Université Mohamed Khider de Biskra
Faculté des Sciences et de la Technologie
Département de génie électrique

MÉMOIRE DE MASTER

Sciences et Technologies
Génie électrique
Systèmes embarqués

Réf. : MEM/M2/ESE/UNIV-BISKRA/2025

Présenté et soutenu par :
ZEGHIDI MOUSTAFA et REHAB MOHAMED AMINE

Le : Click here to enter a date.

Behavior Trees for mobile robot simulation in ROS2

Jury :

Mme.	Athamna Noura	MCB	Université de Biskra	Présidente
M.	BEKHOUCHE Khaled	MCA	Université de Biskra	Encadreur
M.	BENELMIR Okba	MCB	Université de Biskra	Examinateur

Année universitaire : 2024 – 2025

Université Mohamed Khider de Biskra
Faculté des Sciences et de la Technologie
Département de génie électrique

MÉMOIRE DE MASTER

Sciences et Technologies
Génie électrique
Systèmes embarqués

Réf. : MEM/M2/ESE/UNIV-BISKRA/2025

Behavior Trees for mobile robot simulation in ROS2

Présenté par :

Avis favorable
de l'encadreur :

Dr.BEKHOUCHE KHALED

ZEGHIDI MOUSTAFA
REHAB MOHAMED AMINE

Signature Avis favorable du Président du Jury

Dr. Athamna Noura

Cachet et signature

General Summary of the Thesis

This thesis focuses on the design and simulation of a mobile robot using the Robot Operating System ROS2 (Humble), with an emphasis on the use of Behavior Trees to organize and enhance the robot's behavior. The study begins by defining mobile robots, their types, and their mechanical and electronic components such as sensors and motors. It then covers the core concepts of ROS2, including packages, nodes, topics, and services, along with the use of simulation tools like RViz and Gazebo.

The thesis also includes the steps for setting up the development environment (Ubuntu, ROS2, Gazebo, Blender), creating software packages using C++ and Python, and designing robot models using URDF and SDF files. The work concludes with simulating the robot's behavior using Behavior Trees, demonstrating their ability to manage tasks in a flexible and efficient way.

ملخص عام للمذكرة

:تتناول هذه المذكرة تصميم ومحاكاة روبوت متنقل باستخدام نظام تشغيل الروبوتات(ROS2 (Humble ، مع التركيز على استخدام أشجار السلوك (Behavior Trees) لتنظيم وتحسين سلوك الروبوت. تبدأ الدراسة بتعريف الروبوتات المتنقلة وأنواعها، ومكوناتها الميكانيكية والإلكترونية من حساسات ومحركات. ثم يتم التطرق إلى مكونات ROS2 مثل الحزم، والعُقد، والمواضيع، والخدمات، مع استخدام أدوات المحاكاة مثل RViz و .Gazebo

كما تشمل المذكرة خطوات إعداد بيئة التطوير(Blender، Gazebo، ROS2، Ubuntu) وإنشاء الحزم البرمجية باستخدام C++ وPythonملفات، وتصميم نماذج الروبوت باستخدام ملفات URDF وSDF. تُختتم الدراسة بمحاكاة سلوك الروبوت باستخدام أشجار السلوك، لإظهار قدرتها على إدارة المهام بطريقة مرنة وفعّال

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to our supervisor,

Mr. Khaled Bekhouche, whose guidance, support, and valuable advice played a major and pivotal role in the completion of this project. His continuous encouragement and mentorship have been truly inspiring throughout our journey.

We also extend our deep appreciation to all the professors and staff of Mohamed Khider University of Biskra for their dedication and unwavering support, which have greatly contributed to our academic growth.

Our heartfelt thanks go as well to our fellow classmates, whose companionship, collaboration, and positive spirit have enriched our experience and fostered a motivating and supportive environment during this project

Finally, we would like to thank everyone who encouraged and supported us in bringing this work to completion. Your contributions, in every form, are deeply appreciated.

