platform for higher-level software. Firmware is typically responsible for the boot-up process, hardware initialization, and basic operations.

1.4.4 RTOS (Real-Time Operating System)

- **RTOS:** A Real-Time Operating System is software designed to manage hardware resources and execute tasks in a guaranteed time frame. Embedded systems often require RTOS to handle real-time requirements, such as controlling devices within precise time constraints. .[3]

1.5 Basic Structure of an Embedded System

Let's see the block diagram shows the basic structure of an embedded system.

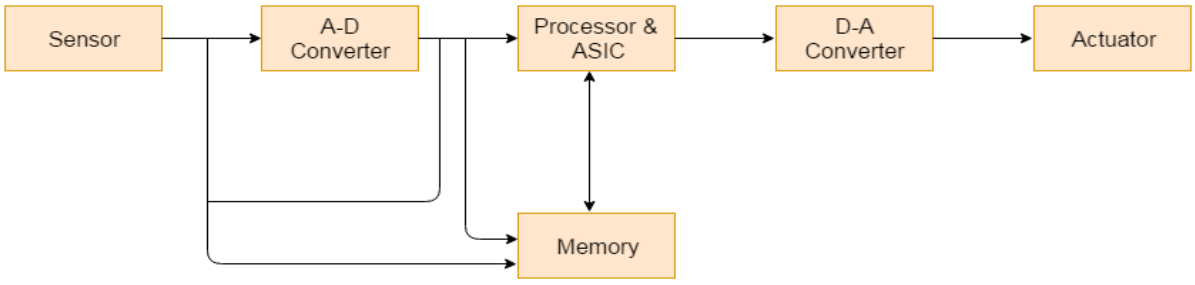


Figure1.2: Simplified Block Diagram of an Embedded System

- **Sensor**: Sensor used for sensing the change in environment condition and it generate the electric signal on the basis of change in environment condition. Therefore, it is also called as transducers for providing electric input signal on the basis of change in environment condition.
- **A-D Converter**: An analog-to-digital converter is a device that converts analog electric input signal into its equivalent digital signal for further processing in an embedded
- **Processor & ASICs**: Processor used for processing the signal and data to execute desired set of instructions with high-speed of operation. Application specific integrated circuit (ASIC) is an integrated circuit designed to perform task specific operation inside an embedded system.
- **D-A Converter**: A digital-to-analog converter is a device that converts digital electric input signal into its equivalent analog signal for further processing in an embedded system.
- **Actuators**: Actuators is a comparator used for comparing the analog input signal level to desired output signal level for providing the error free output from the system.[4]

1.6 Intersection of Embedded Systems and Robotics

The effects of embedded systems and robotics are mutually supportive and synergistic. Robotics requires embedded systems to perform various applications in challenging or unpredictable environments with strict real-time constraints. Innovations in embedded system design such as the modernization of electronics, enhanced processing capabilities, and network connectivity have paved the way for the creation of the next generation of powerful and flexible robotic systems.

Today, the capability and competence of a robot are heavily determined by the sophistication of the embedded systems within it. These systems serve as the central processing unit that coordinates sensor input, delivers algorithms, and facilitates communication both within the robot and with external systems. Embedded systems are fundamental to the robot's ability to function, and their role will continue to be crucial as robotics continues to advance.[5]

1.6.1 Robotics

Robotics is a branch of engineering and science involving the design, construction, operation, and application of robots. Robots are mechanical structures carrying electronic systems with features such as sensors and actuators that sense the outside environment and interact with it. The development of robotics has been from simple-this, before programming-machines into highly autonomous systems capable of performing complex tasks involving object recognition, adaptability in motion control, and decision-making in the future, where embedded systems have been among the driving engines for the developments. .[6]

1.6.2 Main Tasks of Robots

Robots are designed to handle various tasks that are often risky, repetitive, or require precision, providing significant benefits in environments where human safety, efficiency, and productivity are key concerns. The primary tasks of robots can be summarized by the "5 D's": dirty, dull, dangerous, domestic, and dexterous. These categories describe the different types of tasks robots are often used for in human society.[7]

1.6.2.1 Dull

Robots are ideal for "dull" tasks that involve high repetition and low human interaction. These tasks often require minimal thinking and are typically focused on maximizing efficiency and output. Examples include assembly line work, packaging, or sorting. Robots are well-suited for these tasks because they can work continuously, improving productivity and freeing humans for more cognitive and varied work.

1.6.2.2 Dirty

"Dirty" tasks refer to jobs that are unsanitary, hazardous, or unpleasant. These are typically jobs that human would prefer not to do due to health or safety concerns. Robots can take over tasks such as waste management, cleaning up hazardous materials, and working in environments like sewers or toxic sites. By using robots in these scenarios, human workers are removed from dangerous conditions, reducing the risk of injury or illness.

1.6.2.3 Dangerous:

Robots are also employed in "dangerous" tasks where humans would be at high risk. This category includes activities like bomb disposal, deep-sea exploration, space missions, or working in extreme environments, such as nuclear power plants. Robots can perform these

high-risk tasks safely, preventing loss of human life and ensuring that operations in hazardous areas can continue without putting people in harm's way.

1.6.2.4 Domestic:

In the domestic category, robots are used to perform household tasks that help improve daily life for humans. Examples include cleaning robots (e.g., vacuum cleaners), lawn mowers, and home assistants. These robots reduce the amount of manual labor required for routine chores, offering convenience and allowing people to focus on other activities.

1.6.2.5 Dexterous:

Robots with "dexterous" capabilities are designed to perform tasks that require fine motor skills and precision. These robots can handle tasks that demand delicacy, such as performing surgeries, assembling small electronic components, or manipulating objects with great care. These robots are capable of performing tasks that would be difficult or impossible for humans to do with the same level of accuracy or consistency.

Each of these categories illustrates how robots are transforming various industries and improving human productivity, safety, and well-being by taking on tasks that are too dangerous, repetitive, or precise for humans.

1.7 Types of Mobile Robots:

A custom robot design often starts with a "vision" of what the robot will look like and what it will do. The types of robots possible are unlimited, though the more popular are:

. Land-based wheeled robot
. Land-based tracked robot
. Land-based legged robot
. Air-based: plane, helicopter, blimp
. Water-based; boat, submarine
. Misc. and combination robot
. Stationary robot (arm, manipulator, etc.).[8]

1.7.1 Wheels:

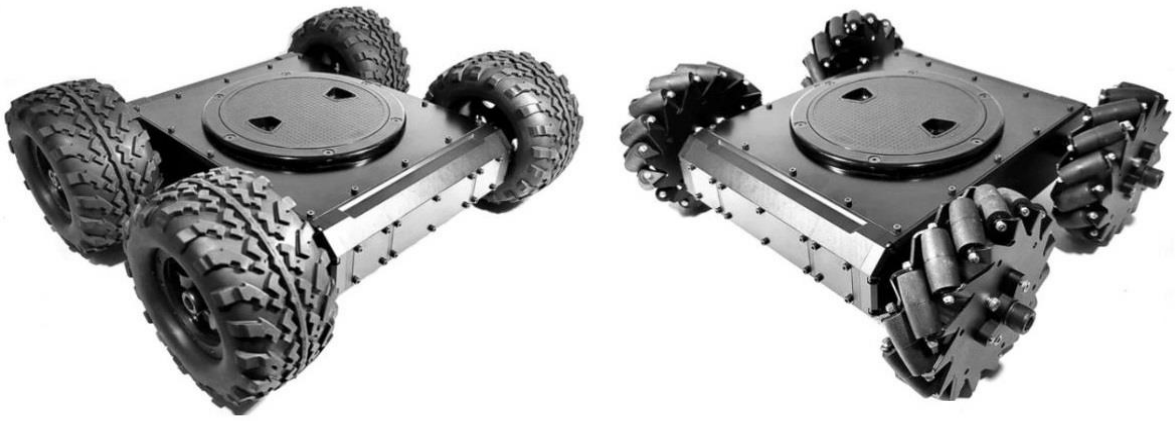


Figure 1.3: Land-based Wheeled Robot

Land-based wheeled robots are the most popular mobile robots among beginners as they usually require the least investment while providing significant exposure to robotics. The most complex type of robot is the autonomous humanoid (resembling a human), as it requires many degrees of freedom, synchronizing many motors and many sensors. This section is intended to help you decide what type of robot to build. Once you have chosen the type of robot, you can brainstorm what tasks/functions you want it to complete. .[9]

1.7.2 Tracks:

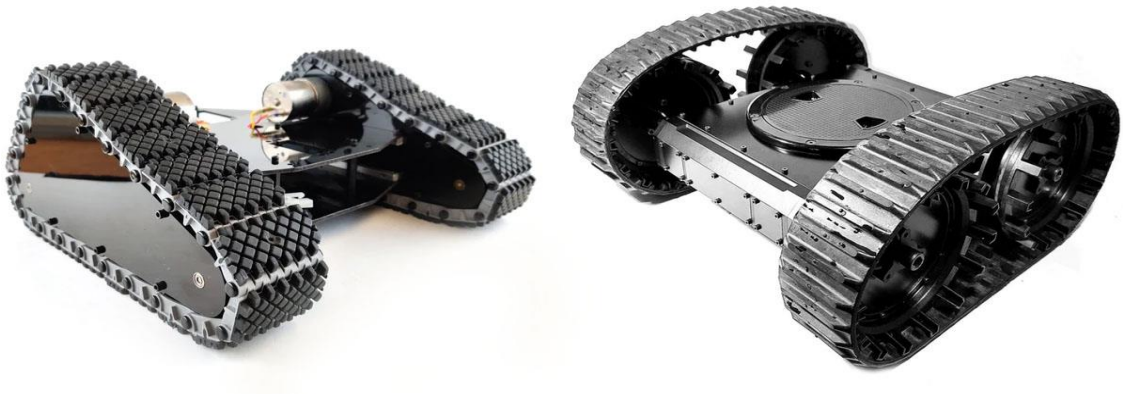


Figure 1.4: Land-based Tracked Robot

Tracks (or treads) are similar to what tanks use. Track drive is best for robots used outdoors and on soft ground. Although tracks do not provide added "force", they do reduce slip and more evenly distribute the weight of the robot, making them useful for loose surfaces such as sand and gravel. Most people tend to agree that tank tracks add an "aggressive" look to the robot as well. .[10]

1.7.3 Legs:

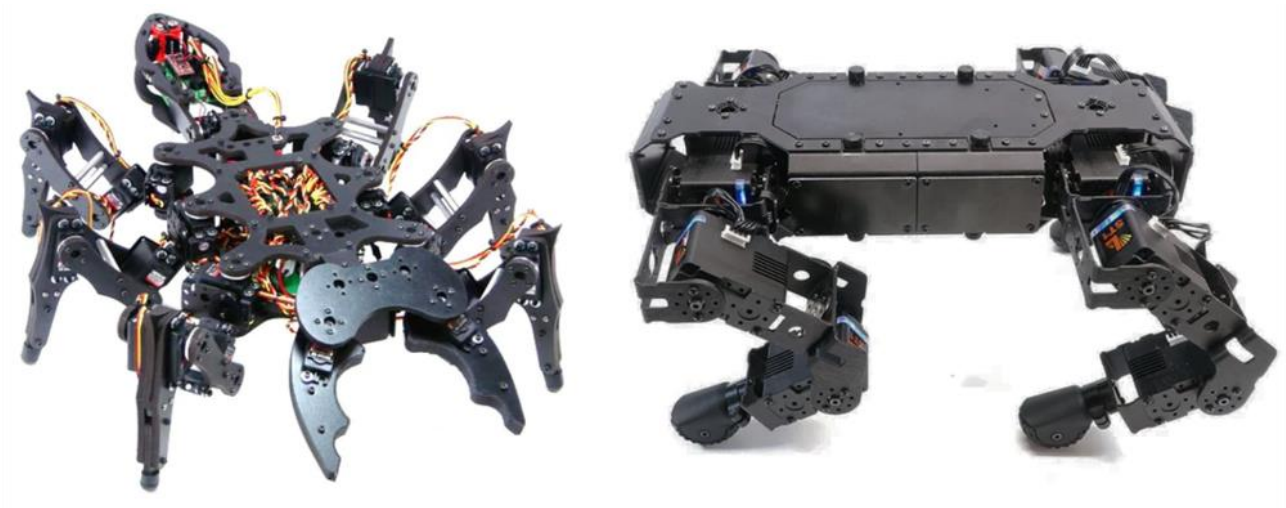


Figure 1.5: Land-based Legged Robot

An increasing number of robots use legs for mobility. Legs are often preferred for robots that must navigate on very uneven terrain. Most amateur robots are designed with six legs, which allow the robot to be statically balanced (balanced at all times on 3 legs). Robots with fewer legs are harder to balance. Researchers have experimented with monopod (one-legged "hopping") designs, though bipeds (two legs) and quadrupeds (four legs), and hexapods (6 legs) are the most popular. .[11]

1.7.4 Air-based:

Figure 1.6: Air-based: plane, helicopter, blimp

An AUAV (Autonomous Unmanned Aerial Vehicle) is very appealing and is entirely within the capability of many robot enthusiasts. However, the advantages of building autonomous unmanned aerial vehicles, especially if you are a beginner, have yet to outweigh the risks. High-altitude AUAV blimps and aircraft may one day be used for communication. When considering an aerial vehicle, most hobbyists still use existing commercial remote-controlled aircraft. Aircraft such as the US military Predator were initially semi-autonomous though in recent years Predator aircraft have flown missions autonomously.[12]

1.7.5 Water-based:

Figure 1.7: Water-based; boat, submarine

An increasing number of hobbyists, institutions, and companies are developing unmanned underwater vehicles. There are many obstacles yet to overcome to make underwater robots attractive to the wider robotic community though in recent years, several companies have commercialized pool-cleaning robots. Underwater vehicles can use ballast (compressed air and flooded compartments), thrusters, tails, fins, or even wings to submerge. Other aquatic robots such as pool cleaners are useful commercial products. .[13]

1.7.6 Miscellaneous and combination / hybrid:

Figure 1.8: Misc. and combination robot

A robot idea may not neatly fit into any of the previously mentioned categories, or it could consist of several distinct functional sections. It is important to note that this typs of robot is focused on mobile robots, rather than stationary or permanently fixed designs, with the exception of robotic arms and grippers. When developing a combination or hybrid design, the use of a modular approach is highly recommended. In this case, each functional component can be detached and tested independently, allowing for more flexible testing and troubleshooting.

Miscellaneous designs may encompass a wide range of concepts. These could include hovercrafts, snake-like mechanisms, turrets, or even more unconventional structures. Such designs allow for greater creativity and innovation, and they often require more unique engineering solutions. By utilizing modular elements, the overall design can be optimized, making it easier to isolate issues or improve specific parts of the robot without affecting the entire system. .[14]

1.8 Two wheeled Differential drive AMR model definition

A Two-Wheeled Differential Drive AMR is a mobile robotic platform that achieves movement by controlling the rotational speeds of its two main drive wheels independently.

It uses differential wheel speeds to move forward, backward, and rotate (turn in place or follow curved paths). This configuration is commonly used for autonomous tasks such as navigation, mapping, and transportation in indoor environments like warehouses or hospitals.

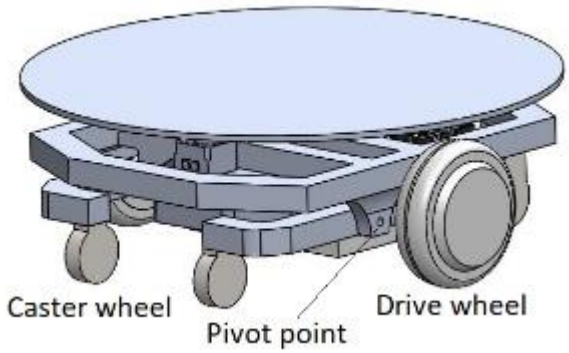


Figure 1.9: First design idea of a differential robot

A mobile robot, or vehicle, has 6 degrees of freedom (DOF) expressed by the pose: (x, y, z, Roll, Pitch, Yaw).

It is composed of two parts: the position = (x, y, z) and the attitude = (Roll, Pitch, Yaw). Informally, Roll can be said to be to the sidewise rotation and pitch the rotation forward or backwards. Yaw, commonly also denoted Heading or Orientation, refers to the direction in which the robot moves in the x-y plane. For a robot on a two-dimensional surface, the 2D pose (x, y, θ), where θ denotes the heading, is sufficient to describe its motion. It is normally defined in a global coordinate system as illustrated below. Note that θ is NOT an angular polar coordinate for the position, instead it points in the forward direction of the robot. .[15]

1.8.1 Representing robot position

The total dimensionality of a robot chassis on the plane is three, two for position in the plane and one for orientation along the vertical axis, which is orthogonal to the plane.

Of course, there are additional degrees of freedom and flexibility due to the wheel axles, wheel steering joints, and wheel castor joints. However, by robot chassis we refer only to the rigid body of the robot, ignoring the joints and degrees of freedom internal to the robot and its wheels.[16]

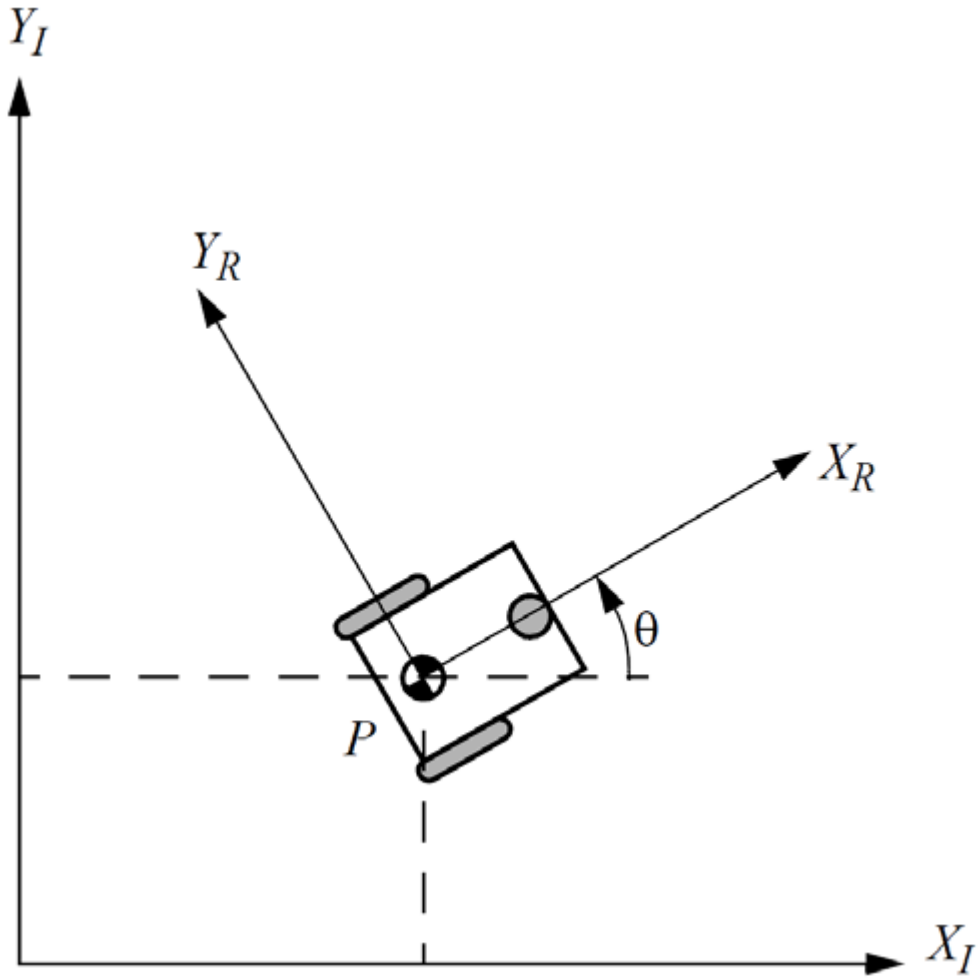


Figure 1.10: The global reference frame and the robot local reference frame.

1.8.2 Differential drive kinematics

It consists of 2 drive wheels mounted on a common axis, and each wheel can independently be driven either forward or backward.

In order to perform rolling motion, while varying the velocity of rotation of each wheel, the robot must rotate about a point that lies along their common left and right wheel axis.

The point that the robot rotates about is known as the ICC -Instantaneous Center of Curvature (see figure 1.11).

The robot's position is expressed considering its center's coordinates (x, y).[17]

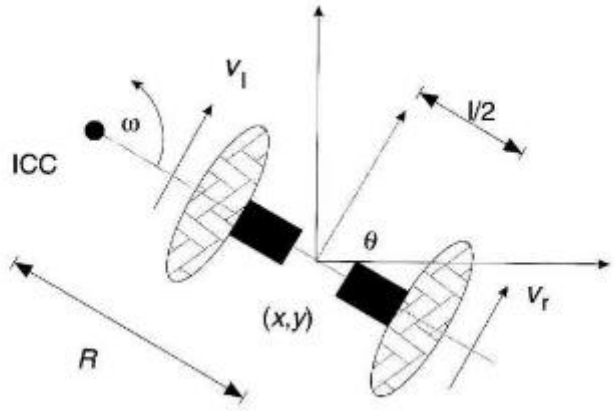


Figure 1.11: Differential Drivers kinematics

By varying the velocities of the two wheels, it is possible to vary the trajectories that the robot takes.

Because the rate of rotation about the ICCmust be the same for both wheels, the following equations can be written

$$w\,(R + l/2) = V$$
$$w\,(R - l/2) = V$$

Where l is the distance between the centers of the two wheels, Vr , Vl are the right and left wheel velocities along the ground (expressed in meter over second), and R is the signed distance from the ICC to the midpoint placed in the middle of the wheels axis. At any instance in time, it is possible to compute R and w (expressed in radiant over second):

$$R = \frac{l}{2}\frac{V_l + V_r}{V_r - V_l} \qquad w = \frac{V_r - V_l}{l}$$

It is possible to consider three different canonical scenarios of the robot motion:

(a) Straight motion: if Vl = Vr, a forward linear motion is achieved in a straight line. R becomes infinite, and there is effectively no rotation, in fact w is zero.

(b) Rotation about the robot center: if Vl = −Vr, then R = 0, and a rotation about the midpoint of the wheel axis is achieved.

(c) Rotation about the left wheel: if Vl = 0, then a rotation about the left wheel is obtained. In this case R = l/2 The rotation about the right wheel can be achieved if Vr = 0.

1.9 Conclusion

Embedded systems form the backbone of modern robotics, providing the computational power, real-time control, and hardware integration necessary for robots to perform tasks ranging from simple repetitive actions to complex autonomous operations.

The integration of specialized hardware (e.g., microcontrollers, sensors, and actuators) and software (e.g., RTOS, firmware, and application software) enables robots to operate efficiently in diverse environments.

The two-wheeled differential drive robot exemplifies how embedded systems facilitate precise control and navigation in mobile robotics. As advancements in embedded system design—such as improved processing capabilities and connectivity—continue to evolve, the potential for robotics to address challenging tasks in industries, healthcare, and daily life will expand significantly. The symbiotic relationship between embedded systems and robotics will remain a cornerstone of technological progress, driving innovation in automation and ntelligent systems..[25]

Chapter II: Sensors for Mobile Robots

2.1 Definition

A sensor is a device that converts a physical parameter or an environmental char- acteristic (e.g., temperature, distance, speed, etc.) into a signal that can be digitally measured and processed to perform specific tasks. Mobile robots need sensors to measure properties of their environment, thus allowing for safe navigation, complex perception and corresponding actions, and effective interactions with other agents that populate it. .[18]

2.2 Overview

Sensors used by mobile robots range from simple tactile sensors, such as bumpers, to complex vision-based sensors such as structured light RGB-D cameras. All of them provide a digital output (e.g., a string, a set of values, a matrix, etc.) that can be processed by the robot's computer. Such output is typically obtained by discretizing one or more analog electrical signals by using an Analog to Digital Converter (ADC) included in the sensor.