Dedication

Dedication Moustafa and Mohamed

To my father

my pillar, my role model, my companion, and my everything. Your unwavering support through every moment of my life has been my greatest strength.

To my mother

the heartbeat of my soul and the companion of my journey. Your love is the light that guides me.

To my sisters

the garden of love and smiles in my heart. Your presence brings warmth and joy into my life.

To my friends

who have always stood by my side. Your encouragement means the world to me.

And a special greeting to my little nephew

the youngest joy of our family. Your innocence brightens our days.

Table of contents

List of figures

List of figures

List of tables

General Introduction

General Introduction

Mobile robots have experienced rapid advancements in the past decade, driven by breakthroughs in artificial intelligence, control systems, and simulation technologies. These robots play a pivotal role in diverse applications, from logistics and hazardous environment exploration to healthcare and smart homes. However, designing a mobile robot capable of autonomous operation in dynamic environments demands a sophisticated integration of mechanical engineering, electronics, and software algorithms. This underscores the importance of frameworks like ROS2 (Robot Operating System) and simulation tools such as Gazebo, which enable the testing and implementation of complex behaviors in virtual environments before real-world deployment, significantly reducing costs and risks.

In this context, this thesis focuses on exploring mobile robot behaviors using Behavior Trees within the ROS2 Humble ecosystem, leveraging simulation as the primary experimental platform. Behavior Trees offer a flexible and hierarchical methodology for decision-making in robotics, allowing tasks to be organized modularly and adaptively. This makes them ideal for addressing dynamic challenges in real-world scenarios, such as obstacle avoidance, path planning, and multi-task coordination.

The research is structured into three key pillars:

chapter 1:
Theoretical and Practical Foundations of Mobile Robots: Covering basic definitions (e.g., autonomy, degrees of freedom), mechanical design, control systems (kinematics and dynamics), and critical components like sensors and actuators.
chapter 2:

The ROS2 Framework: Delving into its core concepts, including packages, nodes, topics, services, actions, and visualization tools like RVIZ and Gazebo.

chapter 3 Full-Scale Mobile Robot Simulation: From setting up the software environment (Ubuntu 22.04, ROS2 Humble, Gazebo, Blender) to designing the robot model using URDF/SDF files, and implementing autonomous behaviors via Behavior Trees in Python and C++.

By bridging theory and practice, this study aims to establish a comprehensive framework for designing and executing complex mobile robot behaviors. It highlights the power of open-source tools like ROS2 for efficient simulation and prototyping while addressing technical challenges such as model accuracy, real-time performance, and behavior tree optimization.

Ultimately, this work contributes to advancing autonomous robotics by demonstrating scalable, adaptable solutions for real-world applications.

Chapter 1

Mobile robots

Intoduction :

This chapter provides a foundational introduction to the world of mobile robots, presenting the key concepts that form the basis of this dynamic and evolving field. It begins with the definition of a robot and an autonomous robot, highlighting the difference between traditional systems and those capable of making decisions and performing tasks without direct human intervention. The concept of Degrees of Freedom (DOF) is also discussed, which defines the robot's ability to move in space.

The chapter then explores various types of mobile robots, with a particular focus on wheeled robots, due to their widespread use and practical applications. It also covers the fundamental principles of mechanical design, including the structural and motion systems that enable efficient mobility.

Moreover, the chapter addresses key analytical aspects such as robot kinematics and dynamics, helping to understand the robot's motion and interaction with its environment. Finally, it presents an overview of the essential components of mobile robots, from the motors that generate motion, to the sensors that allow perception of the surroundings and enable the robot to make informed decisions accordingly.

1.1 definitions of Terms

1.1.1 definition robot : The multi-dimensionality definition of the robot is possible. In simple words, the machine itself helps humans in work and is used as a human substitute in various tasks, they are the robot. According to the IEEE definition,

"A robot is an autonomous machine capable of sensing its environment, carrying out computations to make decisions and performing actions in the real world.

" According to the definition given by the Robot Institute of America, "Any machine made by one of our members" and the definition given by the Robot Institute of America in 1979, "A robot is a reprogrammable, multifunctional manipulator designed to move material, parts, tools or specialized devices through variable programmed motions for the performance of a variety of tasks. [1]

" The robot displays all or part of the following religions:

Not natural, artificial.

- Has the ability to sense the environment.
- Can work with environmental objects.
- There is some intelligence, with which decisions can be made by understanding the environment. Programmable by computer.
- Can move and transfer.
- Can demonstrate well-controlled movements efficiently.
- Can volunteer and give glimpses.

1.1.2 Type of robots :

1.1.2.1 Industrial robots:

An industrial robot is one that has been developed to automate intensive production tasks such as those required by a constantly moving assembly line. As large, heavy robots, they are placed in fixed positions within an industrial plant and all other worker tasks and processes revolve around them.

The characteristics of industrial robots will vary according to the manufacturers, the needs and the scenario in which they are to be located.

According to the international standard ISO 8373:2012, the industrial robot definition is 'a multifunctional, reprogrammable, automatically controlled manipulator, programmable in three or more axes that can be fixed in one area or mobile for use in industrial automation applications'. Industrial robots are not usually humanoid in form, although they are capable of reproducing human movements and behaviours but with the strength, precision and speed of a machine. [2]

Figure 1-1-1 One of the types of Industrial robots

1.1.2.2 Sevice robots:

A service robot is a robot which operates semi or fully autonomously to perform services useful to the well-being of humans and equipment, excluding manufacturing operations .

With this definition, an industrial robot, e.g., a robotic arm, could easily be classified as a service robot if it is installed in a non-industrial environment to perform non-manufacturing task. ForInstance [3]

Figure 1-1-2 One of the types of Sevice robots

1.1.2.3 Military robots:

Are autonomous or remote-controlled devices designed for military applications.

Such systems are currently being researched by a number of militaries. Already remarkable success has been achieved with unmanned aerial vehicles like the Predator drone, which are capable of taking surveillance photographs, and even accurately launching missiles at ground targets, without a pilot. A subclass of these are Unmanned Combat Air Vehicles, which are designed to carry out strike missions in combat. [4]

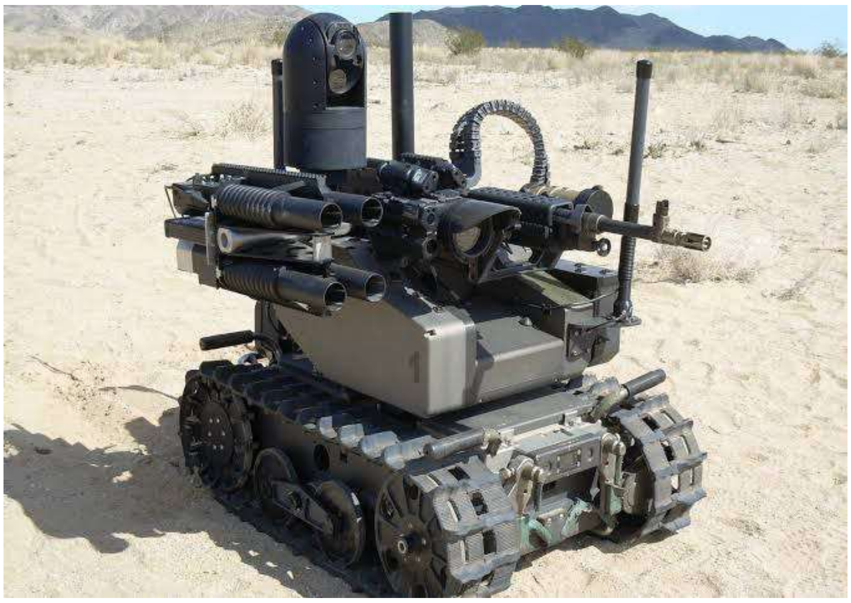


Figure 1-1-3 One of the types of Military robots

1.1.2.4 Medical robots:

Medical robots are specialized robotic systems that can aid healthcare professionals in diagnosing, treating, and managing ailments. These robots can be used for minimally invasive surgery, rehabilitation therapy, drug delivery, patient monitoring, and logistics support in hospitals. They improve accuracy, and minimize human error, and allow for procedures that would be otherwise challenging or impractical to achieve without manual methods. [5]