In this chapter we present the most common sensors used in mobile robotics, providing an introduction to their taxonomy, basic features, and specifications. The description of the functionalities and the types of applications follows a bottom- up approach: the basic principles and components on which the sensors are based are presented before describing real-world sensors (starting from Sec. 5), which are generally based on multiple technologies and basic devices. A sensor can be categorized as an input type *transducer*. A transducer is a device that converts a signal from one form of energy (e.g., mechanical, thermal, etc.) into another. Transducers can be used to either inject energy in the environment (*emitters*, or *actuators*) or to capture the amount of energy from the environment (*receivers*). A receiver that converts a measurable quantity, such as light or sound, to an electrical signal is called a *sensor*. In mobile robotics, it is common to define as a sensor also an ensemble of transducers and other devices, packaged together, that cooperate for a specific sensing function. For example, ranging sensors based on sound (sonar or ultrasonic sensors) are composed of both a receiver and an actuator synchronized together to measure distances. Following this definition, transducers are building blocks of sensors: a sensor is composed of one or more receivers, zero or more cooperating actuators, and/or other devices (mechanical, optical, etc.) needed to observe the measured phenomena. .[19]

2.3 Sensors Characterization and Specifications

Manufacturers provide the specifications of their sensors as a set of metrics designed to characterize and describe the operating mode and to measure the sensor performances; the most common are listed below.

- **Linearity and Non-linearity: A sensor is defined linear if its response y** to the measured stimulus x is represented by a linear or affine function, e.g., in the one dimensional case $y(x) = \alpha x + \beta$, $\alpha, \beta \in$ R. Linearity plays a major role in interpreting the signal. Highly non-linear sensors are usually more complex to model, and the quantization noise tends to vary with the magnitude of the measurement.

- **Measurement Range and Dynamic Range: The** measurement range [$xmin$,] is represented by the smallest and largest values of the sensed signal that can be measured by the sensor. Stimulus outside such interval cannot be sensed, or they are measured with unacceptable errors, or they can damage the sensor. The ratio between and $xmin$ is called dynamic range, often represented by its base-10 logarithm multiplied by 20 and measured in decibels. .[20]

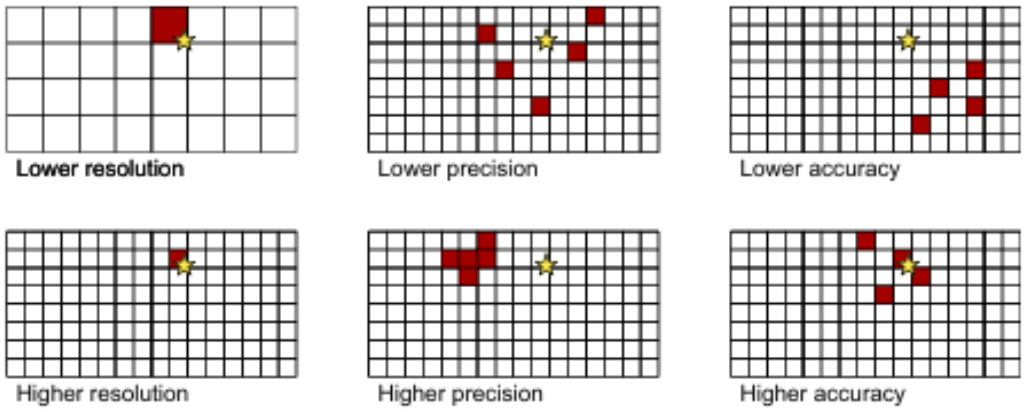


Fig. 2.1 : A visual representation of the resolution, precision, and accuracy parameters for a sensor that provides a two-dimensional (2D) output position, e.g., the position of the robot inside a planar environment. The yellow star is the ideal, perfect measurement (i.e., the ground truth robot position) while the red squares represent a set of output measurements for the steady position represented by the yellow star

- **Resolution:** The resolution is the minimum variation of the measured signal that

can produce a detectable change in the sensor output. For a linear, one-dimensional digital sensor, the resolution can be typically evaluated as the ratio $(xmax-xmin)/\#y$, where $\#y$ is the number of possible discrete output values provided by the sensor. In Fig. 2.1 (left) is reported a representation of two possible resolutions for a sensor that provides a two-dimensional output.

- **Precision:** Precision is a statistical parameter that describes the reproducibility

of the sensor measurements given a steady sensed signal. For such a signal, an ideal sensor with infinite precision should provide the same measured output over time. Real sensors instead provide a range of values over time, statistically distributed with respect to some probability density function (Fig. 1, center). Typically, the precision is evaluated by assuming this density to be Gaussian, so it can be evaluated by computing the variance of a set of sensor readings for a steady sensed signal. .[21]

- **Accuracy:** The accuracy quantifies the correctness of output provided by a sensor compared with the real value of the measured signal (Fig. 1, right). The accuracy can be assessed by taking the difference between the average of the measurements of a steady sensed signal and its true value; an estimate of the true value can be measured for instance by using a sensor with a superior accuracy, or through precise experimental design.
- **Bandwidth:** The sensor bandwidth (typically, represented by a maximum frequency) quantifies how it behaves for inputs that evolve with different frequencies. A sensor with low bandwidth can't properly measure high-frequency changes in the measured input (e.g, vibrations, motions, etc.); on the other hand, the and width is often limited by a low-pass filter to avoid to measure high-frequency noise components.
- **Response Time:** The response time is the duration of the period of time that elapses between a change in the input measured signal and when the sensor output changes accordingly.

2.4 Basic Sensor Components

Sensors that are typically installed on mobile robots are composed of a set of basic devices (receivers and actuators) that directly measure or generate a physical quantity. These

components are often integrated and packaged in a single chip to form the so- called Microelectromechanical Systems (MEMS). In this section, the most common forms of basic receivers and actuators used to build more complex sensors are briefly addressed, classified according to the basic physical quantity that they measure or generate.

- **Force and Deformation**

  Force transducers operate by measuring or imposing a deformation to a body subject to that force. This section focuses only on elementary devices capable to exert/sense elementary linear deformations (Fig. 2.2); electrical motors or dynamos that are complex mechanical compounds are not discussed.

- **Piezoelectric and Piezoresistive** devices measure the force by exploiting, respectively, the voltage or the change in resistance to which a body of a specific material is subject when deformed. Piezoelectric transducers can be used to build both emitters and receivers, while the use of piezoresistive materials is restricted to receivers.
- **Capacitive** devices exploit a condenser whose capacity changes based on the exerted force. Such a capacity can be turned into a voltage by charging the capacitor with a known charge. .[22]

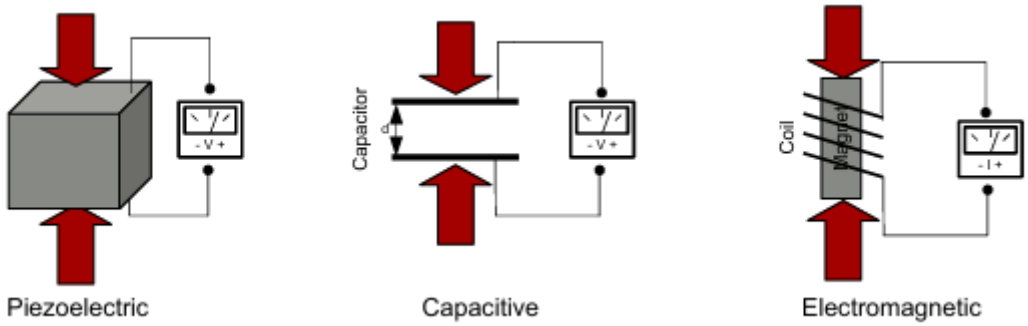


Fig. 2.2 : Overview of basic force transducers.

- **Electromagnetic** devices use a coil of conductive material wrapped around a moving cylindrical magnet that is left free to slide along the coil's axis. Apply- ing a current to the coil induces a magnetic field that in turn moves the magnet. Electromagnetic force transducers can be used both as emitters and as receivers.
- **Light**

Light receivers or emitters are used to construct sensors that measure the amount of light radiation in a certain region of the environment or to directly determine the geometry of the environment by means of distance measures to closest objects. The first task is usually accomplished by assembling arrays of light receivers in a one- or two-dimensional matrix to constitute the sensitive element of a camera. The second task is typically done by either measuring the round-trip time of a beam of light or by processing the return of a known light pattern radiated in the environment by an emitter and sensed by a receiver.

- **Photoresistors** are light receivers whose electrical resistance changes depending on the amount of light radiation to which they are exposed. They can be designed so that they are sensible only to a certain spectrum of the optical wavelengths. An electrical signal is obtained by measuring this resistance of the device. This is typically done by measuring the voltage drop at the end of the photoresistor when mounted in a voltage divider configuration. Typical photoresistors exhibit a latency of around 10 ms. .[23]
- **Photodiodes and CMOS imaging sensors** are receivers consisting of semiconductor devices that can generate a current dependent on the amount of light radiation hitting them. When used in sensors they are typically driven in reverse configuration (i.e., the cathode is driven positive with respect to the anode), where they exhibit a lin- ear relationship between the current and the illuminance within a certain wavelength spectrum. Integrated arrays of photodiodes and amplifiers couples (photosites, or pixels) constitute the typical Complementary Metal-Oxide Semiconductor (CMOS) imaging sensors in consumer cameras.

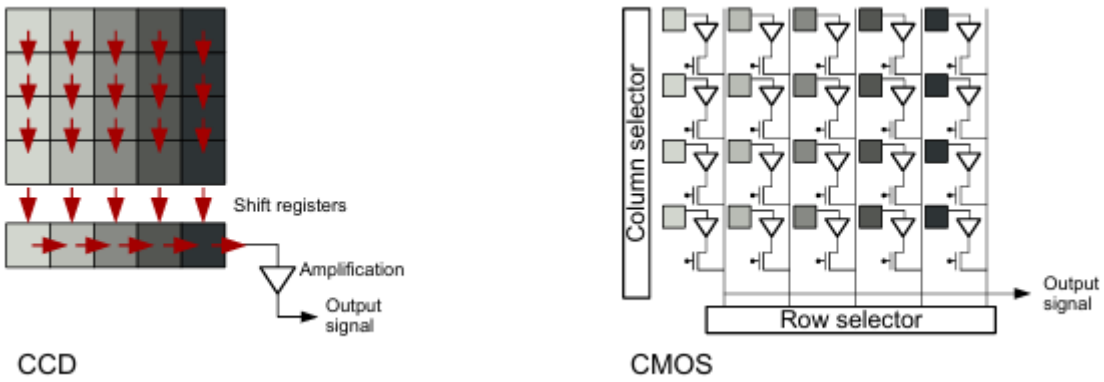


Fig. 2.3 Overview of the popular CCD and CMOS imaging sensors, framing the same image (a gray level linear gradient). In a CCD, the acquired signals are shifted between photosites within the device, amplification is performed one pixel at a time. In CMOS sensors, each photosite includes its own amplifier. In both cases, the output signal is usually converted to digital by an ADC.

- **Charge Coupled Devices (CCD) imaging sensors** are semiconductor light receivers typically used in imaging sensors. A CCD appears electrically as an array of light-sensitive capacitors (as in CMOS sensors, pixels) that accumulate a charge proportional to the amount of light hitting them in a time interval (integration period). The amplification and reading process is performed sequentially, one pixel at a time, thanks to shift registers. Diagrams of the basis structures of both the CCD and CMOS imaging sensors are reported in Fig.2.3.
- **Light Emitting Diodes (LED**) are semiconductor light emitters that produce light radiation when traversed by a current. An individual LED can emit light only in a specific light spectrum, that depends on the doping of its silicon layer.
- **Lyotropic Liquid Crystals** are materials whose molecular configuration changes depending on the current traversing them. A change in the configuration results in a change in the transparency to specific light wavelengths. Usually liquid crystals are assembled in arrays to form light filters used in conjunction with a light emitter to construct projectors or LCD displays.

- **Electromagnetic Field**

Electromagnetic transducers are more commonly known as antennas. They are composed of an array of conductors. When this array is exposed to an electromagnetic field, it produces a current dependent on the electromagnetic field to which it is

exposed. Antennas can be built to have different directionality, and they can be used both as emitters and as receivers. A common use of antennas is in GNSS (Global Navigation Satellite Systems) sensors, where they are used as receivers to sense the signal emitted by the satellites. Radar (see Sec. 10.3) represents another use of antennas where they are used in a highly directional emitter/receiver configuration to determine the range of an object by measuring the round-trip-time of an electromagnetic pulse.

- **Magnetic Field**

Magnetic field in mobile robotics is exploited mostly to sense the direction of the magnetic North. Magnetic field receivers are called magnetometers and can rely on one of the following technologies.

- **Hall Effect** magnetometers consist in a conductor that is traversed by a current. When subject to a magnetic field, the conductor exhibits a voltage in the direction orthogonal to the current traversing it. The intensity of the magnetic field along the direction orthogonal to the flow of current can be measured through this voltage.
- **Magnetoresistive** magnetometers are made of stripes of NiFe magnetic film, whose resistance changes when exposed to a magnetic field. They have a well- defined axis of sensitivity.
- **Converting Signals**

All the receivers mentioned above generate an electric signal. Depending on the type of sensor, one is interested in measuring different characteristics of the signal. The most prominent are the magnitude, the time at which an impulse arrives, or the phase of a periodic signal. .[24]

- The **Magnitude** is converted into a digital number through an analog-to-digital converter (ADC). This process is subject to an unavoidable quantization error that occurs when a continuous quantity is discretized. The time required for conversion and the noise of the resulting measure greatly depends on the type of ADC used.
- The **Time** at which an impulse is received is usually captured by a comparator coupled with a free-running, high-frequency counter. When the impulse is received, the counter is stopped and its value is read. The resolution of these devices depends

on the counter's frequency. The input signal is typically preprocessed by a filtering stage that avoids spurious readings due to noise in the input.

- The **Phase difference** between two periodic signals is the fraction of a cycle that separates the two in phase from the complete overlap. The phase is indirectly used to determine the time difference between the two signals

.2.5 Proximity and Contact Sensors

A proximity sensor is able to sense objects placed in front of it or nearby, at a fixed or parameterizable distance. If the distance is zero, it is called a contact sensor. These sensors are often used in mobile robots for safety functions, for example, to implement emergency stop behaviors in case of unexpected obstacles; they are also used as navigation sensors in simple mobile robots as robot vacuum cleaners.

Fig.2.4: Examples of an Omron E3FA-D diffuse-reflective proximity sensor (left) and a VEX Robotics bumper (right).

- **Photoelectric proximity sensors** (e.g., Fig. 2.4left), also called diffuse-reflective sensors, are typically composed of an emitter/receiver couple embedded in the same case. An emitter transmits a light radiation with a defined wavelength, direction, and beam angle; a receiver placed in the line-of-sight of the emitter (e.g., a *photoresistor* sensitive to the specific wavelength, senses the light eventually reflected back from a nearby object. These devices usually output a voltage corresponding to the detected distance, or a digital output that changes if the distance drops below a fixed distance threshold.
- **Bumpers** (Fig. 2.4 right) are simple contact sensors that detect a physical collision by means of the pressure or release of a microswitch attached to a protective case designed to receive shocks. .[25]

2.6 Encoders

An encoder is a proprioceptive device that converts a linear position (*linear encoders*) or an angular position (*rotary encoders*) into a digital code. The latter type is widely used in mobile robotics, to measure the angular position of the robot wheels. From this information, knowing the wheels' diameter and track, it is possible to derive information such as the linear and rotational velocities of the robot, and consequently an estimate of its relative motions in a planar environment. The most important type of rotary encoders are:

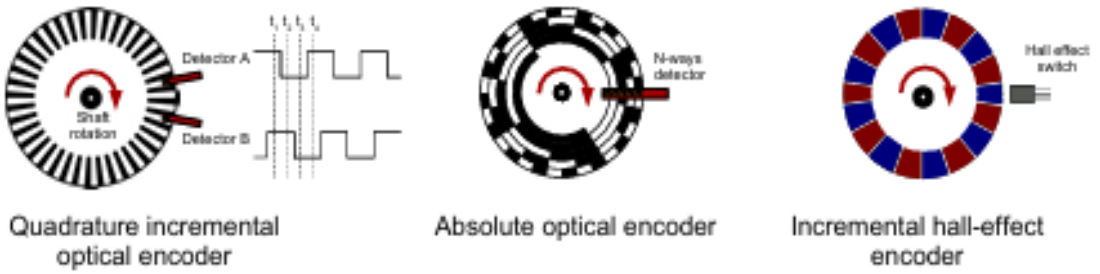


Fig. 2.5 Overview of three types of encoders.

- **Optical encoders**: in their simpler implementation, they are composed of two opposite transducers: a light emitter (e.g., a LED) and a photoreceptor. Between the transducers pair, it is placed a disk (e.g., Fig.2.5 left), fixed to the shaft, with a sequence of transparent (white in Fig.2.5 and opaque areas. Depending on the angular rotation of the shaft, the photoreceptor can detect or not the light diffused by the LED, providing in output a square wave. Incremental encoders just provide information about the motion of the shaft, by counting the periods (*tics*) of such wave. The quadrature incremental encoder improves this design by employing a second emitter-detector pair, shifted in phase of 90 degrees with respect to the first square wave. Combining the two output waves, it is possible to increase the reso- lution by 4 times while detecting the direction of rotation. Absolute encoders (e.g., Fig.2.5 center) also provide the absolute position of the shaft by employing multi- ple concentric discs and detectors: for each tic, the disc provides a unique binary code.
- **Hall-effect encoders**: they are composed of a sequence of north-south magnetic poles, equally spaced and arranged in a circle (e.g., Fig.2.5 right). A Hall effect magnetometer placed near the disc is used to measure the magnetic field. Often, the

magnetometer is combined with a threshold detector, to provide in output a square wave: in this case, the operating principle is very similar to the one presented for the optical encoders.

2.7 Inertial Measurement Units (IMUs)

IMUs are composed of a tri-axial cluster of accelerometers and a tri-axial cluster of gyroscopes, usually based on MEMS technology. The two triads define a single, shared, orthogonal 3D frame, and they are often associated with a magnetometer (Fig. 2.6, left). An AHRS (Attitude and Heading Reference System, e.g., Fig.2.6, right) IMU provides 3D orientation by internally integrating the gyroscopes and fusing this data with the accelerometer and the magnetometer data. IMUs are often used in mobile robotics to integrate and to improve the consistency and the reactivity of the robot navigations systems, e.g., in GNSS-based navigation systems.

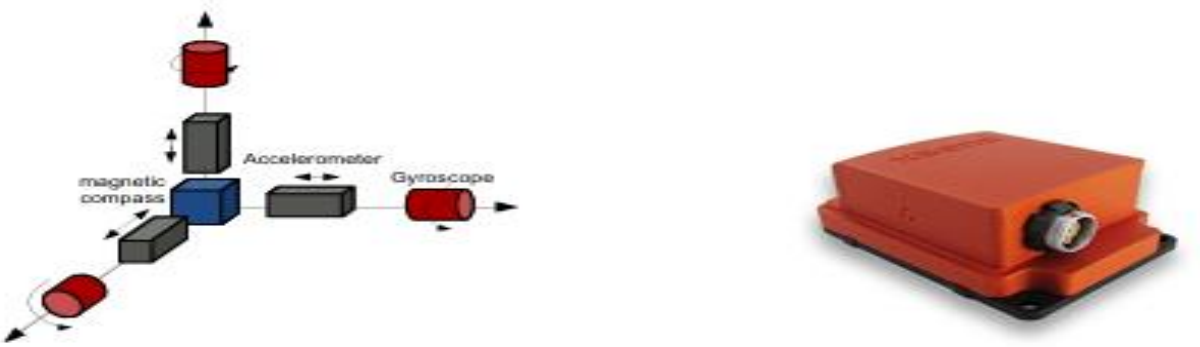


Fig. 2.6: Conceptual arrangement of sensors within an IMU (left); the XSens MTi-300 AHRS IMU (right).

2.8 Digital Cameras

Fig.2.7: Two examples of digital, area scan cameras. A Basler Dart without lens, with visible CMOS imaging sensor (left): an IDS uEye with mounted lens.

Digital cameras (also called *area scan cameras*) are devices able to produce two-dimensional (2D) arrays (called *images*) of measurements of a specific electro- magnetic radiation (e.g., the visible light) coming from non-occluded surfaces of a framed three-dimensional (3D) scene. In this 3D-2D projection, one dimension (i.e., the points' depth) is anyhow lost.

The basic components of a digital camera are (a) an imaging sensor, e.g., a CCD or a CMOS (see Sec. 2.5. e.g., Fig.2.7, left); (b) an optical system, called *lens*, used to route the sensed information (e.g., the visible light) from each 3D point toward a 2D point of the imaging sensor (e.g., Fig.2.7 right); (c) an internal ADC used to convert each pixel measurement into a digital value.

In nature, each 3D point constantly emits radiations that are spread in all directions in the space, possibly reaching the whole area of an imaging sensor exposed to such

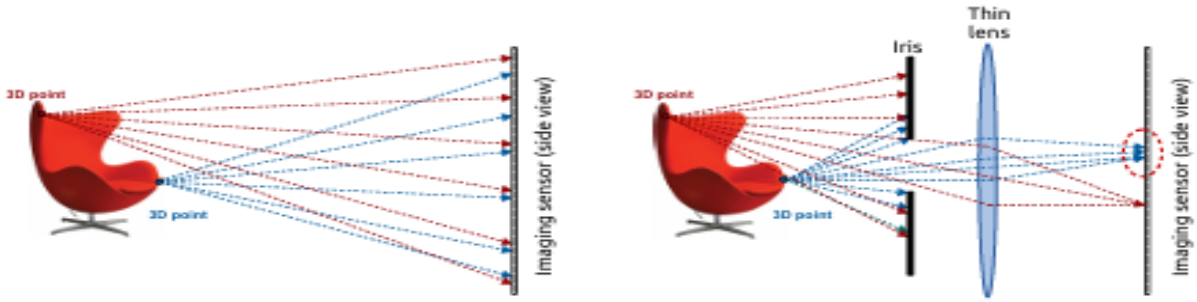


Fig. 2.8: Without a lens, each 3D point of the scene will project its radiation into each point of the 2D imaging sensor, producing a useless image (left); A lens in front of the sensor (right), here represented by its simplest model, the *Thin lens model*, is used to mitigate as possible this phenomenon.

radiations without a lens (see Fig. 2.8, left). In the ideal case, for each 3D point emitting such omnidirectional radiations, there should be exactly one 2D point of the imaging sensor that receives a single ray of this and only this radiation. This ideal model is called Pinhole model. The aim of the lens is to approximate the ideal case by routing as possible multiple radiation rays coming from the same 3D point toward a single 2D point of the imaging sensor (see Fig. 2.8, right). This is also obtained by introducing a barrier with a central opening (called Iris) to block most of the rays. Although this model (called Thin lens model) only approximates the ideal one (e.g., the lower 3D point of the right side of Fig. 2.8 is projected in a neighborhood of points into the 2D imaging sensor), this model and especially the Pinhole model are commonly used to describe and model real cameras. Cameras that fall into such model are also called perspective cameras.

used in a very wide range of applications in mobile robotics, among others: place recognition, visual servoing, ego-motion estimation, SLAM (Simultaneous Localization and Mapping), object detection and classification, etc..[26]

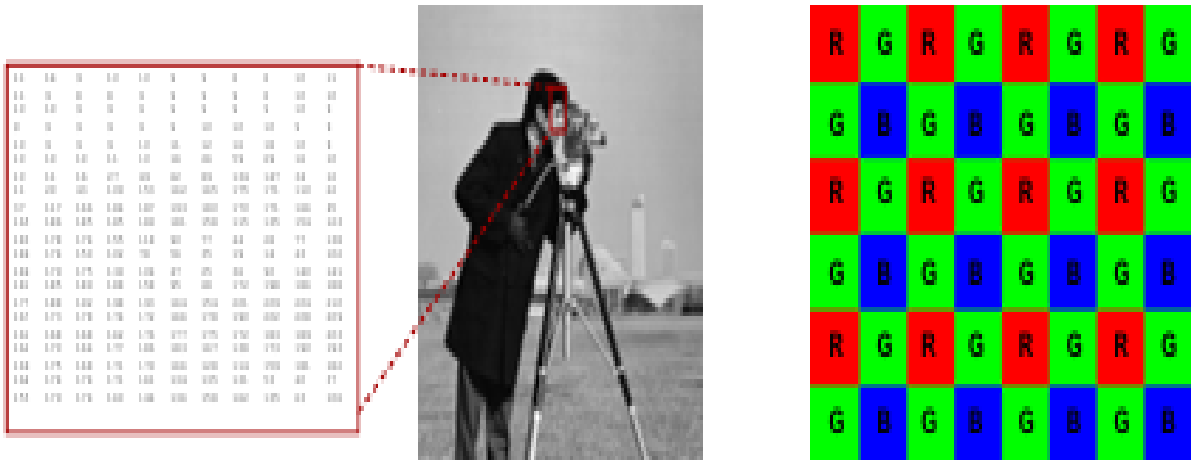


Fig. 2.9: A portion of a single channel, gray level *image* and the corresponding sub-picture (left); Bayer arrangement of an RGB imaging sensor (right).

measuring the intensity of the electromagnetic radiations that lie within the visible light spectrum, that corresponds to a range of wavelengths from about 380 to about 740 nanometers. CMOS or CCD imaging sensors used in such cameras are designed to be sensible only to this spectrum. The provided output is a brightness image: low radiance 3D points will be mapped into small pixel values, saturating toward the zero value; conversely, high radiance points will be mapped into high pixel values, saturating toward the largest integer representation of the digital sensor. To be easily interpreted by the human eye, an image can be represented by a picture, that maps each pixel value into a visible" color ", in this case a gray level (e.g., Fig. 2.9, left).