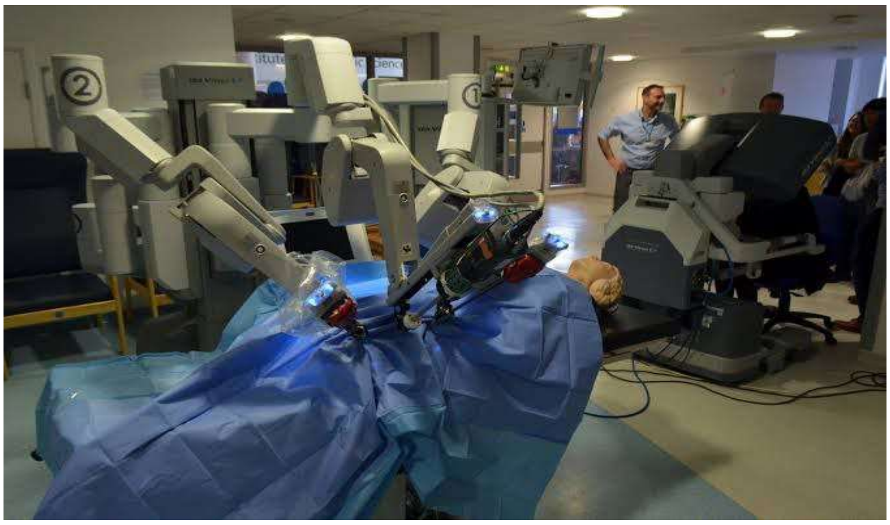


Figure 1-1-4 One of the types of Medical robots

1.1.3 Definition autonomous :

An autonomous robot is an intelligent system that is equipped with sensors or cameras to detect objects and can make decisions to move based on the information it receives without needing to be controlled remotely. This class of robot employs artificial intelligence algorithms and processes data in real-time, allowing them to traverse their environment, avoid barriers and perform in a highly efficient and accurate manner. [6]

There are different types of autonomous robots but we will focus on Autonomous Mobile Robots (AMRS)

1.1.3.1 Autonomous Mobile Robots (AMRS):

Autonomous Mobile Robots (AMRs) are intelligent robotic systems that utilize probabilistic algorithms, such as Simultaneous Localization and Mapping (SLAM), to dynamically navigate unstructured environments and adapt to real-time changes without human intervention (Thrun et al., 2005). [7]

1.1.3.2 Autonomous Mobile robot types :

When developing an autonomous mobile robot, the process often revolves around defining the robot's role and purpose within its environment. While AMRs can take on countless forms depending on their application, three main categories are frequently encountered:

- Legged AMRs
- Wheeled AMRs
- Aerial AMRs

1.1.3.2.1 legged AMRs :

Are robots designed with articulated legs for locomotion, enabling movement across uneven or complex terrains where wheeled/tracked systems fail. They mimic biological leg structures (e.g., bipedal, quadrupedal) and use advanced control algorithms for balance, gait planning, and obstacle avoidance. Applications include search-and-rescue, exploration, and industrial inspections in unstructured environments. Unlike wheeled AMRs, legged systems excel in adaptability but face challenges like higher energy consumption and mechanical complexity. [8]