. Color cameras extend the functionality of GL cameras by providing color vision capabilities, i.e. by measuring also the wavelength of the electromagnetic radiations spread by 3D points. Color cameras are often called RGB cameras since they usually employ the RGB additive color model, that represents the color information by means of three-channel images. Color cameras imaging sensors include three types of photoreceptors; each type is sensible to a specific wavelength spectrum centered around one of the three primary colors of the RGB model: red, green, and blue. For each pixel, such cameras provide a triad of values

corresponding to the intensity of these three colors, so reproducing a wide range of colors. An RGB camera can be assembled by employing three separate imaging sensors, each one sensitive to a specific wavelength spectrum, and by a prism that splits each light beam coming into the lens into three beams that are projected in exactly the same 2D locations of each of the three sensors. A simpler and more popular implementation employs a single imaging sensor in which each pixel is sensitive to a specific wavelength spectrum. The distribution of the pixels follows a fixed pattern, e.g. the Bayer arrangement depicted in Fig. 11 (right). In this case, each pixel provides a single value, corresponding to a specific spectrum. A special interpolation algorithm is then used to recover the R, G, and B values for each pixel.

Digital cameras can be used also to detect radiations that are not visible to humans, such as near-infrared (NIR) radiations. They are basically GL cameras sensitive to wavelengths that spread from 750 to 1400 nanometers.

2.9 Ranging Sensors

Ranging sensors provide the distance to objects on a defined portion of the surrounding scene. They can be considered an extension of the proximity sensors (Sec. 5) since, compared to the latter, ranging sensors directly output metric distances to objects, and provide extended field of view or extended measurement range. Ranging sensors applied to mobile robots can be used in several applications, among others robot localization, SLAM, 3D environment reconstruction, object localization, and obstacle avoidance.

Such sensors output arrays of distances (called ranges), each one associated with a defined direction of measurement, or equivalently the projection of such distances (called depths) along a fixed direction defined by one of the axes of the sensor coordinate system. A 2D array of ranges is often called range image, while a 2D array of depths is called depth map. Cameras that provide a depth map are generally called depth cameras. From both ranges and depth information, it is possible to generate a point cloud (e.g., Fig. 2.17, right) that is a vector of 3D points that represents samples of the 3D structure of the surrounding scene. In the following, we briefly present the most common ranging sensors.

2.10 Sonar and Ultrasonic Sensors

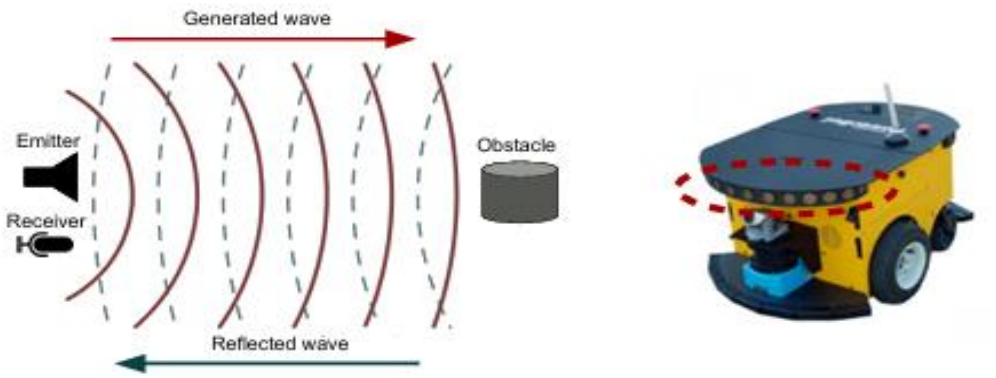


Fig. 2.10: Operating principle of sonars (left); The red dashed ellipse highlight the array of sonars that equip the MobileRobots PowerBot platform (right)

Sonar (acronym for Sound Navigation and Ranging) are devices that use sound waves to detect objects and measure distances. Passive sonar devices only perceive sounds, while active sonars also emit pulses of sound: the latter are commonly used in mobile robots for obstacle avoidance and self-localization. Active sonars consist of an emitter/receiver pair measuring the distance to an obstacle from the round-trip

time of an emitted pulse (Fig. 2.10, left). The pressure wave (sound) that is emitted can have different frequencies ranging from very low to high (ultrasound). The latter are the ones commonly used in sensors for robotics, which are usually called ultrasonic sensors. The cone of the emitted signal can be varied: the narrower the cone, the higher the angular resolution of the sensor. In order to have multiple readings at different directions, multiple sonar devices need to be configured in an array (e.g., Fig.2.10, right). Transducers used to build sonars lie in the family of the force and deformation transducers.[27]

. Piezoelectric Sound Transducers rely on materials that generate a voltage when their shape is changed. This change in shape occurs as a consequence of sound waves. They are used both for detecting sound in the audible and in the ultrasound frequency bands. Piezoelectric transducers can also be used to generate a sound, since piezo- electric materials change shape when subject to a voltage. .[28]

. Capacitive Microphones are condensers that change their capacity when exposed to a sound wave. The charge in the capacitor results in a voltage that depends on the varying

capacity and is thus related to the pressure exerted by the sound waves on the transducer's membrane. .[29]

2.11 LiDAR

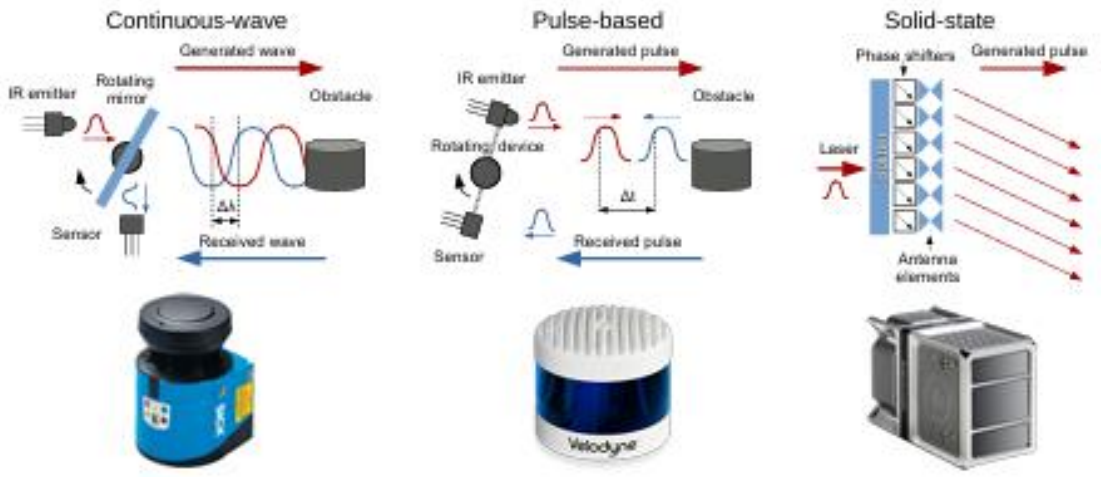


Fig 2. 11: Operating principles (top row) of three types of LiDAR (bottom row): continuous-wave LiDAR with rotating mirror for the SICK LMS 100 (left column); pulse-based LiDAR with spinning laser/detector pairs for the Velodyne Alpha Puck LiDAR that exploits an array of 128 pairs (central column); solid-state LiDAR that uses an optical phased array to deflect laser beams for the Quanergy S3-2 (right column).

LiDAR (acronym for Light Detection and Ranging) devices exploited in mobile robots use coherent light sources (i.e., lasers, typically at infrared wavelengths) to measure the distance and reflective properties of the environment. .[30]

There are two main LiDAR technologies, with different working principles:

. Continuous-wave (CW) LiDARs (also known as phase-shift-based LiDARs) de- tect the range by measuring the phase difference $\Delta\lambda$ between a generated continuous wave and the reflected wave backscattered by an object encountered along the sensor line of sight (see Fig. 2.11, left);

. Pulse-based (PB) LiDARs measure directly the time-of-flight, i.e., the round-trip- time of a pulsed light (see Fig.2.11, center). This type needs very short laser pulses and high temporal accuracy, considering that the speed of light is 0.3 meters/nanosecond.

Standard point-based LiDARs can only measure one point at a time per emitting diode. However, since the time for a single range measurement is rather short, a sequence of

measurements can be rapidly acquired by deflecting the beam. This is done either by employing a deflecting mirror (e.g., as in Fig. 2.11, left) or by mounting the transmitting/receiving element on a movable support that rotates around one or two axes in order to acquire 2D planar scans or 3D range images (e.g., as in Fig. 2.11, center). To acquire more data in parallel, multi-channel LiDARs use several diodes pointing in different directions, usually arranged to acquire a planar profile in one shot. By rotating or translating the diodes array, it is possible to generate a range image. Current state-of-the-art systems offer up to 128 pulse-based diodes that can generate 360-degree range images.

. Solid-state LiDARs Recently, solid-state LiDARs are entering the market of sensors for mobile robots. These sensors allow to obtain a dense 3D scan of the surrounding environment without using mechanical tools for deflecting the laser beams. Solid-state LiDARs typically use a pulsed laser that feeds an Optical Phased Array (OPA) that is a 2D array of closely spaced (around 1 µm) optical antennas. Variable phase control is applied at each antenna to generate a radiation pattern that points in the desired direction (see Fig. 2.11, right).

Along with sonar arrays, LiDARs are the most commonly used sensors in mobile robot navigation and obstacle avoidance. Compared to sonars, LiDARs have several advantages: (i) increased range; (ii) higher frequency; (iii) increased accuracy. The disadvantages of a LiDAR compared to a sonar are higher cost and greater sensitivity to fog and dust, although sensors that report multiple return echoes are nowadays more common, resulting in greater robustness to adverse events. .[41]

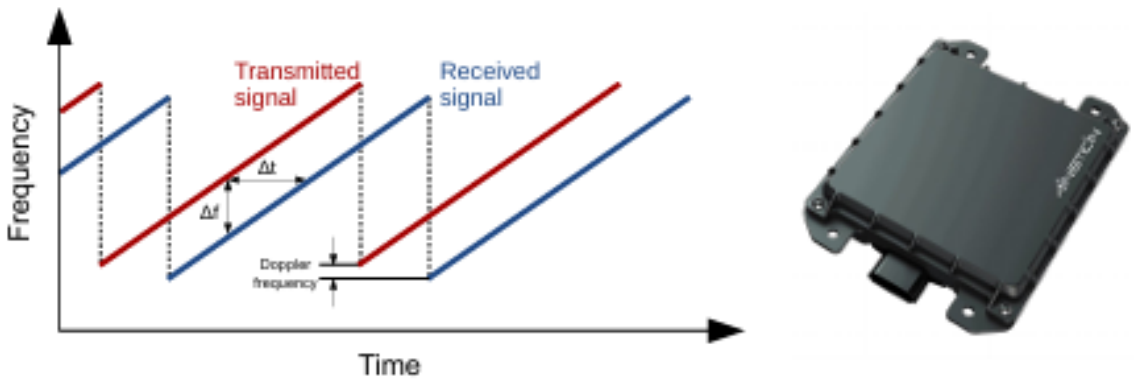


Fig. 2.12 Operating principle of Frequency-Modulated Continuous-Wave (FMCW) radars (left); The Ainstein O-79 Imaging Radar (right).

2.12 Radar

Similar to LiDARs, radar devices (acronym for Radio Detection and Ranging) emit an electromagnetic wave, but uses radio waves (wavelengths typically between 3 mm and 30 cm) instead of laser light. The signal is transmitted and received by two distinct antennas or antenna arrays when required to estimate the angular positions of objects. Compared to LiDARs, radars have lower angular and range resolution/accuracy but this disadvantage is compensated by the possibility of using the Doppler effect to compute the relative velocity between the sensor and the detected objects. Another key benefit of radars is their robustness to harsh environmental conditions such as snow, fog, rain, dust etc. Radars typically used in automotive and robotics are based on FMCW (Frequency-Modulated Continuous-Wave) transceivers, have a frequency range spanning from 20 to 80 GHz, and can detect the position, relative speed, and direction of motion of objects up to 200 meters. FMCW radars emit a signal called chirp where the frequency varies linearly over time. The difference between the frequency of the signal sent and that received is linearly related to the distance from the radar of the object that generated the reflected signal (see Fig. 2.12, left). Nowadays, Cascade/Imaging Radar (CIR) systems are becoming very popular (e.g., Fig. 2.12, right).: in CIRs, multiple FMCW transceivers are cascaded together, to increase operational safety and achieve high angular resolution, thus enabling to generate a dense 3D point cloud mapping of the surrounding environment.

2.13 Conclusion

Sensors are critical components for mobile robots, enabling them to perceive, navigate, and interact with their environments effectively. The chapter outlines a comprehensive taxonomy of sensors, ranging from simple contact and proximity sensors to advanced ranging and vision-based systems like LiDAR, radar, and RGB-D cameras. These sensors are classified by their excitation signal (passive vs. active), measurement domain (proprioceptive vs. exteroceptive), and measurement type, with performance characterized by metrics such as linearity, resolution, precision, accuracy, bandwidth, and response time.

Basic sensor components, including force, light, electromagnetic, and magnetic transducers, form the building blocks of complex systems like MEMS-based IMUs, digital cameras, and ranging sensors. Applications in mobile robotics are diverse, with wheel encoders providing odometry, LiDARs and ultrasonic arrays supporting navigation and obstacle avoidance, and cameras enabling high-level tasks like object detection, SLAM, and human-robot interaction. In outdoor settings, GPS and robust radar systems enhance localization and environmental mapping, while indoor environments leverage UWB and AHRS IMUs for precise positioning.

The choice of sensors depends on the robot’s application, environment, and operational requirements, with multi-sensor integration often employed to ensure redundancy, safety, and enhanced perception for complex tasks.

Chapter III: ROS2 Basics, Localization, and Navigation

3.1 Introduction to ROS2 Framework:

Robot Operating System (ROS) is a set of open-source algorithms, hardware driver software and tools developed to develop robot control software. Even though it has operating system in its name it is not an operating system. It is

• Communication System (Publish Subscribe and Remote Method Invocation),

• Framework & Tools (Build system & dependency management, Visualization, Record and Replay)

• Ecosystem (Language bindings, Drivers, libraries and simulation (Gazebo)).

The first version of ROS was mostly used in academic projects. ROS 2 was developed so that it can be used in commercial projects. The biggest change that came with ROS2 was the selection of the DDS middleware for the communication layer. The DDS middleware, which has proven its worth in defense industry projects, has strengthened the communication between robot components with its decentralized publish subscribe architecture. Currently, ROS2 has been adapted 3–4 different DDS products. Since ROS2 is open source, related companies have also offered DDS libraries as open source to the sector. At the same time, ROS 2 has found a wider area of use with multi-robot communication, real-time communication and different platform support. Currently, ROS 2 is used in different areas from humanoid robots to industrial robots and autonomous vehicles.

ROS includes mature open-source libraries to be used for navigation, control, motion planning, vision and simulation purposes. The 3D visualization tool called RVIS is an important tool used with ROS. Similarly, the simulation tool called Gazebo is seen as a usefull tool for robot developers. Apart from this, Open CV library is a library used for detection purposes in ROS 2. In addition, we see that the QT graphic library is also used for the user interface in ROS 2 projects and is an add-on available for this purpose. Figure 3.1. [31]

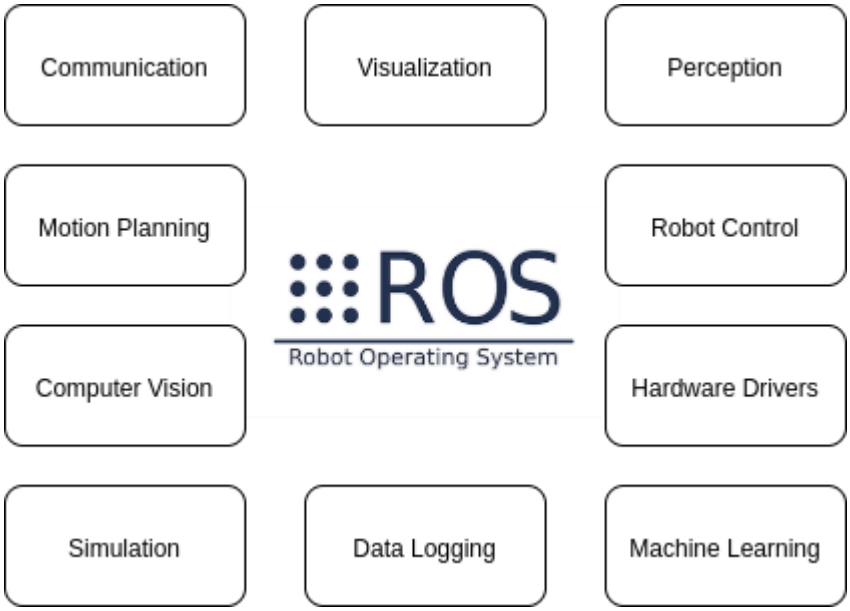


Figure 3.1: ROS 2 content (picture from Ros2 wiki)

In this article, I will not mention the capabilities of the existing libraries and tools in ROS 2, how to set up ROS 2, and the details of how to develop code with ROS 2. There are resources and trainings that explain these issues. I will try to explain to you the ROS 2 architecture, which I have not seen clearly explained

3.2 ROS 2 Architecture

The ROS 2 has distributed real-time system architecture. Sensors in robots, motion controllers, detection algorithms, artificial intelligence algorithms, navigation algorithms, etc are all components (called as node) of this distributed architecture. The DDS middleware selected with ROS 2 for data exchange enables these components to communicate with each other in a distributed environment. Figure 3.2

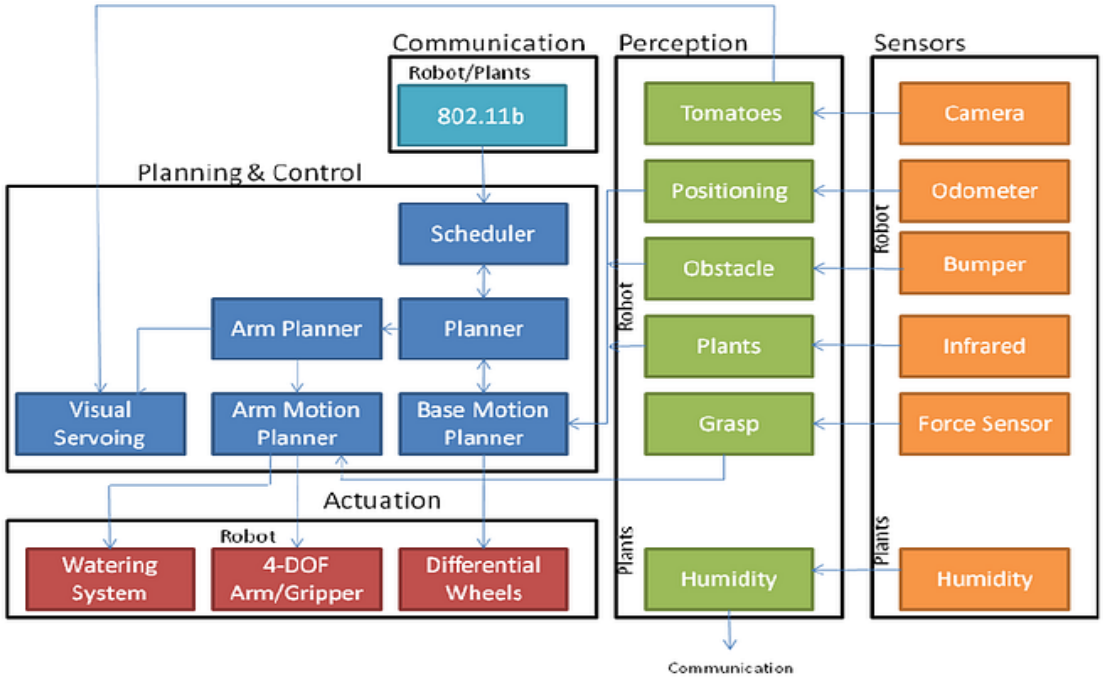


Figure 3.2: Architecture of a ROS 2-Based Robotic System for Plant Interaction

3.3 Components of robot control software

ROS 2 has provided its own abstraction layer (rmw) on top of DDS instead of directly using the DDS middleware. Thus, the details of the DDS middleware interface are abstracted from the user. Apart from that, rmw support is available in different DDS products. The user can select and use the DDS library they want, and even thanks to the network level interoperability, it is possible to use more than one DDS library in the same project. Figure 3.3. [32]

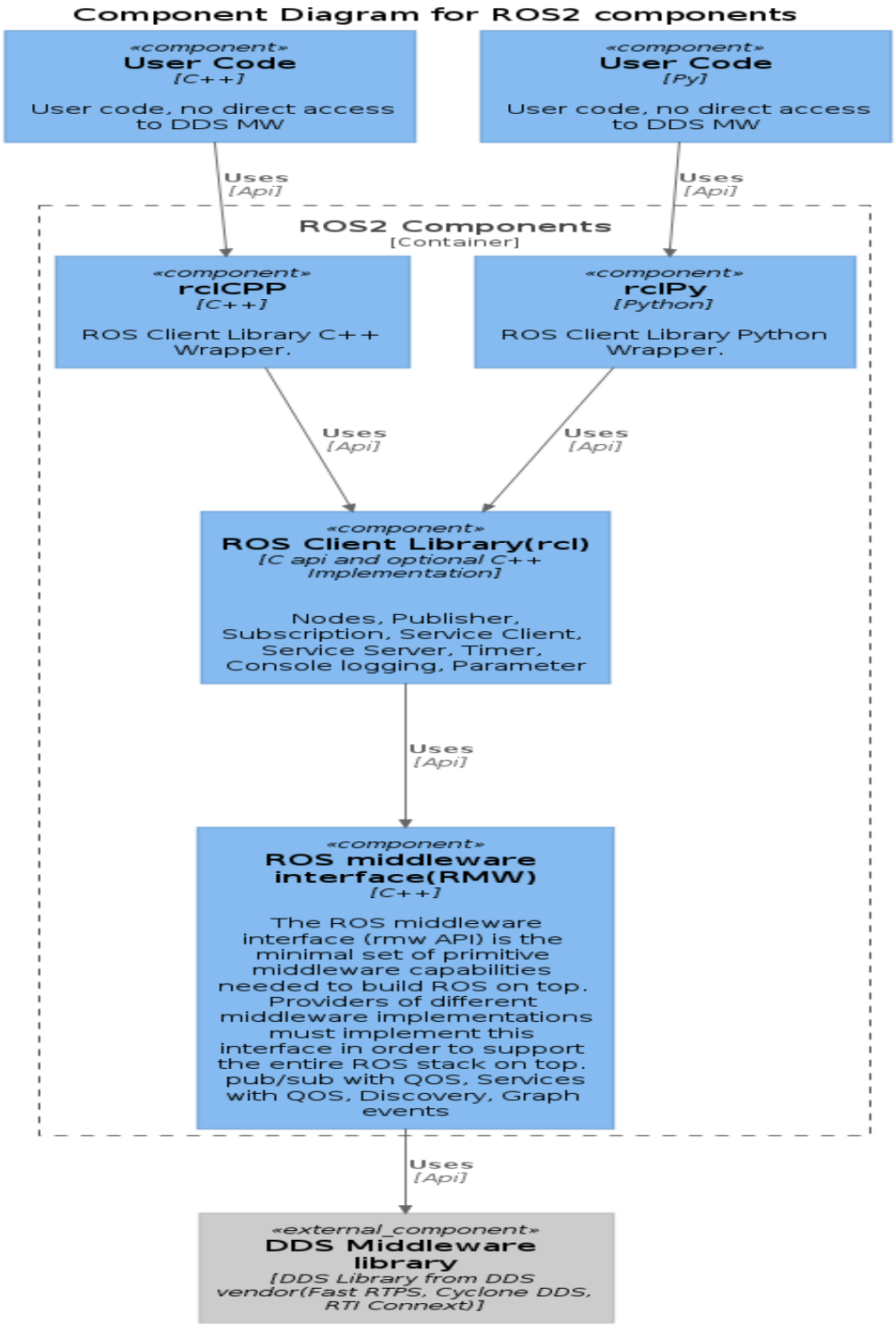


Figure 3.3: component diagram for ROS2 components

3.4 ROS 2 Client Library

ROS 2 applications access ROS 2 features through the ROS Client Library (RCL) client library. The Rcl library is written in C language and on it there are Rclcpp client libraries for C ++ language and Rclpy for Python language. There are independently written ROS 2 client libraries in other languages such as Java, Go. The client library is primarily provided with the standard interface required to exchange data with Topic and Service approaches over ROS 2. In addition, the ROS2 library has capabilities to provide operating system abstraction and ready-made micro-architectural structures.

3.5 ROS 2 Graph Structure

The node, topic, message structure, and discovery form the basic distributed architecture of ROS 2. This structure is called a graph in ROS 2 terminology. Let's get into the details of these architectural basic structures.

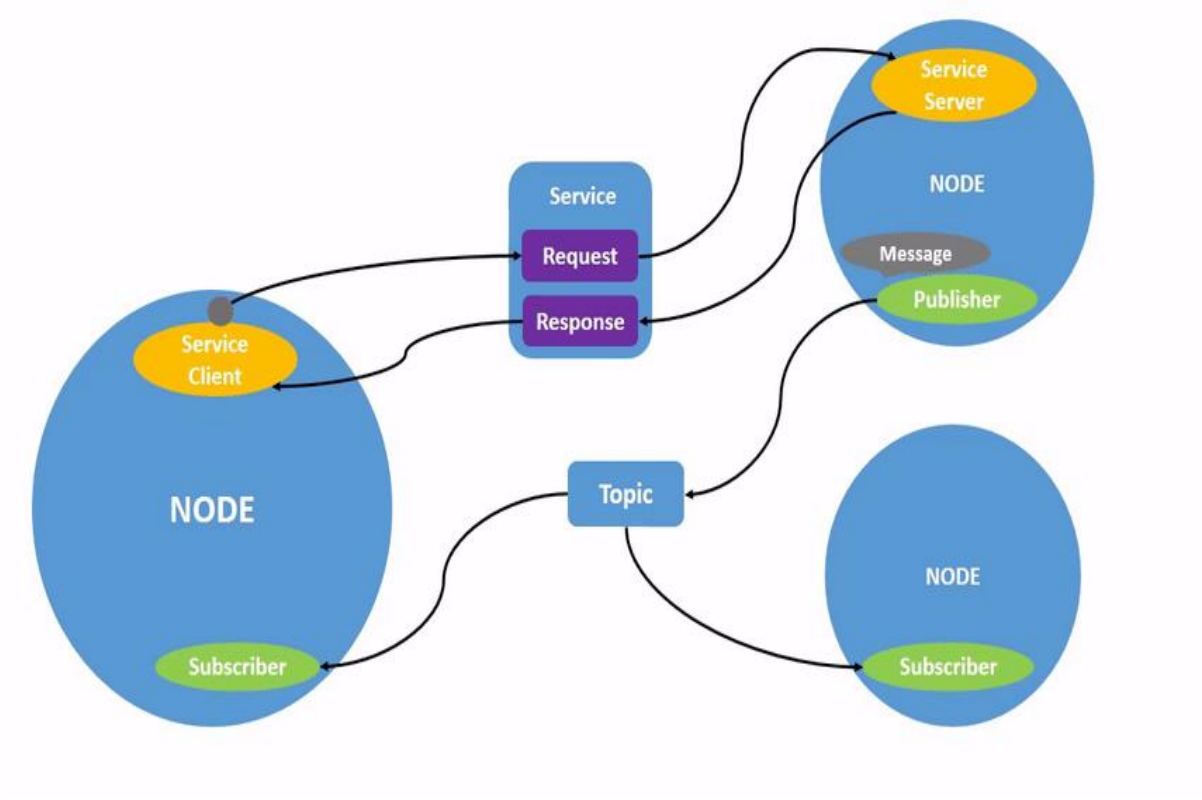


Figure 3.4: Ros 2 Graph (picture from Ros2 wiki)

3.6 ROS 2 Core Concepts:

ROS 2 (Robot Operating System 2) is a framework for building robot applications. It provides a set of tools and libraries that simplify communication, data exchange, and task management between different software components within a robot system. Here's a breakdown of the key concepts that make ROS 2 function:

3.6.1 Nodes:

• Individual Processes: As you mentioned, nodes are independent software processes that handle specific tasks within a ROS 2 system. They can represent various functionalities like sensor data processing, motor control algorithms, or high-level decision making for a robot.

• Decoupled from OS Processes: ROS 2 separates the node concept from the underlying operating system process structure. This allows for more flexibility in creating and managing nodes. You can even have multiple nodes within a single process if needed.

• Distributed or Centralized: Nodes can be deployed on a single computer for simpler setups or distributed across multiple computers for complex systems requiring more processing power or geographically separate functionalities.

3.6.2 Topics:

• Communication Channels: Topics act as named communication channels that facilitate message exchange between nodes. They are like named pipes or message queues where nodes publish (send) and subscribe (receive) messages.

• Asynchronous Communication: Nodes subscribing to a topic receive and process messages asynchronously. This means the publisher doesn't wait for the subscriber to receive the message before continuing.

3.6.3 Messages:

• Data Structures: Messages are the data structures that carry information between nodes. ROS 2 offers a variety of predefined message types for common data formats like sensor readings, control commands, and status updates. You can also define custom message types for specific needs.

• Structured Communication: Messages provide a structured way to exchange information, ensuring all nodes understand the format and meaning of the data being transmitted.

3.6.4 Services:

• Synchronous Communication: Services enable synchronous communication between nodes. Unlike topics (asynchronous), services allow one node to request a specific action or information from another node and wait for a response.

• Request-Response: Services use request-response message pairs. A node sends a request message specifying the desired action or information. The service node processes the request and sends back a response message containing the result.

3.6.5 Parameter Server:

• Centralized Configuration: The parameter server acts as a central repository for storing and managing configuration parameters and settings used by multiple nodes within a ROS 2 system.

• Dynamic Reconfiguration: This server provides a way to dynamically change robot configurations at runtime without restarting nodes. Nodes can access and update parameters stored on the server, allowing for flexible system adjustments.

This core concepts work together to create a modular and scalable robot software architecture. Nodes act as independent units performing specific tasks, communicating with each other through topics and services using structured messages. The parameter server provides a centralized way to manage robot configuration, enabling flexible adjustments. By leveraging these concepts, ROS 2 simplifies the development, deployment, and maintenance of complex robot applications. .[33]

3.7 Communication Patterns (Topic, Service, Action)

Nodes communicate with each other via Topic, Service Invocation and Actions. The topic-based communication has published subscribe architecture where the data produced and published by a node is subscribed by more than one node and similarly, one topic data can be produced to more than one node. Topics defined in ROS 2 exactly match DDS topics. Service calls use a remote method invocation approach from an application developer perspective. Actions are an approach used for long service calls. The client requests an action with goal,

the action server starts the long-running process, publishes intermediate results, and reports the final result when the goal is achieved. These service and actions are also implemented using topics over the DDS middleware. .[34]

Decentralized architecture of Topic structure creates advantage in terms of both performance and fault tolerance. In addition, rich quality of service features provided by the DDS middleware have also been a factor that raised the ROS 2 architecture to higher levels.

3.8 Messages (Message)

The data types used in topic communication are called message (Message). These Message data structures basically match the DDS data types (Type). A sample camera message is given below:

```
sensor_msgs/CameraInfo
Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
RegionOfInterest roi
  uint32 x_offset
  uint32 y_offset
  uint32 height
  uint32 width
float64[5] D
float64[9] K
float64[9] R
float64[12] P
```

The messages defined in ROS 2 are actually of great importance for interoperability. With the help of these standard messages, cameras developed by different companies can be quickly integrated into the system and can communicate with standard messages regardless of the

model camera. In this case, the Node component developed for the relevant camera will convert the camera's special communication protocol messages into standard DDS data. Preliminary studies can be performed using the camera simulator in the Gebazo simulation.

ROS 2 also effectively uses the quality of services (QOS) defined in the DDS middleware. For example, while continuously updated sensor data is sent in best effort, a command sent to the robot to do a job can be sent reliably. Matching the rich quality of service features in DDS with the topics is an important design decision. Ready-made service quality profiles that can be used according to the type of data are defined and the service quality related to newly defined topics is determined by selecting one of these profiles.

The QOS profiles defined in ROS 2 are known by service, sensor data, parameters and default. For example, the parameters use the reliable service quality value while the sensor data uses the best effort QOS value. [35]

3.9 Discovery of applications in a distributed environment

In ROS1, the discovery mechanism and message transmission were communicating through a service called Rescore. DDS middleware does not need a broker to distribute data. In addition, applications in the DDS middleware dynamically discover each other through special DDS topics. As a result, DDS has switched ROS 2 to a decentralized architecture by meeting the discovery, distribution and data encoding needs of ROS 2. So, in the new architecture, the Rescore service has been retired. [36]

3.10 ROS2 Device adaptation

We mentioned that devices such as sensors and handlers in robot systems are integrated into the ROS 2 system as a node. The code required to be written for the integration of these devices is called the Adapter (wrapper) in ROS terminology. This approach shows a similar structure to the approach we have seen in open architecture systems. For example, the adapter software to be developed for a motor controller will publish the data as topic (current speed and motor status) independently from the relevant motor controller interface and transmit the topic data to the motor (Speed command). If requested, a service interface can be provided for camera control (engine shutdown). This adapter software will be able to get the parameters of the motor from the parameter server and start the motor with these parameters (maximum speed).

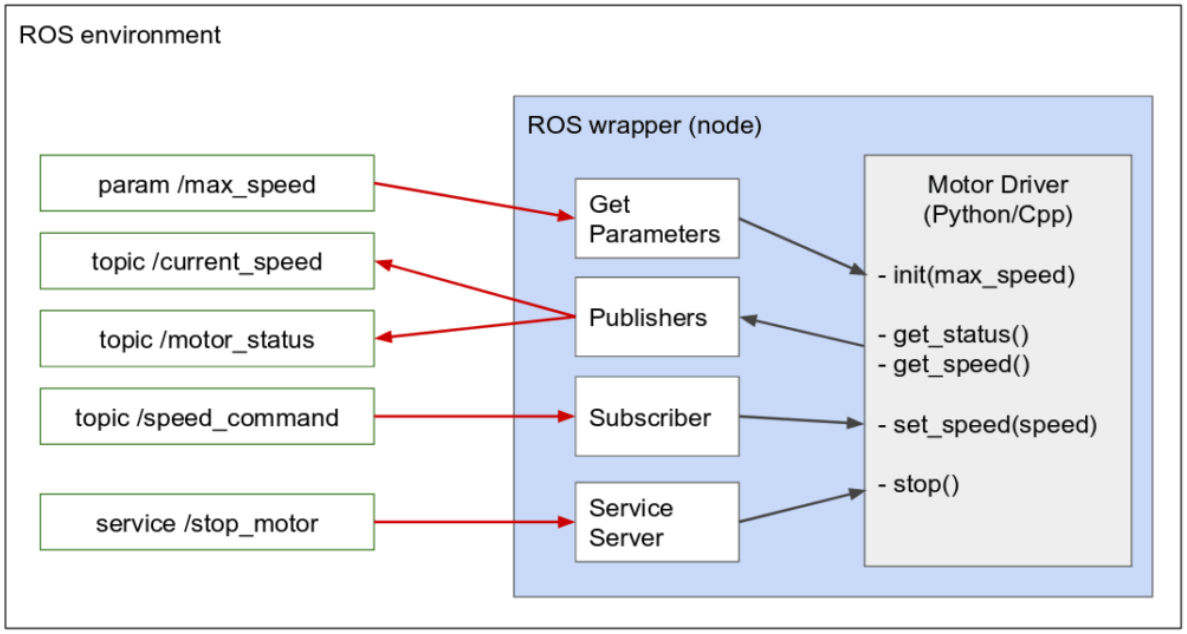


Figure 3.5: An engine adapter node architecture (picture from roboticsbackend.com)

3.10.1 Parameter Server

One of the common services that can be considered as an architectural unit in ROS 2 is the parameter server. In this way, all nodes can store and retrieve the parameters they want to keep.

Recording and Playback

ROS 2 has ROS2 Bag service for recording and replaying DDS data. Recording and replay can be used effectively in such distributed real-time systems for post-mission behavior analysis and testing purposes. .[37]

3.11 Localization Techniques in ROS2:

Localization in ROS2 involves estimating the robot's pose (position and orientation) relative to a known map or environment. Common localization techniques include:

• Odometry: Utilizing sensor data such as wheel encoders or inertial measurement units (IMUs) to estimate the robot's movement and update its pose over tme.

• SLAM (Simultaneous Localization and Mapping): Simultaneously building a map of the environment while localizing the robot within that map. ROS2 provides packages such as GMapping and Cartographer for SLAM. .[38]

• Particle Filters: Probabilistic techniques that maintain a distribution of possible robot poses based on sensor measurements and motion models. Packages like AMCL (Adaptive Monte Carlo Localization) in ROS2 implement particle filter-based localization.

3.12 Navigation Systems and Algorithms in ROS2:

ROS2 provides comprehensive navigation capabilities for autonomous robot movement in dynamic environments. Key components include:

• Path Planning: Algorithms such as A* (A-star), Dijkstra, and RRT (Rapidly-exploring Random Tree) are used to generate feasible paths from the robot's current pose to a desired goal pose while avoiding obstacles.

• Obstacle Avoidance: Techniques such as potential fields, artificial potential fields, and costmaps are employed to detect and circumvent obstacles along the planned path.

• Motion Control: PID controllers, trajectory tracking algorithms, and motion primitives enable precise control of the robot's velocity and orientation during navigation. .[39]

3.13 Integration of Localization and Navigation in ROS2:

Integration of localization and navigation in ROS2 involves coordinating the estimation of the robot's pose with the planning and execution of its movement. This includes:

• Sensor Fusion: Combining data from multiple sensors (e.g., odometry, IMU, lidar) to improve localization accuracy and robustness.

• Map Management: Updating and maintaining an accurate representation of the environment using SLAM or pre-built maps, which is used for localization and path planning.

• Feedback Control: Continuously updating the planned path based on real-time localization and sensor feedback to adapt to dynamic environments and ensure safe navigation. .[40]

3.14 Case Studies and Applications:

ROS2 localization and navigation techniques find applications in various robotic systems, including:

• Mobile Robots: Autonomous mobile robots for warehouse logistics, indoor navigation, and service robotics applications.

• Unmanned Aerial Vehicles (UAVs): Drones equipped with ROS2-based navigation systems for aerial mapping, surveillance, and search and rescue missions.

• Autonomous Vehicles: Self-driving cars and ground-based vehicles utilizing ROS2 for localization, path planning, and obstacle avoidance in urban and highway environments.

• Robotic Manipulators: Industrial robots equipped with ROS2-based navigation for autonomous navigation in manufacturing environments and collaborative robotics applications.

Case studies and real-world applications demonstrate the effectiveness and versatility of ROS2's localization and navigation capabilities across various robotic platforms and domains.

This chapter provides an overview of ROS2 basics, localization techniques, navigation systems and their integration, laying the foundation for understanding and developing advanced ROS2 applications in these following steps and chapter. .[41]

3.15 Navigation2 stack:

The Navigation2 stack is a critical and essential part of ROS2 framework. It addresses the significant challenges of enabling robots to navigate autonomously in diverse and un-predictable environments. As a core element of ROS2's ecosystem, Navigation2 includes a set of packages and libraries that equip robots with the ability to move, sense, and re- act in real time. The package provides an extensive collection of tools and algorithms to handle complex robotic tasks, such as path planning, obstacle avoidance and sensor data processing. Moreover, it makes use of the ROS2's architectural foundations, including its node-based architecture and communication functions, which enhance its effectiveness. .[42]

3.15.1 Navigation server

The navigation server serves as the foundation of the overall structure of Navigation2 stack. It is designed to guide robots through real-world spaces and is responsible for planning, controlling, and executing the autonomous ovement of robots in diverse environments. The structure of Navigation2 is present in Fig. 3.7 Controllers and planners are the "brain" of a navigation process. Recovery behaviors deal with the problem when robots encounter difficulties, offering solutions to potential issues and enhancing the system's fault tolerance. Smoothers optimize the planned path. A detailed introduction to these components and their roles in the navigation process is provided in this section. .[43]

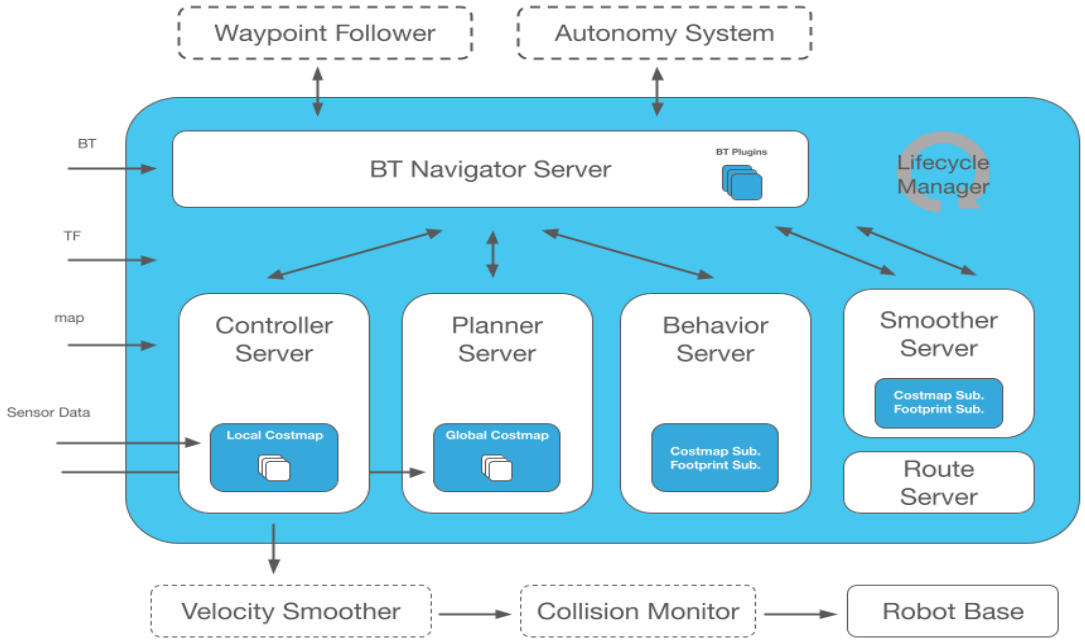


Figure 3.6: Nav2 architecture

- **Planner Server**

The primary function of the planner server is to consider the current location of a robot and its target position in terms of coordinates, and then to calculate optimal routes for guiding the robot to that predefined destination within a complex environment. The routes could be the shortest path, a complete coverage path, or along sparse or predefined routes, depending on the configuration of the planners. The objective of all routes is to enable robots to navigate safely, efficiently, and intelligently.

- **The controller server**

previously known as local planners in ROS1, is responsible for ensuring the precise execution of navigation routes generated by the planner server. While

the planner server calculates high-level navigation paths, the controller server translates these paths and breaks them down into a series of control commands that guide the robot's movements. Its primary function is to supervise the execution of these plans, adjusting the robot's trajectory as it navigates through the dynamic and challenging environment.

- **Behavior Server**

The behavior server is integrated with the behavior trees, allowing for a dynamic ap-proach to managing robot actions. One of the significant aspects of the behavior server is handling

recovery behaviors. The recovery behavior server is designed to deal with unknown or failure conditions that robots may encounter during navigation. The obsta-cles may include dynamic objects, temporary blockages, or items that did not initially appear on the navigation map. While the planners and the controllers focus on guiding the robot through a known and expected environment, the recovery server manages the unforeseen challenges to ensure that the robot can recover gracefully. This could involve backing up or spinning in place, attempting an alternative path, or even moving from a poor location into free space. .[44]

- **Smoother Server**

The general task of a smoother server is to receive a path from the planner server and output an enhanced version. It concentrates on improving the path's quality by consider- ing factors that may influence the robot's movement, such as kinematics and acceleration.

One of the crucial functions of the smoother server is to solve issues related to abrupt movements that may arise when following the navigation path. It aims to minimize these sudden changes in velocity or direction, and to increase the distance from obstacles and high-cost areas.

- **Waypoint Following**

Waypoint following is often utilized in conjunction with planners. A path is typically represented as a sequence of waypoints that define the route for the robot. The primary function of waypoint following is to provide precise control of the robot's movement. .[45]

3.15.2 TF tree

The TF tree (transform tree) in Nav2 is a data structure that organizes the frames in a robot system and manages the relationships between them dynamically . Each frame represents a coordinate system through which the positions of robots, sensors, and other objects can be expressed. The TF library in ROS2 is responsible for maintaining the TF tree. It works by broadcasting and listening to transform messages that describe the relationships between different frames. These messages contain the relative positions and orientations between a child frame and its parent frame. For instance, a simple robot (Fig. 3.8) might have a "base_link" frame representing its mobile base center, and a "base_laser" frame for the center of a mounted laser sensor.Data from the laser is transformed from the "base_laser" frame to

the "base_link" frame to help the robot avoid obstacles. This relationship is defined in the TF tree,

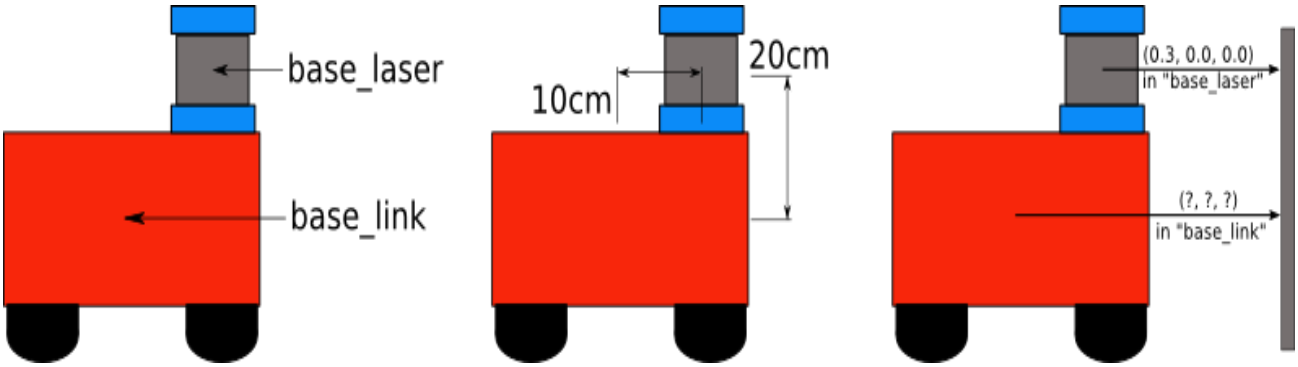


Figure 3.7: Base_link and base_laser frame in a robot

The structure of the TF tree used in the navigation process is presented in Fig. 3.10, with explanations provided for some of the main frames.