As shown in the following figure

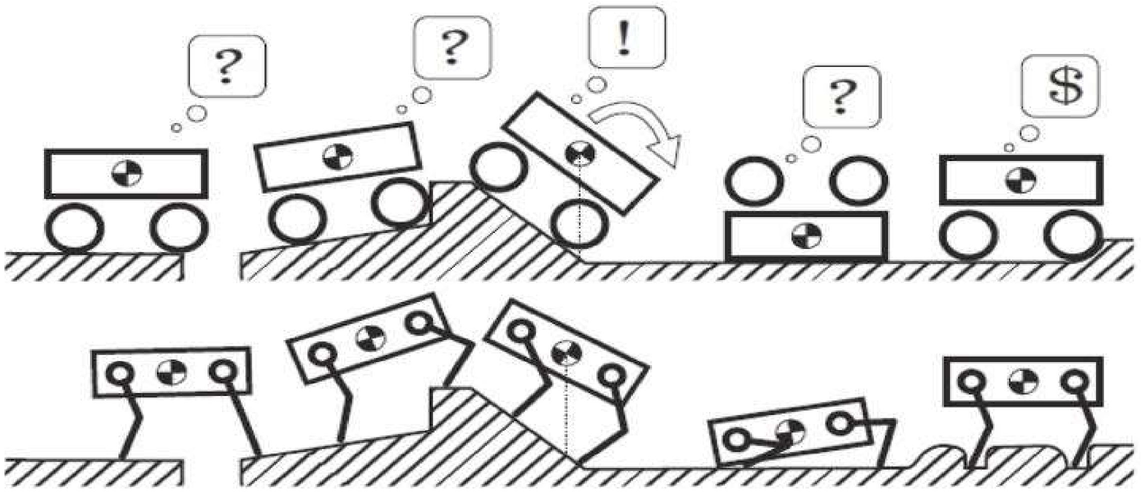


Figure 1.1.2.1 How legged AMRs handle rough terrain

1.1.3.2.2 wheeled AMRs :

Wheels remain the dominant locomotion choice in mobile robotics and human-engineered vehicles due to their mechanical simplicity and high energy efficiency. Unlike legged systems, wheeled designs typically avoid complex balance challenges by maintaining continuous ground contact across all wheels. Three-wheeled configurations inherently ensure stability, though even two-wheeled robots can achieve equilibrium through careful engineering. When employing more than three wheels, suspension mechanisms become essential to adapt to uneven surfaces while preserving traction. Research in wheeled robotics prioritizes optimizing ground adhesion and directional stability over balance concerns. Key challenges include enhancing maneuverability in constrained spaces and improving control systems for diverse terrains. Multi-wheel designs often incorporate adaptive load distribution to prevent slippage during acceleration or climbing. The mechanical advantage of wheels enables efficient power transfer from motors to movement with minimal energy loss. Current innovations focus on developing omnidirectional wheels and advanced traction control algorithms. This ongoing refinement aims to expand wheeled robots' operational capabilities across industrial, agricultural, and exploratory applications.

Figure 1.1.2.2 A picture showing wheeled AMRs.

1.1.3.2.3 Aerial AMRs :

A drone, technically referred to as an Unmanned Aerial Vehicle (UAV) or Unmanned Aircraft System (UAS), is an autonomous or remotely piloted aircraft that operates without an onboard human crew. Functionally, it represents a flying robotic system capable of executing preprogrammed flight plans through integrated software embedded within its onboard systems. These systems synergize with built-in sensors and Global Positioning System (GPS) technology to enable precise navigation, real-time data processing, and mission-specific task execution. Drones may be controlled remotely via ground-based operators or operate autonomously by relying on algorithmic guidance and environmental feedback mechanisms. [9]

Figure 1.1.2.1 (UAV) dron with camera

1.1.4 definition degree of freedom (DOF) :

Degrees of Freedom (DOF) in Robotics The number of independent movements a robot's body or end-effector is capable of making in 3D space. There exists 6 degrees of freedom (DOF), where each DOF is either a translational (moving in a linear way following x, y, z axes) or rotational (moving angular way following roll, pitch, yaw axes). For instance, a 5-DOF robotic arm pretends human arm dexterity and can attain precise positional and orientational values for the processes like welding or assembly.