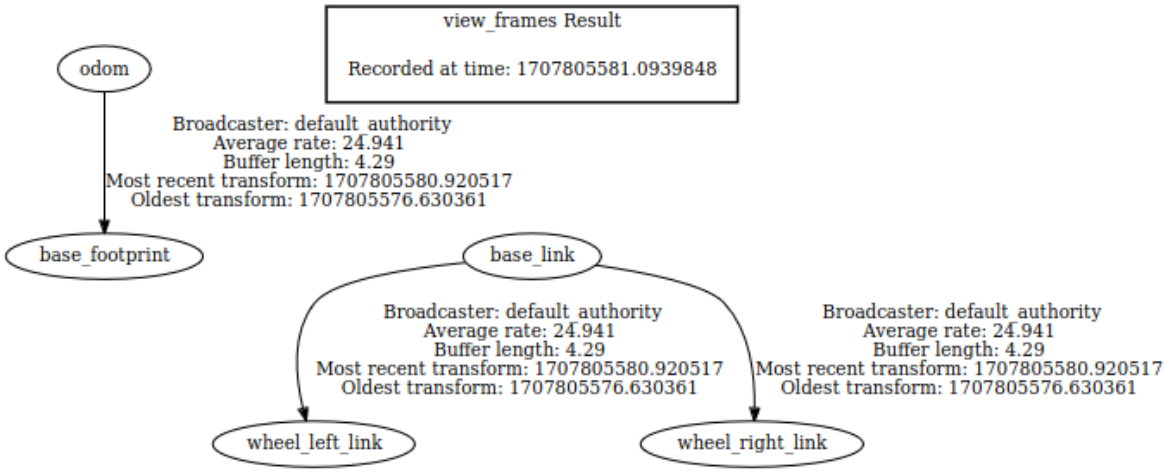


Figure 3.8: TF tree in Navigation2

• map frame: A global frame that represents a stationary reference on the map where all static objects are assumed to be fixed.

• odom frame: A local frame that represents the position and orientation of the robot based on odometry, which ca n move over time and provides relative motion information.

• base_footprint frame: This is usually a two-dimensional projection of the robot on the ground. It's the reference for the robot's contact with the floor or the ground.

• base_link frame: This is the robot's main body frame, from which positions and motions of all other parts of the robot are defined.

• base_scan frame: This frame is associated with a laser scanner or LIDAR sensor on the robot. It's the reference point from which scan data is measured.

3.15.3 Gazebo and Rviz

In the realm of robot navigation, the importance of testing and validation cannot be ignored. Ensuring that robots navigate in the environment with accuracy and safety is a complex task that requires extensive experimentation. To facilitate this, some powerful tools are utilized: Gazobo and Rviz.

Gazebo is an open-source, 3D robotics simulator that has gained popularity in the re-search and development communities. It offers a comprehensive, physics-based simulation environment, enables the modeling of complex building structure and robotic platforms. It also supports the integration of various sensors, such as LiDAR and cameras, which makes it a powerful tool for testing perception capabilities.

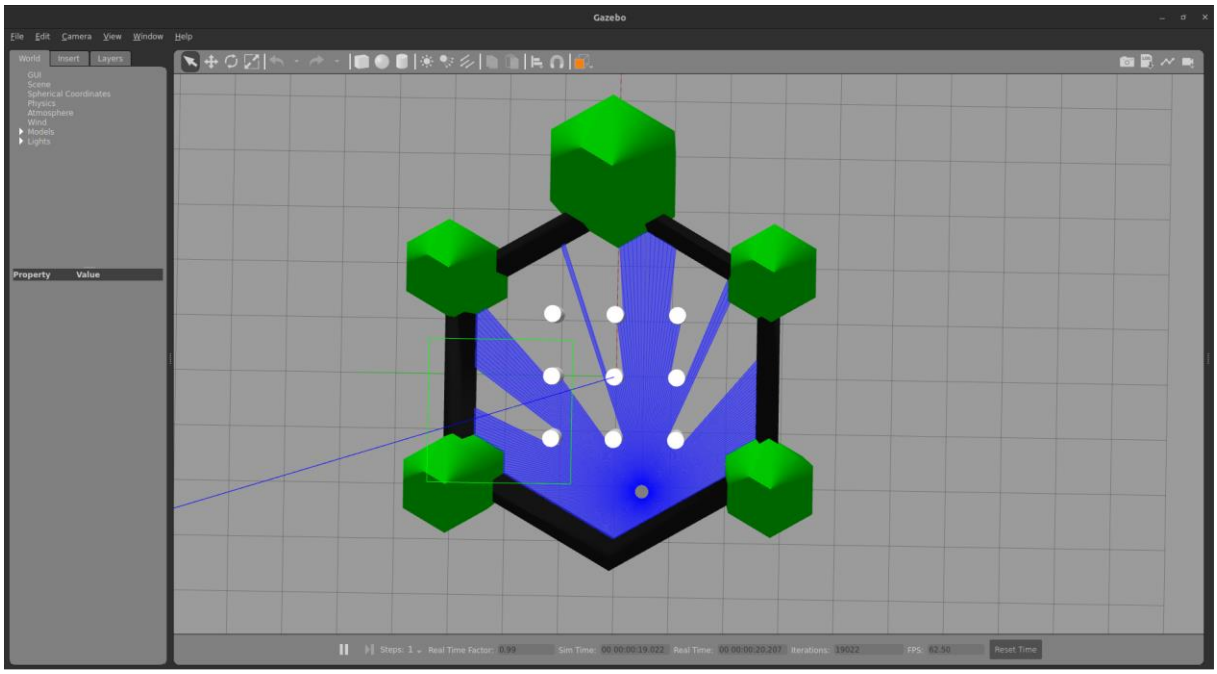
ROS Visualization (Rviz) is another necessary tool that complements Gazebo in the simulation environment. It enables real-time visualization of robot sensor data, assisting in the assessment of the navigation process performance [31]. With its interactive inter-face, users can manipulate robots and environments during the development and testing phases.

Fig. 3.10 is an example of the interface of Gazebo and Rviz. It is better to supply

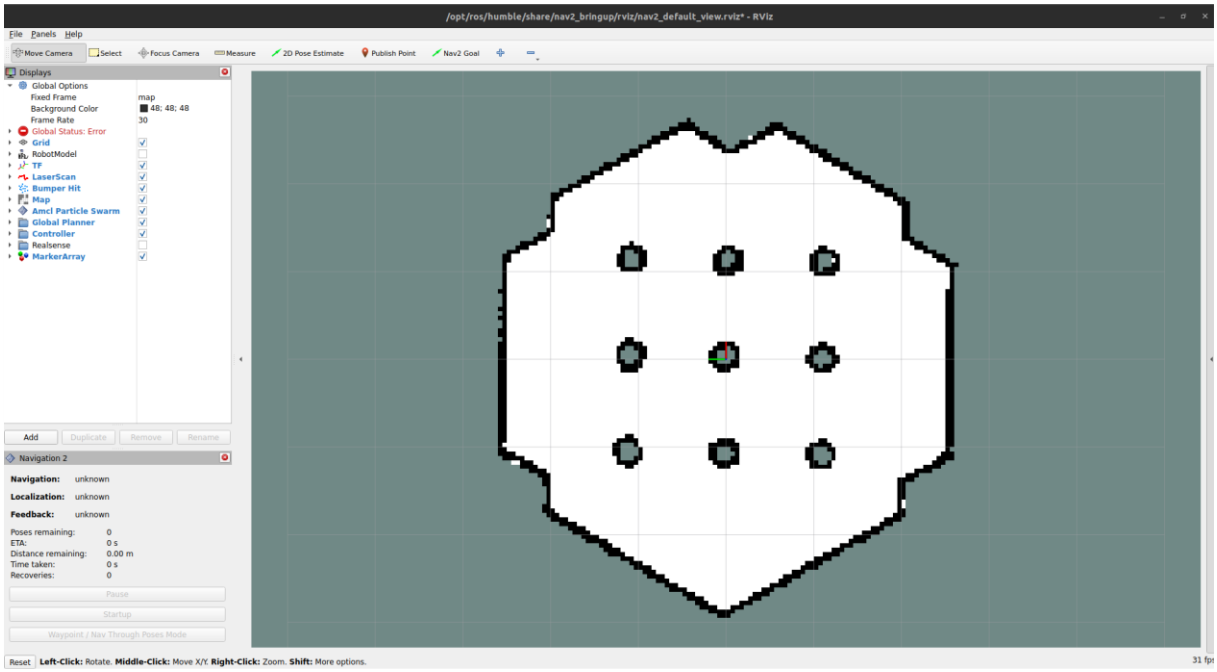
Rviz with a map for robot navigation. The map typically consists of two files: a PGM file and a YAML file. The PGM file is a gray scale image where black indicates obstacles, white represents free space, and shades of gray represent unknown areas. This file serves as the visual layout of the map. The accompanying YAML file (Listing 1.1) contains detailed data for the PGM file, including the path to the image file, the map's spatial resolution, a element array that defines the 2D pose of the lower-left pixel in the map, and thresholds for occupied and free spaces. Pixels with an occupancy probability above the 'occupied' threshold are considered fully occupied, while those below the 'free' threshold are considered completely free. .[46]

```
1 image : turtlebot3_world . pgm
2 resolution : 0.050000
3 origin : [ -10.000000 , -10.000000 , 0.000000]
4 negate : 0
5 occupied_thresh : 0.65
6 free_thresh : 0.196
```

Listing 1.1: Turtlebot3_world.yaml



(a) Gazebo interface



(b) Rviz interface

Figure 3.9: Tools utilized in the simulation

3.16 conclusion

This chapter provided a comprehensive overview of **ROS2**, detailing its distributed architecture, core components (nodes, topics, services), and communication patterns (topics, services, actions) powered by DDS middleware. Localization techniques (odometry, SLAM, particle filters) and navigation systems (path planning, obstacle avoidance, motion control) were explored, with a focus on the **Navigation2 Stack** and its components (planner, controller, behavior, smoother servers). Tools like **Gazebo** and **Rviz** support testing and visualization, while applications across mobile robots, UAVs, and autonomous vehicles demonstrate ROS2's impact. This foundation supports the practical implementation in Chapter IV, where ROS2 and TurtleBot3 are applied to real-world scenarios

Chapter IV: TurtleBot3 simulation with Gazebo in ROS2 environment

4.1 Introduction

In the realm of modern robotics, simulation has become an essential tool for developing and testing robotic systems before deploying them in real-world environments. Tools like ROS 2 (Robot Operating System 2) and Gazebo provide a robust environment for simulating autonomous mobile robots (AMRs) such as the TurtleBot3 Waffle. In this chapter, we present a practical example of simulating a mobile robot using ROS 2, focusing on setting up the environment, developing nodes, creating maps using SLAM (Simultaneous Localization and Mapping), and configuring autonomous navigation. We will start with installing the Ubuntu operating system and ROS 2 Humble, then move on to setting up a workspace, developing nodes, and running a simulation in a custom Gazebo environment called meza.world. The goal is to provide a comprehensive guide for simulating mobile robots using ROS 2 tools.

4.2 General description of the system

And before we get started, the terms that will be used should be clarified. When Navigation, Navigation 2, or Nav2 is mentioned, the Navigation 2 stack in ROS2 is being referred to. Similarly, when ROS is mentioned, ROS2 is also being implied.

As is probably already known, ROS2 is considered a powerful framework that enables the rapid development of new robotics applications. A lot of the “plumbing” is already provided, along with a vast set of plug-and-play libraries, a supportive community, and more.

Transitioning from just knowing the basics of ROS to successfully implementing navigation for a differential drive mobile robot (Fig. 4.1) is definitely not a trivial task. That’s exactly why the Navigation 2 (Nav2) stack was created — to help developers go from theoretical knowledge to a fully functioning navigation system.

This "stack" is a collection of packages all built with a single goal: to enable autonomous navigation for robots.

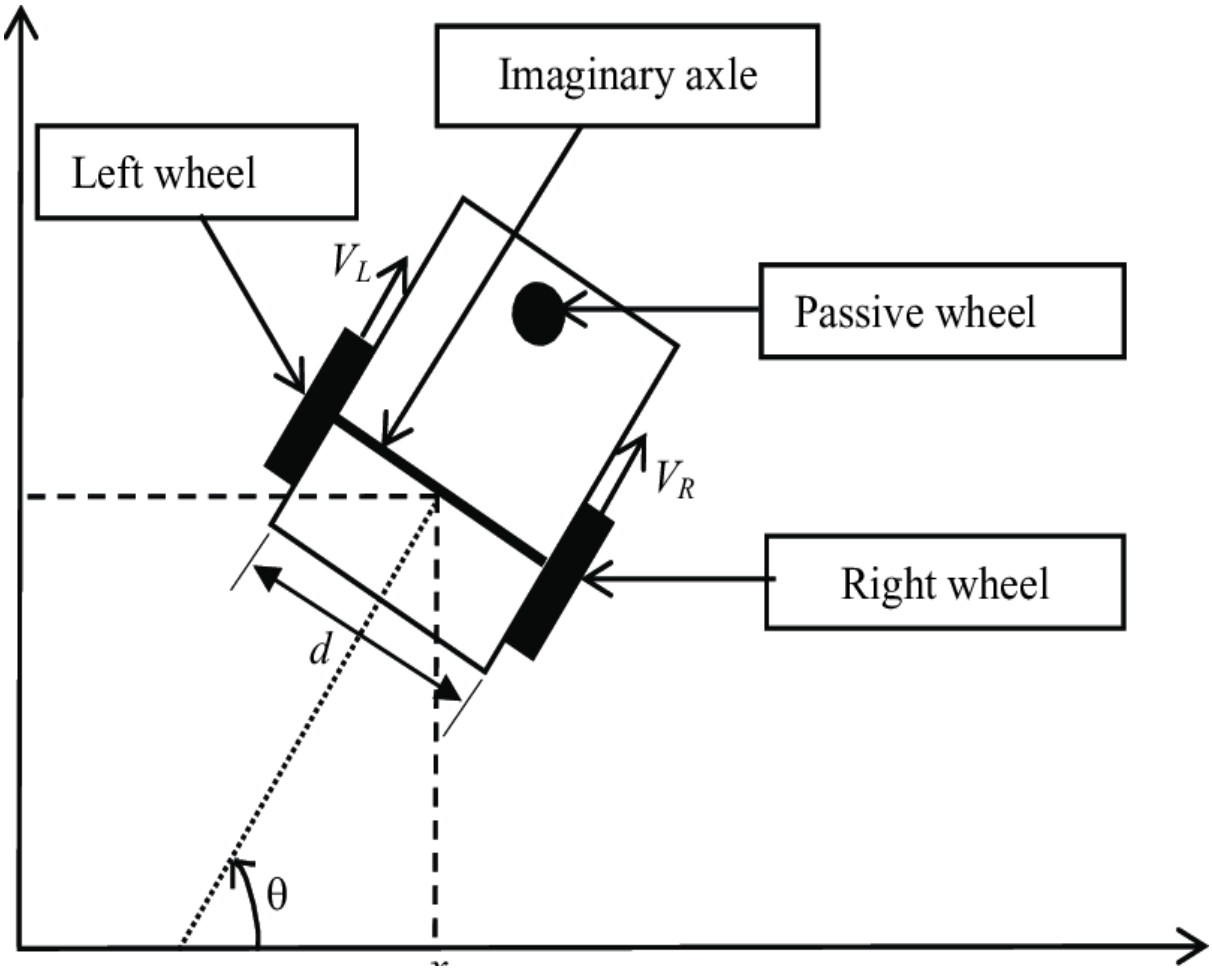


Fig.4.1: Basic Model of a Two-Wheel Differential Drive Robot

The main objective is for a differential drive robot to be moved from point A to point B in a safe manner (Fig 4.2).

In other words, a path must be found that enables the destination to be reached, while ensuring that collisions with other robots, people, and obstacles are avoided.

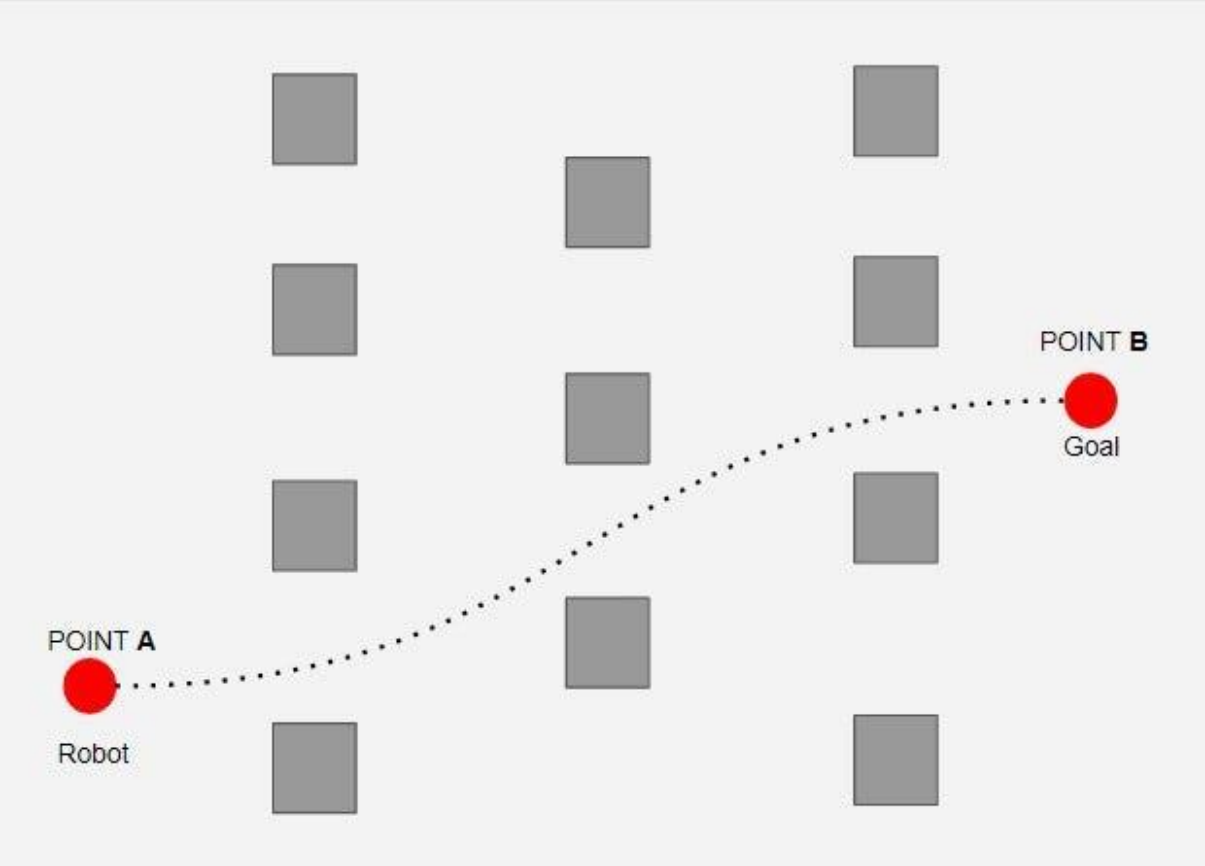


Fig.4.2: Safe point-to-point navigation of a differential drive robot

To achieve this, a two-step process will be followed:

a. A map of the environment will be created using SLAM.

b. The robot will be made to navigate using this map—with the navigation functionalities and tools.

As a prerequisite for this chapter, it should be ensured that Ubuntu and ROS2 have been installed on the computer. The use of a dual boot for Ubuntu is strongly recommended.

For the following examples, Ubuntu 22.04 and ROS2 Humble will be used.

So, the first step is to install Ubuntu 22.04, followed by ROS2 Humble, along with the tools needed for simulation, such as:

Gazebo – for 3D robot simulation in a virtual environment.

- **RViz2** – for visualizing sensor data, paths, and maps.
- **SLAM Toolbox** – for real-time map creation using sensors.
- **Nav2** – the core stack for autonomous navigation.

4.3 Ubuntu Installation

Ubuntu is the recommended operating system for ROS 2, as ROS is primarily developed and tested on Ubuntu-based distributions. Below is a detailed guide on how to install Ubuntu on a machine.

- **Steps to Install Ubuntu:**
  - **Download Ubuntu:**

- Visit the official Ubuntu website and download the Ubuntu version compatible with ROS 2. For ROS 2 Humble, it is recommended to use Ubuntu 22.04 LTS or Ubuntu 20.04 LTS.

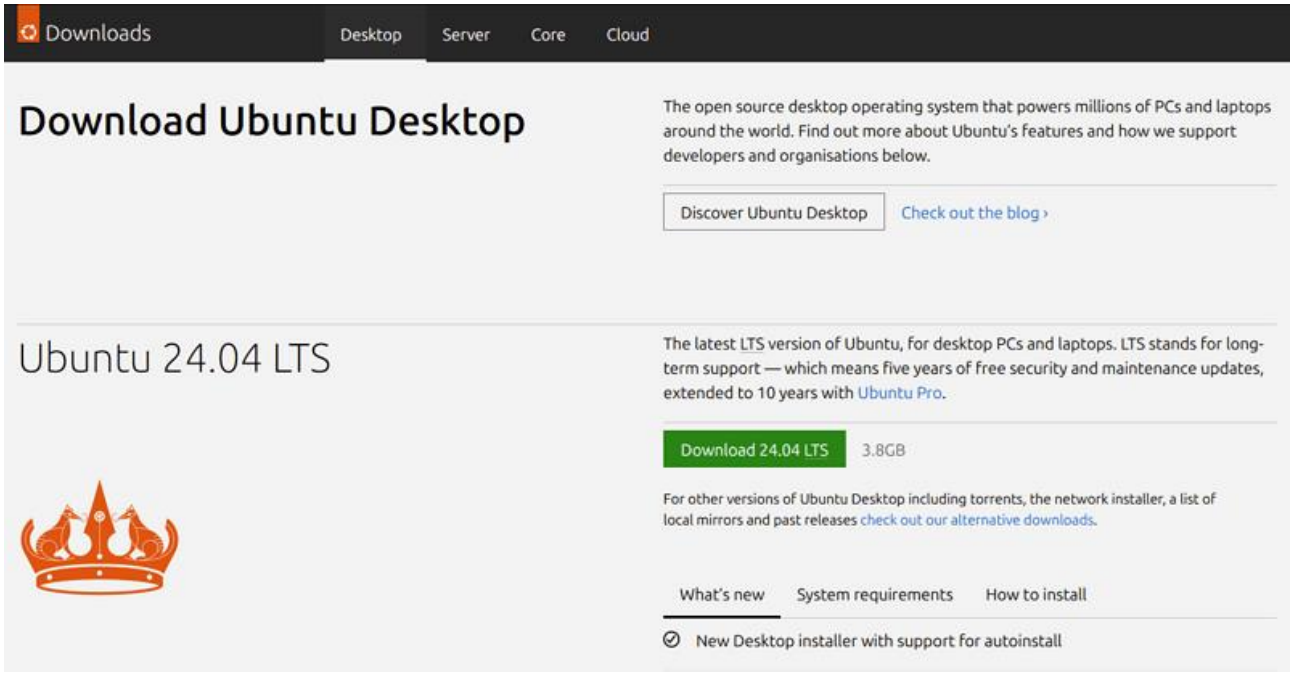


Figure 4.3: showing the Ubuntu 24.04 LTS download interface with the Ubuntu logo.

- **Create a Bootable USB:**
  - Use a tool like Rufus (on Windows) or Startup Disk Creator (on Ubuntu) to create a bootable USB stick with the downloaded Ubuntu ISO file.
  - Insert the USB stick into your machine and reboot. Make sure to set BIOS/UEFI to boot from USB.
- **Install Ubuntu:**
  - Once the machine boots from the USB stick, follow the instructions to install Ubuntu on your hard drive. Choose the "Install Ubuntu" option to guide you through the installation process.
  - Select your language, timezone, keyboard layout, and configure partitions if needed.

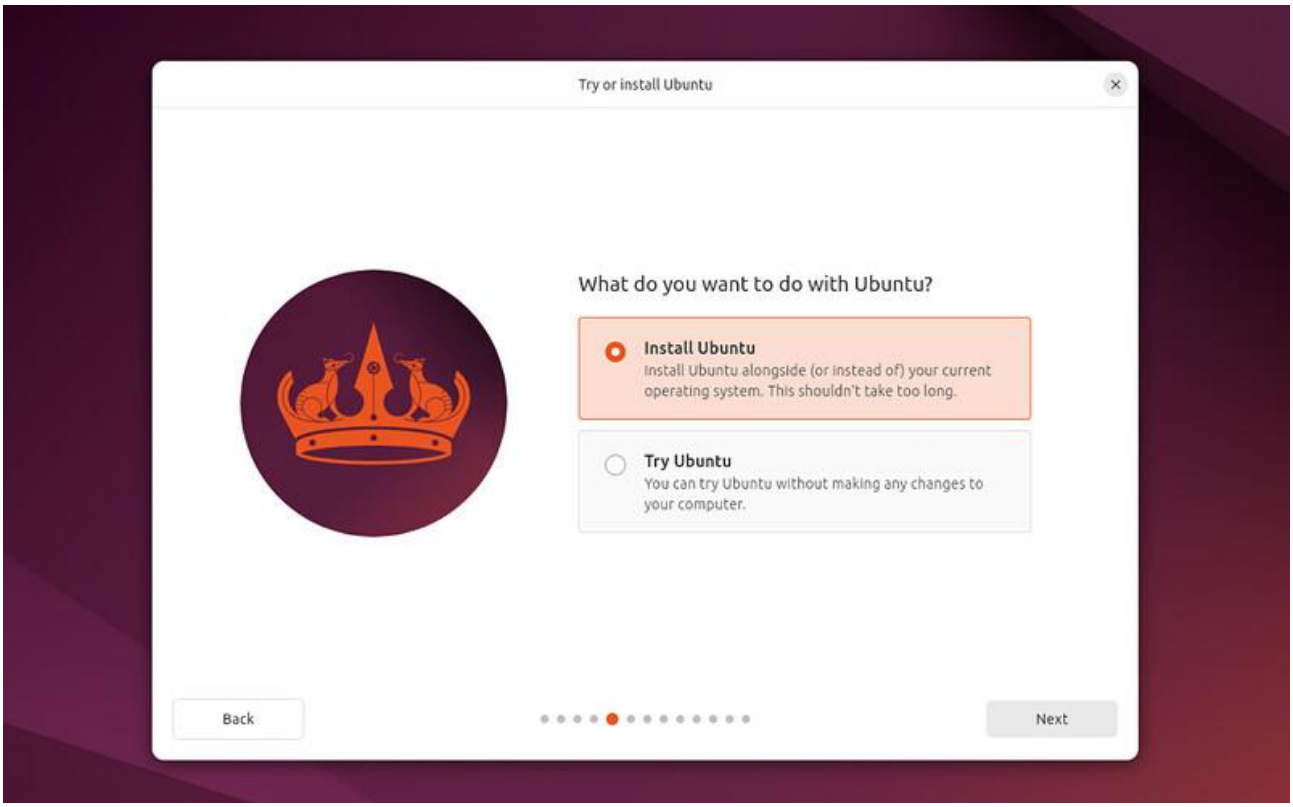


Figure 4.4: displaying the "Try or Install Ubuntu" options with the Ubuntu logo.

- **System Updates:**
    - After installation, ensure your system is up-to-date by opening a terminal and running:

```
sudo apt update
sudo apt upgrade
```

- **Install Additional Drivers:**
    - If your system requires additional drivers (such as for graphics or Wi-Fi), you can install them through the "Software & Updates" tool under the "Additional Drivers" tab.

4.4 Install ROS 2 Humble and Dependencies (Nav2, Gazebo, RViz)

In this step, you'll install ROS 2 Humble and the necessary dependencies: Nav2 (Navigation2 stack), Gazebo (for robot simulation), and RViz (for visualization).

- **Install ROS 2 Humble**

Before install ROS2 need to add the ROS 2 repositories to system:

. Add ROS 2 Repositories

```
sudo apt update
sudo apt install curl gnupg2 lsb-release
curl -sSL https://get.ros.org/ros2/ubuntu/ros2.key | sudo apt-key add -
sudo sh -c 'echo "deb [arch=amd64] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" >
/etc/apt/sources.list.d/ros2.list'
```

- **Explanation:**
    - curl: Downloads the ROS 2 signing key.

- gnupg2: A tool for managing GPG keys.
- lsb-release: Retrieves the Ubuntu version details.
- The echo command adds the ROS 2 repository to your system.

. Install ROS 2 Humble Desktop

Now, install the desktop version of ROS 2 Humble, which includes common tools and packages:

```
sudo apt update
sudo apt install ros-humble-desktop
```

- **Explanation:**
  - ros-humble-desktop: Installs the standard version of ROS 2, including the ROS 2 core libraries, tools like RViz and Gazebo, and dependencies like Navigation2.

. Set up ROS 2 Environment

To automatically set up the ROS 2 environment when opening new terminal windows, add the following to your .bashrc file:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

- **Explanation:**
  - The first command appends the ROS 2 setup command to your .bashrc file.
  - The second command sources the environment in your current terminal session so ROS 2 commands are available immediately.

- **Install Dependencies**

To install Nav2, Gazebo, and RViz, w'll use the following commands:

. Install Nav2 (Navigation2 Stack)

Nav2 is the ROS 2 Navigation stack used for autonomous robot navigation, including path planning and obstacle avoidance:

```
sudo apt install ros-humble-navigation2 ros-humble-nav2-bringup
```

- **Explanation:**
  - ros-humble-navigation2: This is the Navigation2 package for ROS 2, which provides functionality for autonomous navigation, path planning, mapping, and localization.
  - ros-humble-nav2-bringup: This package provides launch files and example configurations for setting up Navigation2 on a robot.

. Install Gazebo (Simulator)

Gazebo is a powerful robot simulator that allows you to simulate robots in realistic environments:

```
sudo apt install ros-humble-gazebo-ros-pkgs
```

- **Explanation:**
  - ros-humble-gazebo-ros-pkgs: This installs the necessary ROS 2 packages that integrate Gazebo with ROS 2, including plugins and tools for simulation.

. Install RViz (Visualization Tool)

RViz is a tool for visualizing robot sensor data, robot models, and other relevant data:

```
sudo apt install ros-humble-rviz2
```

- **Explanation:**
  - ros-humble-rviz2: This installs RViz 2 for ROS 2, allowing you to visualize robot states, sensor data, and other information in 3D.

4.5 Create a ROS 2 Workspace

A ROS 2 workspace is where stored ROS 2 packages. In this section, we'll go through the steps to create a ROS 2 workspace inside it.

4.5.1 Create the Workspace Directory

Use the following commands in your terminal to create the workspace:

```
mkdir -p ~/ros2_humble_ws/src
cd ~/ros2_humble_ws
```

- mkdir -p ~/ros2_humble_ws/src: Creates a directory called ros2_humble_ws in your home directory, with a subdirectory src where ROS 2 packages will reside.
- cd ~/ros2_humble_ws: Changes into the workspace directory.

4.5.2 Build the Workspace

Now that you created the workspace, you need to build it. Use colcon, the build tool for ROS 2, to build the workspace:

```
colcon build --symlink-install
```

- colcon build: Compiles the packages within your workspace. Since there are no packages in the workspace yet, this will simply initialize the workspace structure.

4.5.3 Source the Workspace

After building the workspace, source the setup.bash file to set up the environment for ROS 2 in this terminal session:

```
source ~/ros2_humble_ws/install/setup.bash
```

- source ~/ros2_humble_ws/install/setup.bash: Ensures ROS 2 commands work properly in the current terminal session.
- **Optional:** Add the source command to .bashrc so it's automatically sourced in every new terminal session:

```
echo "source ~/ros2_humble_ws/install/setup.bash" >> ~/.bashrc
```

```
source ~/.bashrc
```

4.6 Create a Node to Publish/Subscribe to a Topic

In this step, you'll create a node that publishes and subscribes to a topic. We'll walk through creating a simple publisher and subscriber in Python and C++.

4.6.1 Create a Python Publisher and Subscriber Node

4.6.1.1: Create the Publisher Node

In my_python_package, inside the my_python_package directory, create a new file called publisher.py:

```
touch my_python_package/publisher.py
```

Now, open publisher.py and write a Python publisher node:

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class PublisherNode(Node):
    def __init__(self):
        super().__init__('publisher_node')
        self.publisher_ = self.create_publisher(String, 'topic_name', 10)
        self.timer_ = self.create_timer(1.0, self.timer_callback)

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello, ROS 2 world!'
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)

def main(args=None):
    rclpy.init(args=args)
    node = PublisherNode()
    rclpy.spin(node)
    node.destroy_node()
```

```
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

4.6.1.2 Create the Subscriber Node

In the same my_python_package directory, create a file called subscriber.py:

```
touch my_python_package/subscriber.py
```

Now, open subscriber.py and write the subscriber node:

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class SubscriberNode(Node):
    def __init__(self):
        super().__init__('subscriber_node')
        self.subscription = self.create_subscription(
            String,
            'topic_name',
            self.listener_callback,
            10)
        self.subscription

    def listener_callback(self, msg):
        self.get_logger().info('Received: "%s"' % msg.data)

def main(args=None):
    rclpy.init(args=args)
    node = SubscriberNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

4.6.2 Modify setup.py to Include the Executable

.Open your setup.py **file** located in the my_python_package directory.

. Make sure the entry_points **section is correctly defined**. This ensures that ROS 2 knows how to execute the publisher.py and subscriber.py nodes when you use ros2 run.

Here's an example of what your setup.py should look like:

```
from setuptools import setup

package_name = 'my_python_package'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    install_requires=['setuptools', 'rclpy'],
    data_files=[
        ('share/ament_index/resource_index/packages',
         ['resource/' + package_name]),
    ],
    entry_points={
        'console_scripts': [
            'publisher = my_python_package.publisher:main',  # for the publisher node
            'subscriber = my_python_package.subscriber:main',  # for the subscriber node
        ],
    },
)
```

.Explanation:

- The entry_points section under console scripts tells ROS 2 how to run your Python nodes. It specifies that the publisher node is located in the my_python_package. publisher module, and it will execute the main() function from publisher.py.
- Similarly, the subscriber node is registered with the same pattern.

- ### Build the Package Again

. After modifying setup.py, go back to the workspace root directory:

```
cd ~/ros2_humble_ws
```

- Run the build command again to recompile your package:

```
colcon build --symlink-install
```

-Once the build finishes, source the setup file again:

```
source install/setup.bash
```

- **Run the Publisher Node**

-Now, try running the publisher node:

```
ros2 run my_python_package publisher
```

"The publisher node should now be seen running and outputting the messages 'Publishing: 'Hello, ROS 2 world!'."

Run the Subscriber Node

In another terminal, after sourcing the setup file, you can run the subscriber node:

```
ros2 run my_python_package subscriber
```

.Final Thoughts

The key to resolving the issue was ensuring that the setup.py file properly registers your Python nodes and their entry points. After that, rebuilding and sourcing the workspace will make the executables available to ROS 2. If everything is done correctly, both the publisher and subscriber nodes should work as expected.

"The publisher should be seen outputting messages every second, and those messages should be received by the subscriber."

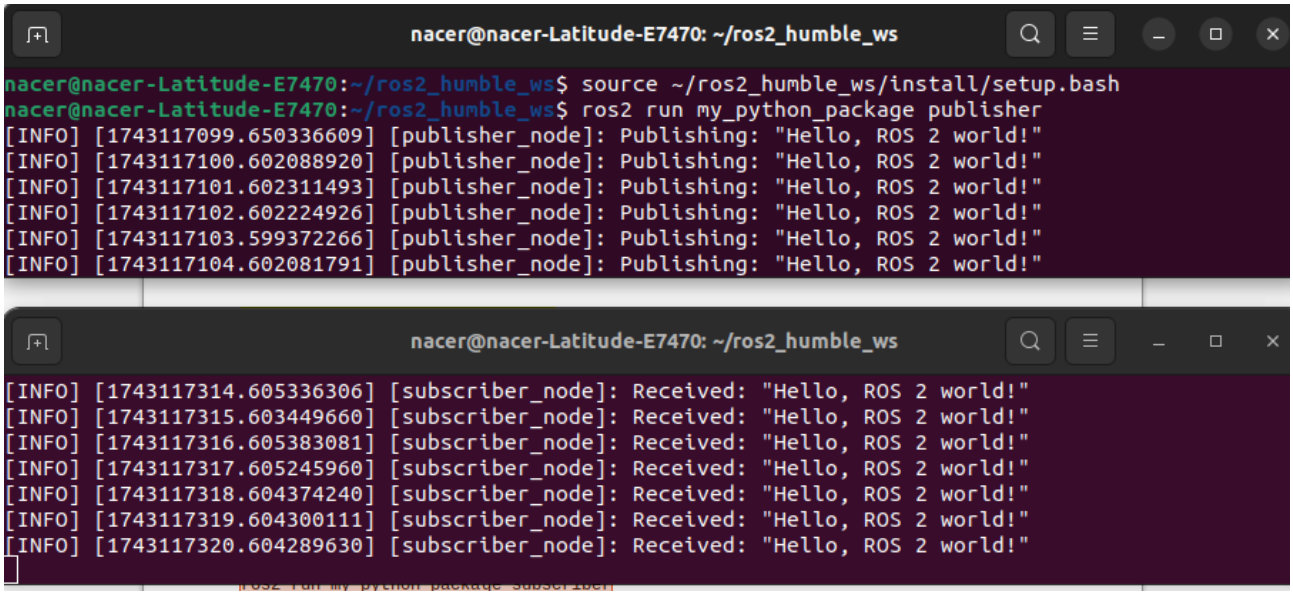


Figure 4.5: The first terminal screenshot showing the ROS publisher and subscriber nodes with the "Hello, ROS 2 world!" messages.

4.7 Create a Launch File to Launch Nodes

ROS 2 uses launch files to start multiple nodes at once. These files are written in Python.

. Create the Launch File

In the my_python_package, create a new directory called launch:

```
mkdir my_python_package/launch
```

Now, create a new launch file in this directory:

```
touch my_python_package/launch/publish_subscribe_launch.py
```

Open publish_subscribe_launch.py and write the following code:

```
import launch
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, LogInfo
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        # Launch publisher node
        Node(
            package='my_python_package',
            executable='publisher',
            name='publisher_node',
            output='screen'
        ),

        # Launch subscriber node
        Node(
            package='my_python_package',
            executable='subscriber',
            name='subscriber_node',
            output='screen'
        ),
    ])
```

Explanation:

- This launch file will launch two nodes: publisher_node and subscriber_node from the my_python_package package.
- output='screen' ensures that the node's output will be displayed in the terminal.

. Launch the Nodes

After creating the launch file, you can use the ros2 launch command to launch both nodes at once:

```
ros2 launch my_python_package publish_subscribe_launch.py
```

This will start both the publisher and subscriber nodes, and you should see messages being published and received in the terminal.

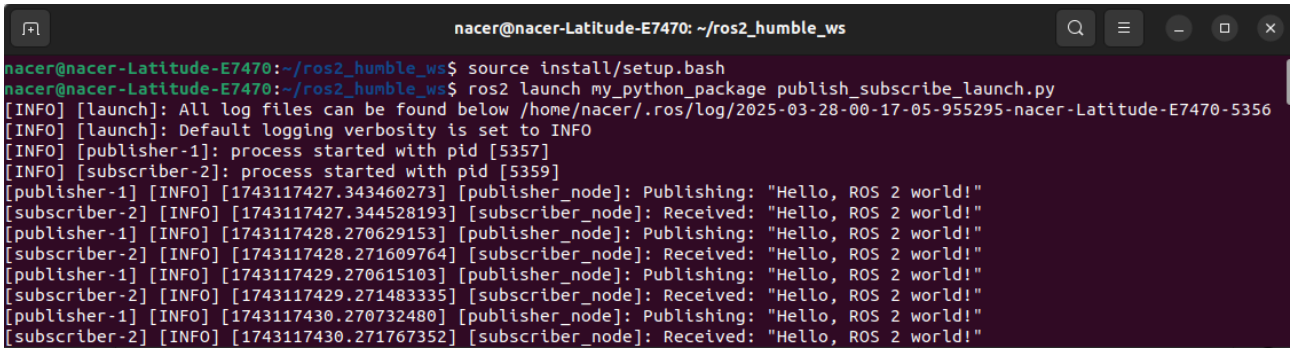


Figure 4.6: The second terminal screenshot showing the ROS launch file execution and multiple publisher/subscriber nodes.

4.8 Create a Launch File to Launch Multiple Launch Files

Sometimes, you may want to launch several different launch files at once. This is useful when you need to start multiple configurations or sets of nodes. In ROS 2, you can include one launch file within another.

. Create the Main Launch File

In the launch directory of your my_python_package, create a new launch file:

```
touch my_python_package/launch/main_launch.py
```

Now, open main_launch.py and write the following code to include the previous launch file (publish_subscribe_launch.py):

```
import launch

from launch import LaunchDescription

from launch.actions import IncludeLaunchDescription

from launch.launch_description_sources import PythonLaunchDescriptionSource

def generate_launch_description():

return LaunchDescription([

IncludeLaunchDescription(

PythonLaunchDescriptionSource(

'/home/nacer/ros2_humble_ws/my_python_package/launch/publish_subscribe_launch.py'

),

),

])
```

Explanation:

- This launch file includes another launch file, allowing you to start multiple nodes or configurations by calling another launch file.
- You can replace /path/to/your/package with the actual path to your package or environment variable.

. Launch the Multiple Launch Files

To launch both launch files, use the following command:

```
ros2 launch my_python_package main_launch.py
```

This will execute all the nodes and launch files specified within main_launch.py, launching the publish_subscribe_launch.py file and all of its nodes (publisher and subscriber in this case).

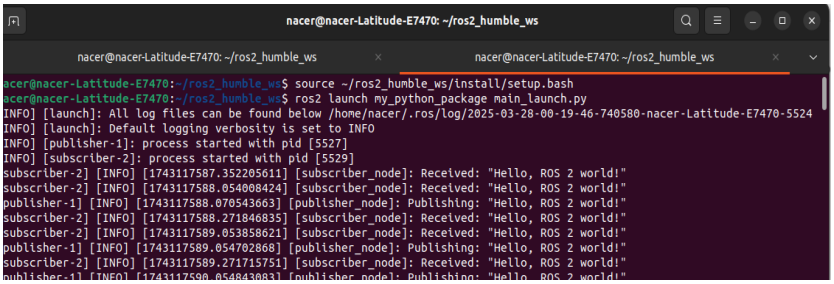


Figure 4.7: The terminal screenshot showing the ROS launch file execution with multiple publisher and subscriber nodes (with "Hello, ROS 2 world!" messages).

4.9 Simulation the robot turtlebot3 withe nav2 stack on ros2 humble

4.9.1 Overview

to be first installed the Nav2 stack on ROS2 Humble, then to start directly to generate a map with SLAM. After this, we will check what's inside the map, and then use this map with the Navigation 2 stack, so that we can make a robot navigate inside the map.

To do all that, you actually don't need to buy any hardware, we will use the simulation of an already configured robot, the Turtlebot3, on Gazebo.

4.9.2 Install Nav2

As a prerequisite for this chapter, make sure you have installed Ubuntu and ROS2 on a computer.

For the following, I will use Ubuntu 22.04 and ROS2 Humble.

So, once have correctly installed and setup ROS2 on a computer, install the Navigation 2 stack.

install the Nav2 packages.

```
$ sudo apt install ros-humble-navigation2 ros-humble-nav2-bringup
```

As to use a simulation of a robot on Gazebo, to be installed the packages for this robot (Turtlebot3).

```
$ sudo apt install ros-humble-turtlebot3
```

And that's it to get started!

4.9.3 Make the robot move in the environment

To make the robot move in the environment, the following steps should be taken:

Before a map is generated with Nav2 and used for navigation, it is essential to ensure that the robot can move within the environment.

If a custom robot is being used, it should first be adapted for Nav2. This process can be challenging, especially for those who are doing it for the first time. To simplify things – as the primary goal here is to introduce the stack – the Turtlebot3 in Gazebo will be used.

An environment variable needs to be exported to select the version of the Turtlebot3 to use (burger, waffle, waffle_pi).