Mobile robots, such as drones, have 6 DOF for maneuvering in the air, though their propulsion may limit their maneuverability. 309 issues and optimize trajectories; as a simple consequence, they are able to accommodate and manipulate objects with varying geometries. [10]

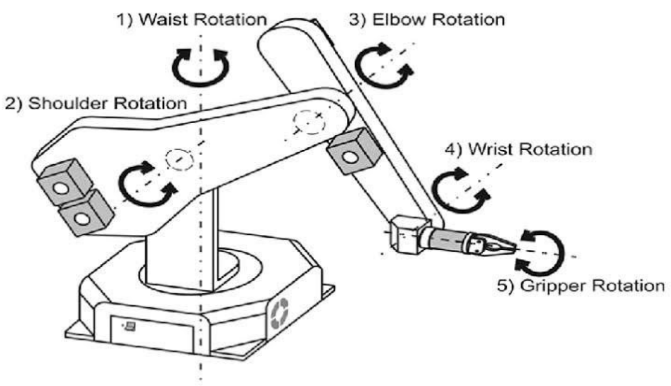


Figure 1-1-3-1 Five degrees of freedom robot arm

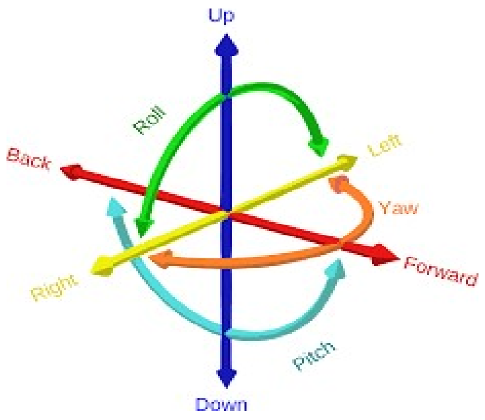


Figure 1-1-3-2 Six degrees of freedom

1.2 Wheeled robots

Introduction

Movement has always been one of the fundamental challenges faced in the development of robots. Throughout history, we have drawn inspiration from nature to design mechanisms that enable smooth and efficient locomotion. In the natural world, we find many examples of intelligent solutions for movement, such as the circular motion used by various creatures, from insects to large mammals. As a result, scientists and engineers began studying and applying these mechanisms to the design of mobile robots.

By understanding how living organisms move, such as insects that use multiple legs or mammals that rely on limbs or wheels, we started developing robots that use wheels as the primary means of movement. Wheels, one of humanity's oldest inventions, became a key inspiration in the development of mobile robots. These wheels enhanced movement stability and speed, allowing robots to navigate different environments with minimal effort.

Over time, the movement technologies in robots have rapidly advanced. Starting from simple two-wheeled robots to four-wheeled robots and tracked systems (Tracked Robots), we now have the ability to design robots capable of handling diverse environments and various movement strategies. This development in robot design heavily relies on a deep scientific understanding of engineering and physics principles, which are applied to movement systems to ensure efficiency and stability when navigating through various terrains.

1.2.1 Two wheeled robot :

A two-wheeled robot works a lot like someone balancing on a bicycle—it stays upright on just two wheels by constantly making small adjustments. It uses sensors like gyroscopes and accelerometers to sense how it's tilted, and smart control systems to quickly correct its position. This helps it stay balanced, move around, turn corners, and even handle bumps or pushes.

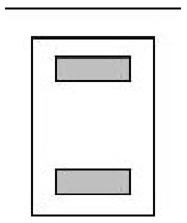


Figure 1.2.1 Two-wheel differential drive with the center of mass (COM) below the axle

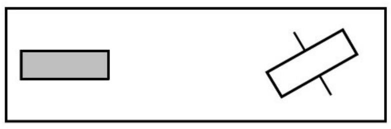


Figure 1.2.1.1 One steering wheel in the front, one traction wheel in the rear

1.2.2 Three wheeled robot :

1.2.2.1 Omnidirectional Three-Wheeled robot :

The three-wheeled robot is a mobile robot that uses a special wheel configuration, allowing it to move in any direction. Each wheel has a radius of 24 mm, and its rotational speed is controlled by a microcontroller through a specific output port.

In this design, there's a clear relationship between the direction of wheel rotation and its angular velocity, which helps calculate the exact speeds needed for specific movements. For example, to move the robot along the Y-axis, wheels 1 and 2 spin at the same speed but in opposite directions, while wheel 3 remains stationary. This setup generates the desired motion.

Overall, the robot's velocity is calculated by adding the velocity vectors of all three wheels, giving a precise result for both direction and speed of movement. [11]

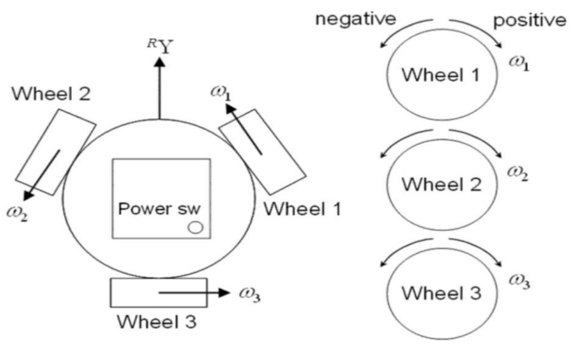


Figure 1.2.2 architecture of three- wheeled robot

1.2.3 Four wheeled robot :

The Four Wheeled Robot is a type of robot that moves using four wheels powered by motors. Here's a simplified explanation [12]:

Advantages of the Four Wheeled Robot:

- **Strong and stable:** Because it has four wheels, it offers better stability during movement.
- **Ideal for uneven surfaces:** It is preferred when you want the robot to move on rough or uneven terrain.
- **Can carry objects:** It has a wide platform that can be used to place small objects.
- **Consumes more power:** Since it is heavier and uses four motors, the battery life is shorter compared to a two-wheeled robot.

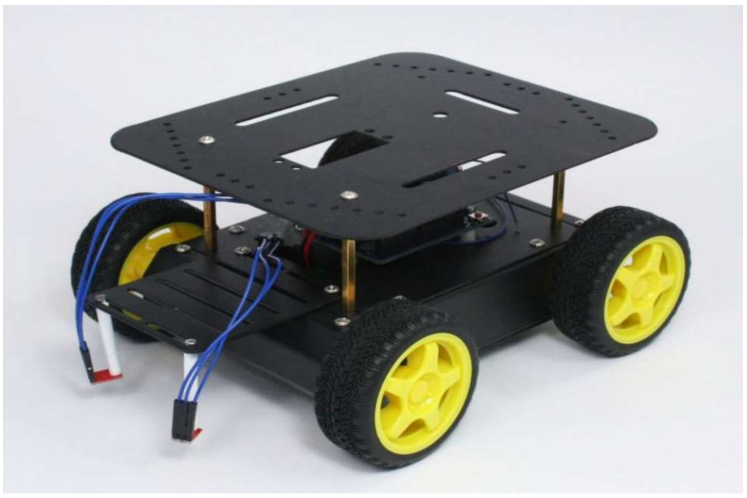


Figure 1.2.3 four-wheeled robot mobile

1.2.4 Tracked robots:

Tracked robots are mobile robots that use continuous tracks instead of wheels, providing excellent capability for climbing obstacles and adapting to rough terrains.

In the following figure. We explain the Polymorphic Tracked Robot as an advanced model that can change the position of its front idler vertically. This allows it to pass through low gaps and climb over high obstacles while maintaining proper track tension through a smart mechanical design. [13]