This line should be added at the end of the. bashrc file: export TURTLEBOT3_MODEL = waffle. After this is done, the. bashrc file should be sourced, or a new terminal should be opened. Afterwards, the simulated robot should be started in a Gazebo world.

```
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

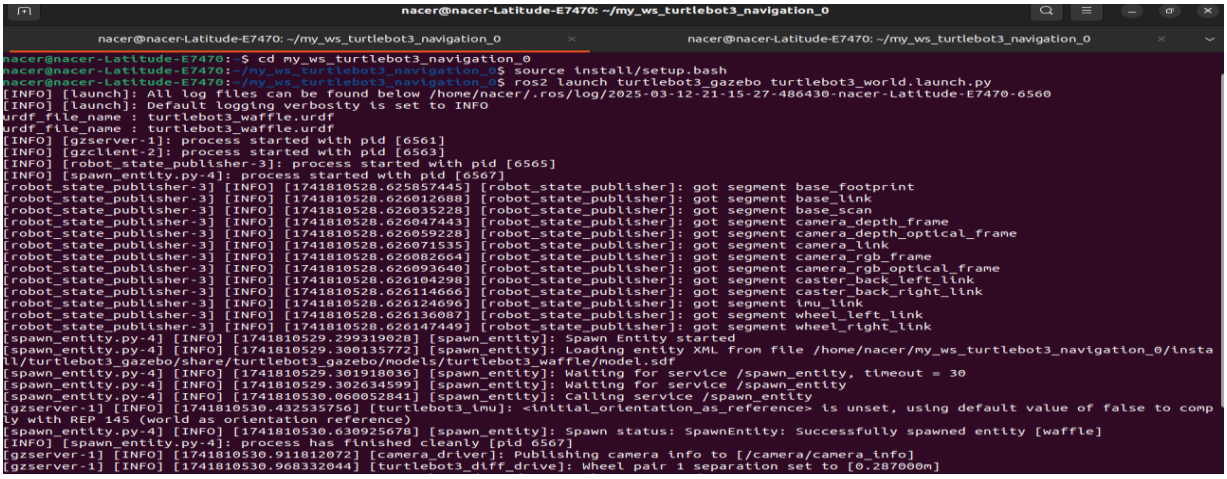


Figure 4.8: The terminal screenshot showing the ROS launch with detailed node information and "turtlebots_waffle"

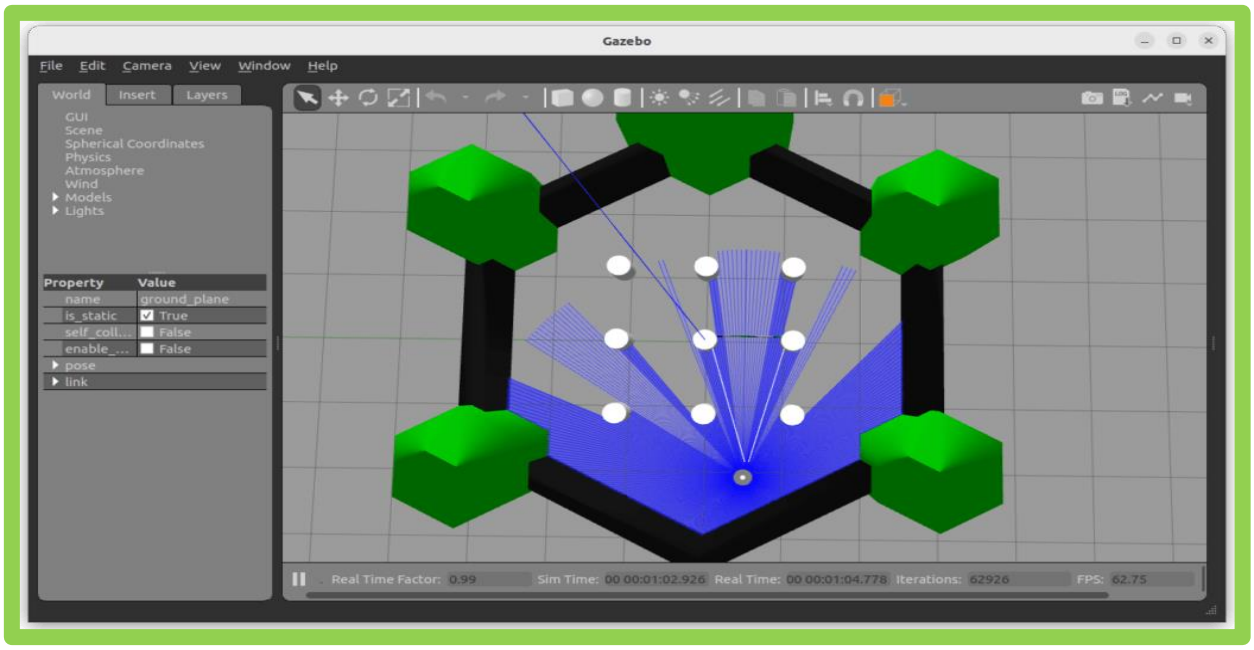


Figure 4.9: The Gazebo simulation screenshot displaying a robot model with sensor data visualization (Gazebo window).

- **Analysis of Terminal Output**
  - The command `source install/setup.bash` is executed, activating the ROS 2 environment without errors. A properly built workspace is confirmed, which must be replicated for the `ros2_ws` workspace associated with `m2wr`.
  - command `ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py` is run, initiating the simulation, with logs stored at `/home/my_home/.ros/log/xxxxxx` and verbosity set to INFO.
  - The Gazebo server (`gzserver`) process is started with PID 8436, the Gazebo client (`gzclient`) with PID 8438, the `robot_state_publisher` with PID 8440, and the `spawn_entity.py` with PID 8442.
  - The `robot_state_publisher` node is utilized to process the `turtlebot3_waffle.urdf` file, with frames like `base_footprint`, `wheel_left_link`, and `wheel_right_link` recognized. Compatibility with `m2wr` URDF frames (e.g., `link_chassis`, `wheel_left`, `wheel_right`) is anticipated.
  - The `spawn_entity.py` node is employed to load `/model.sdf`, with a 30-second wait for `/spawn_entity`, and the `waffle` entity is successfully spawned (REP 145 compliant). The `spawn_robot.launch.py` for `m2wr` should be validated to ensure similar spawning success.
  - Camera information is published to `/camera/camera_info` by the `camera_driver` node, `turtlebot3_diff_drive` is configured with a 0.287m wheel separation and 0.066m diameter, `/cmd_vel` is subscribed to, `/odom` is advertised, and joint states for wheels are prepared by `turtlebot3_joint_state`. Corresponding configurations (e.g., wheel dimensions, `/m2wr/odom`) are required for `m2wr`.
- **`rqt_graph` Image**

rqt_graph is a powerful graphical introspection tool within the ROS 2 ecosystem, designed to visualize the ROS computation graph. rqt_graph displays all active ROS 2 nodes and the topics that connect them. This allows for a clear understanding of the data flow within your robotic system. It visually highlights the publishers and subscribers for each topic, illustrating how different nodes interact and exchange information.

In the graph below, the ROS 2 robot_state_publisher node automatically subscribes to the /joint_states topic to receive sensor_msgs/msg/JointState messages. Laser scanner driver publishs sensor_msgs/msg/LaserScan messages to the /scan topic, making the sensor data available to other nodes in the ROS 2 graph. The turtlebot3 node integrates the diff_drive_controller from the ros2_controllers' package. This controller takes linear and angular velocity commands (typically received on the /cmd_vel topic as

a geometry_msgs/Twist message) and translates them into individual wheel velocities for the left and right wheels.

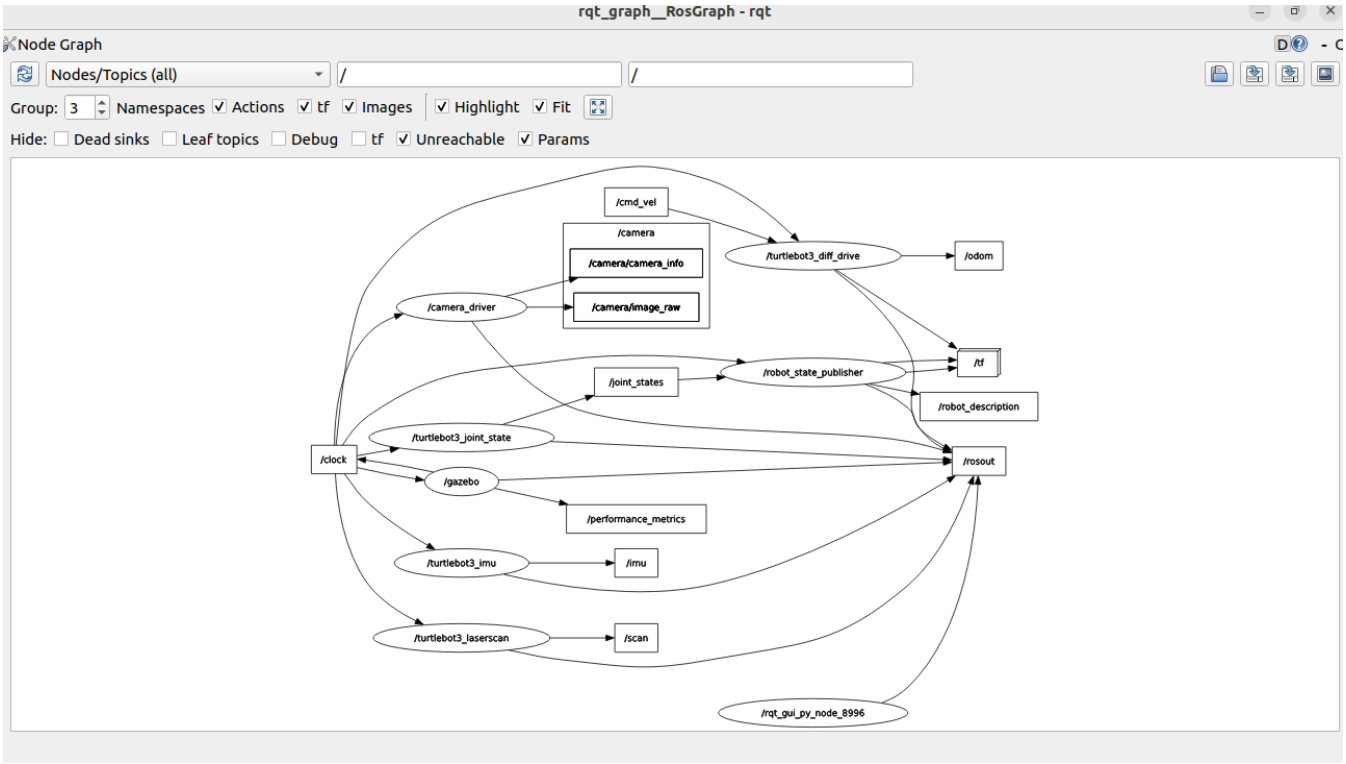


Figure 4.10: ROS 2 Node Graph of TurtleBot3 Simulation

now a robot is inside a room. To make it move, the teleop node should be started in a new terminal using the following command:

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

```
nacer@nacer-Latitude-E7470:~/my_ws_turtlebot3_navigation_0$ source install/setup.bash
nacer@nacer-Latitude-E7470:~/my_ws_turtlebot3_navigation_0$ ros2 run turtlebot3_teleop teleop_keyboard

Control Your TurtleBot3!
---------------------------
Moving around:
        w
   a    s    d
        x

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit

currently:      linear velocity 0.0     angular velocity 0.0
currently:      linear velocity -0.01   angular velocity 0.0
currently:      linear velocity -0.02   angular velocity 0.0
currently:      linear velocity -0.03   angular velocity 0.0
currently:      linear velocity -0.03   angular velocity -0.1
currently:      linear velocity -0.03   angular velocity -0.2
currently:      linear velocity -0.03   angular velocity -0.30000000000000004
```

Figure 4.11: Teleoperation of TurtleBot3 Using ROS 2 Teleop Keyboard

This figure 4.11 shows the terminal output when controlling the TurtleBot3 Waffle robot using the `teleop_keyboard` node in ROS 2. The user can move the robot interactively with keyboard keys:

- **w / x**: increase or decrease linear velocity (forward/backward).
- **a / d**: increase or decrease angular velocity (rotation).
- **space**: emergency stop.
- **CTRL + C**: quit the teleoperation mode.

The terminal continuously displays the robot's current **linear velocity** and **angular velocity**, which update according to the user's inputs. This provides real-time feedback on motion commands sent to the robot.

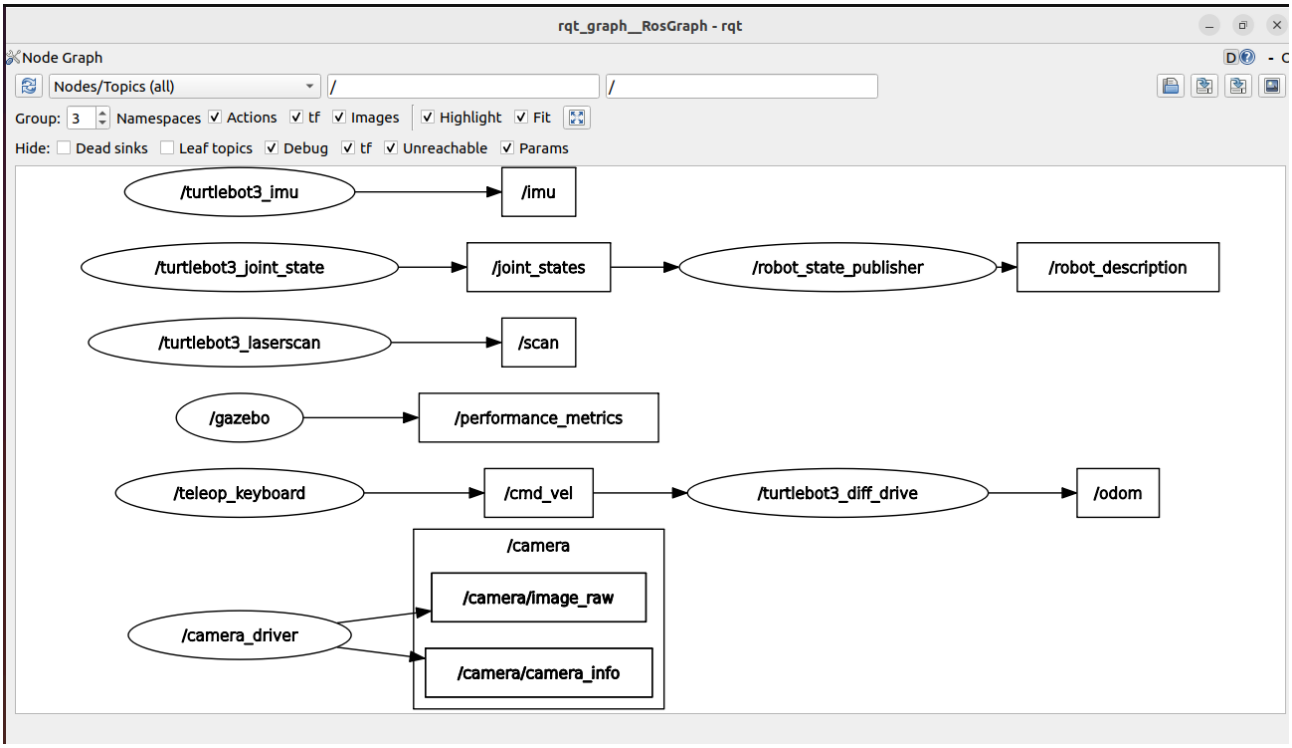


Figure 4.12: ROS2 Computation Graph for the TurtleBot3 Simulation with a keyboard and receives data from virtual sensors.

- This action can be replicated for m2wr by developing a teleoperation node to publish /m2wr/cmd_vel and monitor movement.

- The use of robot_state_publisher and differential drive integration is paralleled by m2wr's libgazebo_ros_diff_drive.so.

4.9.4 Generate a map with ROS2 Nav2 – using SLAM

In order for the robot to navigate autonomously in the world, a map of the world must first be generated. To accomplish this, the SLAM functionality of the ROS2 Nav2 stack should be used.

Note: There are various tools and algorithms available for SLAM. To simplify the process, the default SLAM tool for Turtlebot3, Cartographer, will be chosen.

4.9.4.1 Start the SLAM functionality and Rviz

The SLAM functionality and RViz should be started as follows:

First, it should be ensured that the robot – in this case, Turtlebot3 in Gazebo – has been started using the following command:

```
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

Next, the SLAM functionality for Turtlebot3 should be started with the command:

```
$ ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=True
```

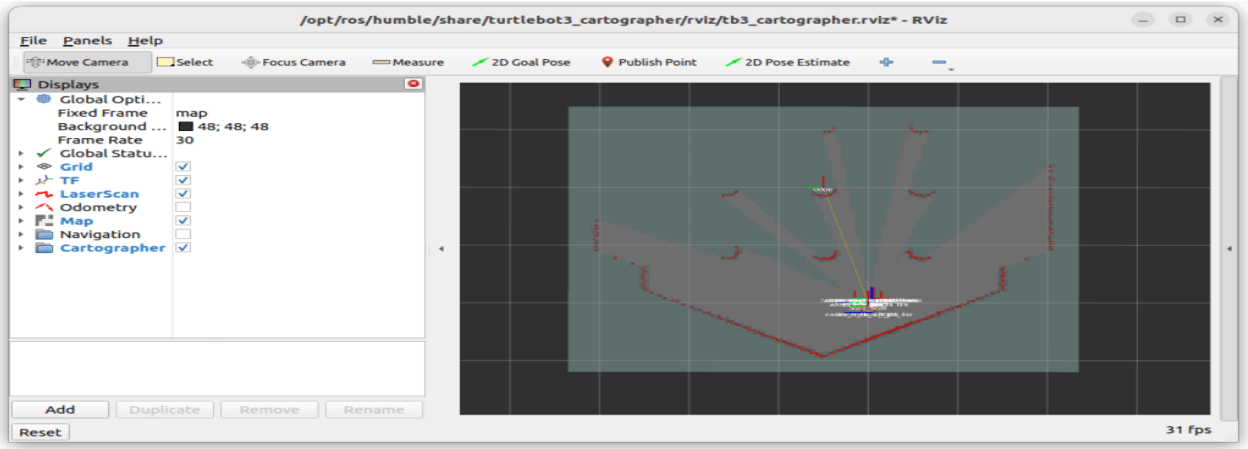


Figure 4.13: ROS2 Rviz Visualization for TurtleBot3 Cartographer

. The "use_sim_time" argument is provided because Gazebo time is being used. If a real robot is being used, this argument should be skipped.

Once the command is executed, RViz will be started automatically.

The TFs of the robot, along with the LaserScan data from the Lidar sensor (represented by red dots), are visible. As the robot moves, pixels are cleared into three categories:

. Free space: These pixels will turn white.

. Obstacle: These pixels will turn black.

.Unknown space: These pixels will remain grey.

4.9.4.2 Generate and save the map

To make the robot move in the world, the following steps are taken:

.Start the TurtleBot3 Teleop Node: The teleop node is started using the following command:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

.Move the Robot: The robot is made to move around the environment using the keyboard. As the robot moves, the map will be updated automatically. The sensor data from the Lidar (represented by red dots) is used to clear the map in the following ways:

- **Free space** (white pixels) is cleared as the robot explores more of the environment.
- **Obstacles** (black pixels) are marked as the robot detects them.
- **Unknown space** (grey pixels) remains until the robot moves through those areas.

.Achieving a Satisfying Result: A satisfying result is reached when most of the **free space** (white pixels) is cleared and most **obstacles** (black pixels) are detected and marked. There is no need to clear 100% of the pixels; as long as the robot has mapped a significant portion of the environment with accurate obstacle detection and cleared free space, navigation will work well.

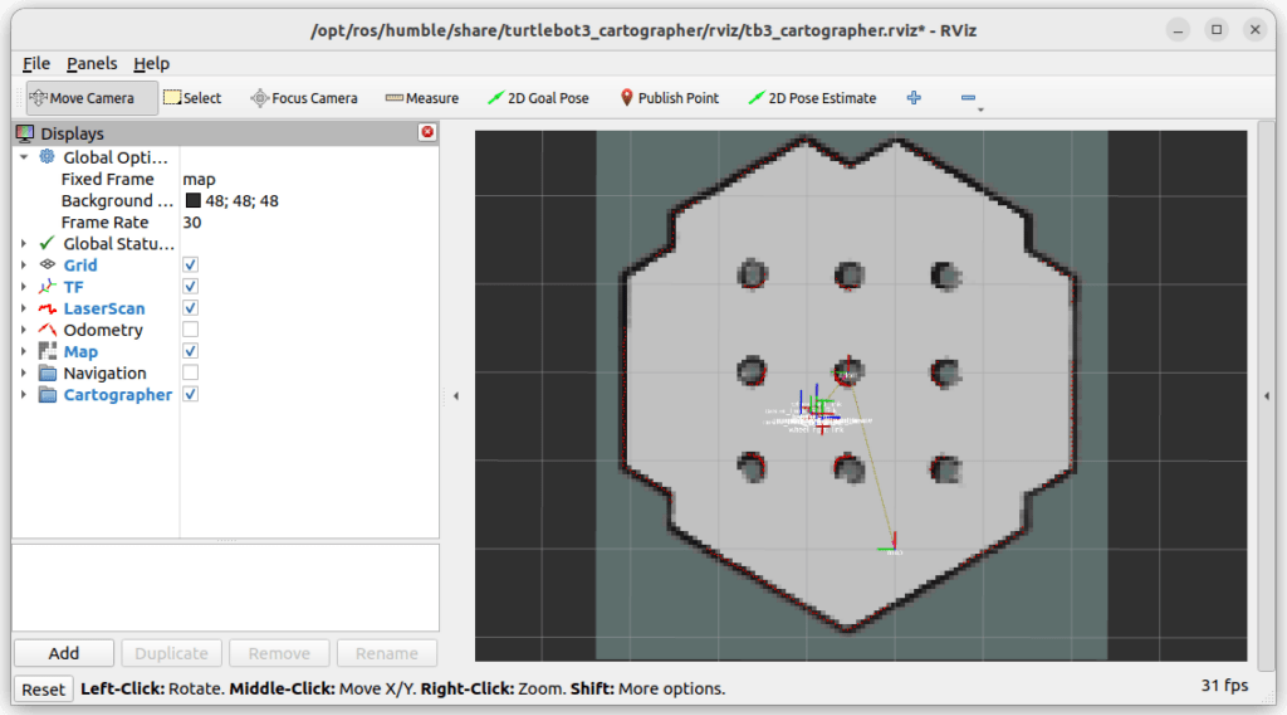


Figure 4.14: Final, Saved Map of The Robot's Environment

Once a good enough result is achieved, the map is saved. It is important not to stop the navigation terminal before saving the map, as doing so would require starting the process over again.

To save the map, a new terminal is opened and the following command is run:

```
ros2 run nav2_map_server map_saver_cli -f my_map
```

The optional -f argument is used to specify the path or name for the map. No extension needs to be added, as it will be handled automatically.

After running the command, logs in white (and possibly yellow) should appear. If error logs in red appear, the command should be retried several times.

Once the map has been successfully saved, all commands in the terminals can be stopped.

4.9.5 Make the Robot Using the Map and ROS2 Nav2

Great! You now have a map and everything is set up to make the robot navigate autonomously while avoiding obstacles.

4.9.5.1 Starting the Navigation 2 for the Robot:

.Start with a Clean Environment: To reduce potential issues with RViz and Gazebo, it is recommended to stop everything, close, and reopen all terminals.

.Launch the Robot in the World: In the first terminal, launch the robot in the simulation environment:

```
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

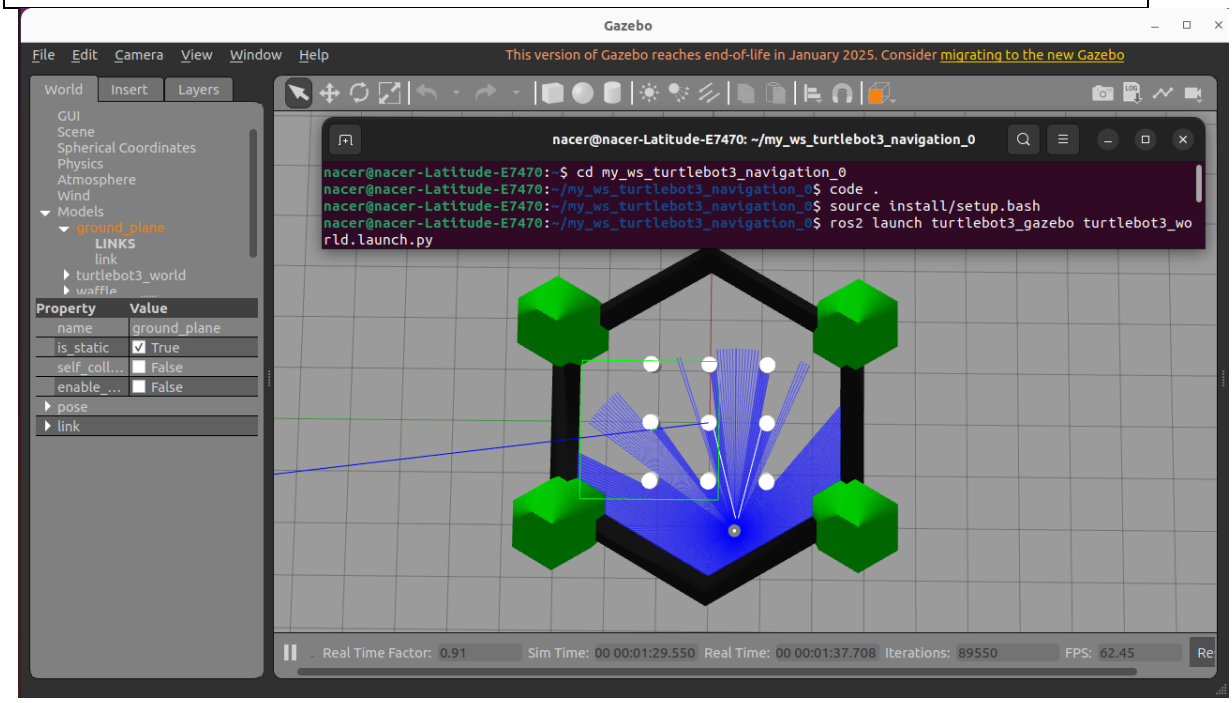


Figure 4.15: Gazebo Simulation of the TurtleBot3 Robot.

.Start the Navigation Stack: In a new terminal, start the Navigation stack and provide the map file as an argument:

```
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=False
map:=/home/nacer/map/my_map.yaml
```

- **use_sim_time**: Set to True if running with Gazebo.
- **map**: Provide the path to the YAML file of the map.

.Launche RVIZ

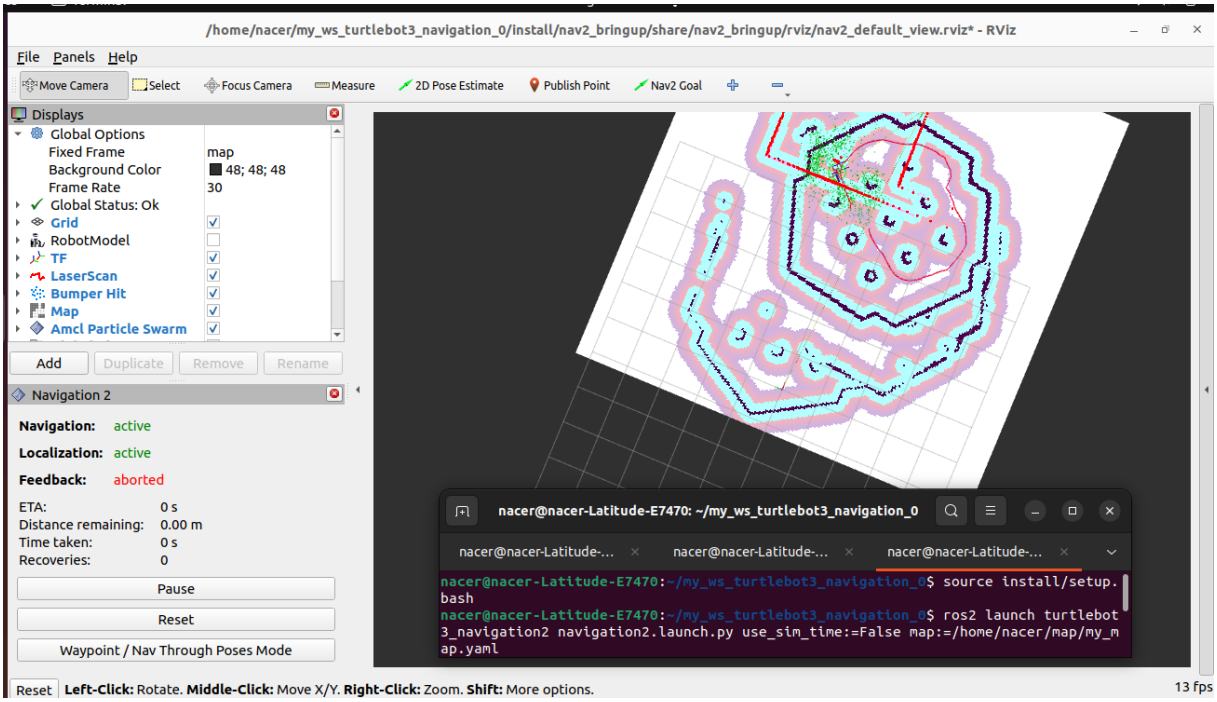


Figure 4.16: RViz Visualization of a Map with an localized Robot.

.Display the Map in RViz: If the map is not visible in RViz, locate the "map" topic option on the left panel and change it to "transient local" instead of "volatile." If the map still doesn't show, try restarting everything, or even rebooting your computer.

.Issue with the Robot Pose:

At this point, the map should be visible on the screen, but no robot will be shown.

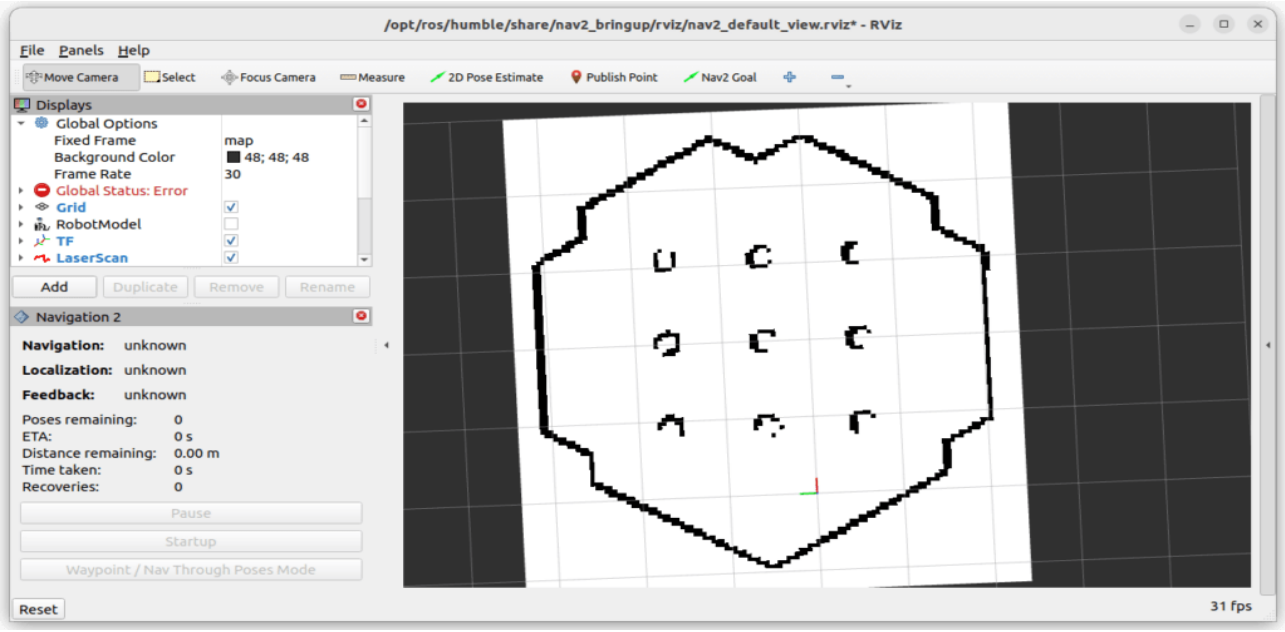


Figure4.17: RViz State Before Initial Pose Estimate Withe Map Saved.

Additionally, error logs may be observed in the terminal. This occurs because the location of the robot is not known to Nav2, and an initial 2D pose estimate needs to be provided.

4.9.5.2. 2D Pose Estimate and Navigation Goals

To provide the first 2D pose estimate for the robot:

. Click on the "2D Pose Estimate" Button in RViz: In RViz, locate and click on the "2D Pose Estimate" button.

. Click on the Map: On the map displayed in RViz, click where the robot is located (you should be able to see the robot's position in Gazebo as well). Hold the click and drag to also specify the robot's orientation using a green arrow.

. Completion: Once the pose is set, you should see something similar to the 2D pose estimate in RViz, indicating the robot's position and orientation on the map.

At this point, the robot's initial position and orientation have been estimated, and navigation can proceed.

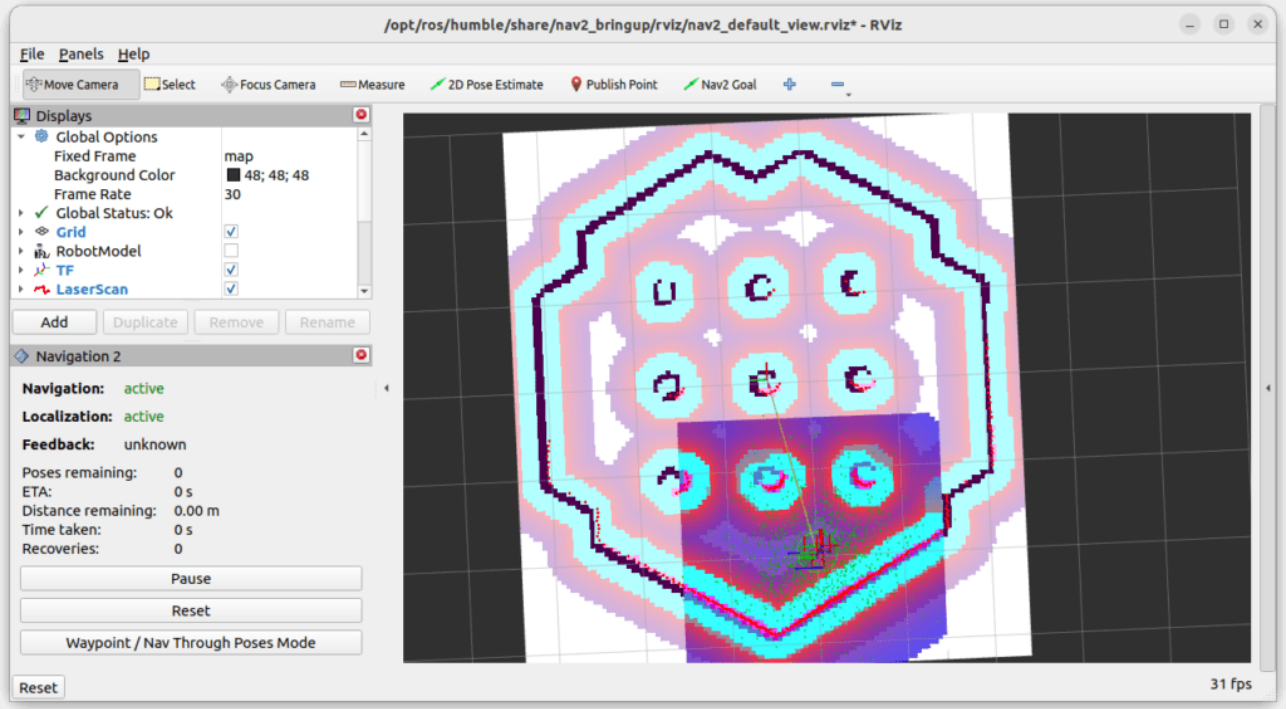


Figure 4.18: RViz visualization of the robot's pose after the initial 2D estimate is given.

.Finally, Navigation Commands!

To give navigation commands to the robot:

.Click on the "Nav2 Goal" Button: In RViz, click on the "Nav2 Goal" button.

.Select a Position and Orientation: Click on the map to select a goal position and orientation. The robot should begin navigating toward that pose.

.Verify the Robot's Movement: You can verify that the robot is actually moving by checking its position in Gazebo. The robot should follow the selected path and reach the desired pose.

At this point, navigation commands are successfully given, and the robot moves toward the target pose.

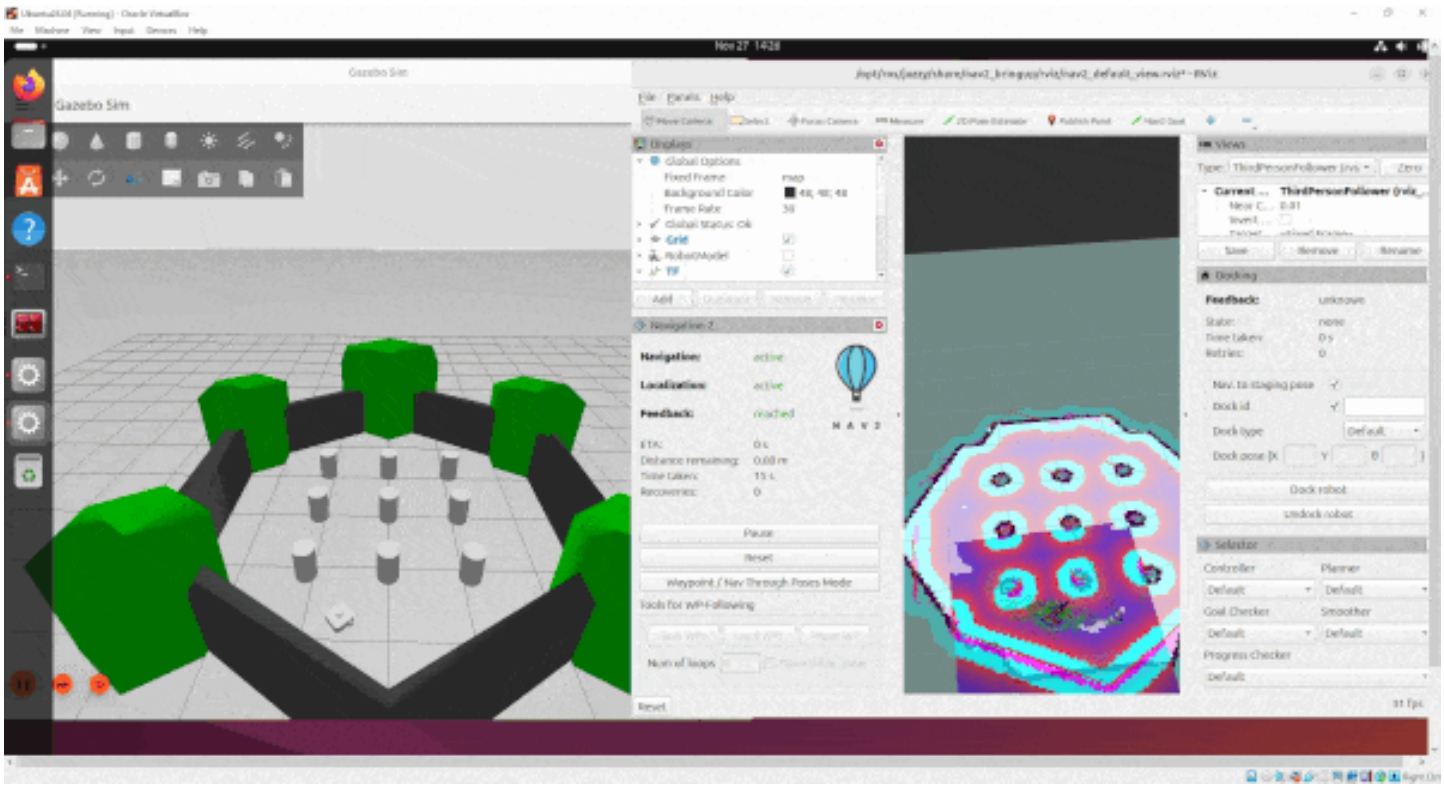


Figure 4.19: Simulation in Gazebo and Visualization in RViz with Nav2

Now, Different Poses Can Be Experimented With

Valid poses and invalid ones can be given:

. Valid Poses: Poses that the robot can reach easily, such as open spaces or clear paths, can be selected.

.Invalid Poses: Poses that cannot be reached, like points blocked by obstacles or narrow areas the robot cannot pass through, can also be selected.

- The robot's behavior can be observed when given both valid and invalid

- This **figure 4.19** shows the simulation environment in **Gazebo** (left), where the TurtleBot3 robot navigates within a space containing green obstacles and cylindrical objects at the center. On the right, **RViz** displays the generated occupancy grid map using the LiDAR sensor, along with the **Nav2 navigation panel** for path planning and trajectory execution. The navigation and localization modules are indicated as active, confirming the proper functioning of the autonomous navigation system

4.9.6 GRAPHICAL REPRSENTATIONS ROS2 with SLAM

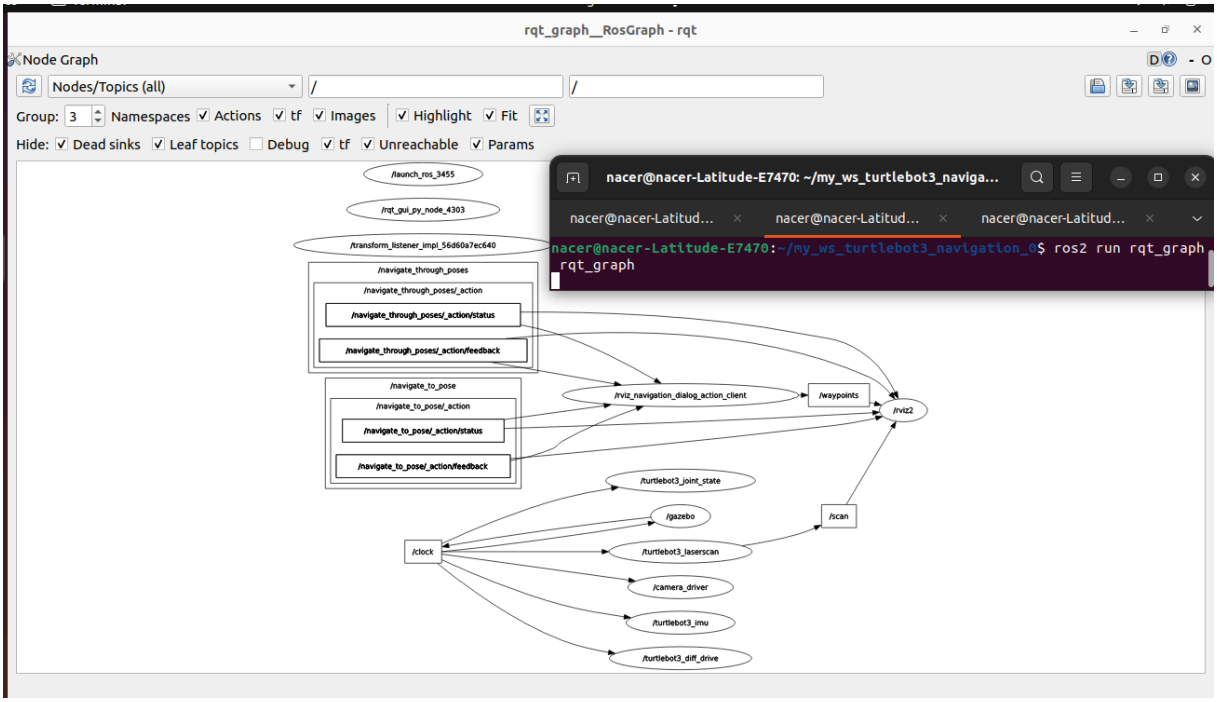


Figure 4.20: ROS 2 Node Graph (rqt_graph) of the TurtleBot3 Navigation System

This **figure 4.20** presents the **node graph** generated by the **rqt_graph** tool, illustrating the interaction between nodes, topics, and actions within the TurtleBot3 navigation system.

- Sensor nodes such as **/turtlebot3_laserscan**, **/turtlebot3_imu**, and **/camera_driver** provide perception data.
- The **/gazebo** node supplies the simulation clock via the **/clock** topic.
- The **/navigate_to_pose** and **/navigate_through_poses** nodes represent the Nav2 action servers for autonomous navigation.
- The **/rviz2** node subscribes to multiple topics for visualization and trajectory monitoring.

4.10 Conclusion

In this chapter, we presented a complete workflow for simulating the TurtleBot3 robot in a ROS 2 Humble environment using the Gazebo simulator. The process began with the installation of the Ubuntu operating system and the ROS 2 framework, followed by the creation of a workspace and the development of simple publisher–subscriber nodes to illustrate the fundamental communication mechanisms in ROS 2. We then introduced the concept of launch files to manage multiple nodes simultaneously, improving the modularity and efficiency of system execution.

Subsequently, the TurtleBot3 robot was simulated in Gazebo, where teleoperation was first used to validate motion control. The Simultaneous Localization and Mapping (SLAM) approach was then employed to generate an occupancy grid map of the environment. Once a map was obtained, the Navigation 2 (Nav2) stack was configured to enable autonomous navigation. Using RViz, the robot's initial pose was estimated, navigation goals were assigned, and successful point-to-point navigation was demonstrated while avoiding obstacles.

Overall, this chapter highlights the importance of simulation tools in robotics development, as they allow the design, testing, and validation of algorithms without the need for physical hardware. The integration of Gazebo, SLAM, and Nav2 within ROS 2 provides a robust framework for exploring advanced robotic functionalities such as mapping, localization, and autonomous navigation. This simulation environment serves as a foundation for future extensions, including the adaptation of custom robots, multi-robot systems, and real-world deployment.

Conclusion

in this thesis, we successfully explored the simulation of an autonomous mobile robot using ROS 2 and Gazebo**,** with the TurtleBot3 serving as a case study. The work highlighted the crucial role of embedded systems, sensors, and software frameworks in enabling robots to perceive their environment, build maps, and navigate autonomously. Through a step-by-step implementation, the study demonstrated the integration of ROS 2 nodes, teleoperation, SLAM-based mapping, and autonomous navigation using the Navigation2 stack.

The results confirmed the effectiveness of ROS 2 and Gazebo as reliable platforms for testing and validating robotic algorithms in a controlled and risk-free environment. These simulations reduce development costs, accelerate prototyping, and provide a foundation for real-world deployment.

Despite the promising outcomes, challenges remain, particularly in improving localization accuracy, reducing uncertainty, and ensuring robustness in highly dynamic or unstructured environments. Future perspectives include integrating advanced sensor technologies, machine learning approaches, and real-world experiments to enhance autonomy and adaptability.

Ultimately, this work contributes to the broader effort of advancing ROS-powered autonomous robotic systems**,** paving the way for their application in industrial automation, service robotics, and beyond

Bibliography

1. Valvano, J. W. (2013). *Embedded Systems: Introduction to Arm® Cortex™-M Microcontrollers*.
2. Ganssle, J. (2008). *The Art of Designing Embedded Systems*.
3. Liu, J. W. S. (2000). *Real-Time Systems*.
4. Forecr.io. (n.d.). *The Embedded Systems*. Retrieved from https://www.forecr.io/blogs/embedded-systems/the-embedded-systems.
5. Craig, J. J. (2004). *Introduction to Robotics: Mechanics and Control* (3rd ed.).
6. Fu, K. S., Gonzalez, R. C., & Lee, C. S. G. (1987). *Robotics: Control, Sensing, Vision, and Intelligence*.
7. Siciliano, B., & Khatib, O. (Eds.). (2008). *Springer Handbook of Robotics*.
8. Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). *Introduction to Autonomous Mobile Robots*.
9. Siegwart et al. (2011). *Introduction to Autonomous Mobile Robots*.
10. Siegwart et al. (2011). *Introduction to Autonomous Mobile Robots*.
11. Pratt, J., & Tedrake, R. (2019). *Legged Robotics*.
12. Floreano, D., & Wood, R. J. (2015). *Science, Technology and the Future of Small Autonomous Drones*.
13. Yuh, J. (2000). *Design and Control of Autonomous Underwater Robots: A Survey*.
14. Murphy, R. R. (2000). *Introduction to AI Robotics*.
15. Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). *Introduction to Autonomous Mobile Robots*. MIT Press.
16. LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.
17. Craig, J. J. (2004). *Introduction to Robotics: Mechanics and Control* (3rd ed.) Pearson Prentice Hall.
18. Valvano, J. W. (2013). *Embedded Systems: Introduction to Arm® Cortex™-M Microcontrollers*. CreateSpace Independent Publishing Platform.
19. Siciliano, B., & Khatib, O. (Eds.). (2016). *Springer Handbook of Robotics* (2nd ed.). Springer.

Bibliography

20. Borenstein, J., Everett, H. R., Feng, L., & Wehe, D. (1997). Mobile Robot Positioning: Sensors and Techniques. *Journal of Robotic Systems*, 14(4), 231–249.
21. Bar-Shalom, Y., Li, X. R., & Kirubarajan, T. (2001). *Estimation with Applications to Tracking and Navigation*. Wiley.
22. Yates, R., & Macdonald, B. (2005). *Embedded Systems for Robotic Applications*. Wiley.
23. Durrant-Whyte, H., & Bailey, T. (2006). Simultaneous Localization and Mapping (SLAM): Part I & II. *IEEE Robotics & Automation Magazine*, 13(2), 99–110.
24. Gautier, M., & Laumond, J. P. (2001). *Robot Motion Planning and Control*. Springer.
25. Zhang, Z. (2012). Microsoft Kinect Sensor and Its Effect. *IEEE Multimedia*, 19(2), 4–10.
26. Merrill, W. (2009). *Introduction to Sensors for Mechatronics*. Elsevier.
27. Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). *Introduction to Autonomous Mobile Robots* (2nd ed.). MIT Press.
28. Kuc, R., & Siegel, M. W. (1987). Physically based simulation model for acoustic sensor robot navigation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(6), 766–778.
29. Everett, H. R. (1995). *Sensors for Mobile Robots: Theory and Application*. A K Peters.
30. Leonard, J. J., & Durrant-Whyte, H. F. (1991). Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, 7(3), 376–382.
31. Zhang, J., & Singh, S. (2014). LOAM: Lidar Odometry and Mapping in Real-time. In *Proceedings of Robotics: Science and Systems* (RSS).
32. Macenski, S., Foote, T., Gerkey, B., Lalancette, C., & Woodall, W. (2022). Robot Operating System 2 (ROS 2): Design, Architecture, and Use Cases. In A. Koubaa (Ed.), *Robot Operating System (ROS): The Complete Reference (Volume 6)* (pp. 47–90). Springer. https://doi.org/10.1007/978-3-030-91104-5_3
33. Maruyama, Y., Kato, S., & Azumi, T. (2016). Exploring the Performance of ROS2. In *Proceedings of the 13th International Conference on Embedded Software* (EMSOFT). ACM. https://doi.org/10.1145/2968478.2968502

Bibliography

34. ROS 2 Documentation — The Robot Operating System 2 Project (Open Robotics). https://docs.ros.org/en/foxy/index.html
35. DDS Foundation. (2020). *Introduction to DDS: Data Distribution Service*. Object Management Group. https://www.dds-foundation.org
36. Quigley, M., Gerkey, B., & Smart, W. D. (2015). *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, Inc.
37. Macenski, S., & Moulard, T. (2021). *Navigation2 Documentation*. Open Source Robotics Foundation. https://navigation.ros.org
38. Robotics Backend. (2021). *ROS2 Engine Adapter Node Architecture*. https://roboticsbackend.com
39. ROS Index. (2024). *TF2 – Transform Library*. https://index.ros.org/p/tf2/
40. ROS Wiki. (2024). *sensor_msgs/CameraInfo*. http://wiki.ros.org/sensor_msgs
41. ROS Wiki. (2024). *AMCL Package*. http://wiki.ros.org/amcl
42. ROS Wiki. (2024). *GMapping*. http://wiki.ros.org/gmapping
43. Google Cartographer Documentation. (2024). https://google-cartographer.readthedocs.io
44. OpenCV Documentation. (2024). *Open Source Computer Vision Library*. https://docs.opencv.org
45. Gazebo Documentation. (2024). *Gazebo Robot Simulator*. https://gazebosim.org
46. RViz Documentation. (2024). *RViz – 3D Visualization Tool for ROS*. https://wiki.ros.org/rviz
47. TurtleBot3 Documentation. (2024). *TurtleBot3 Simulation and Navigation*. ROBOTIS. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview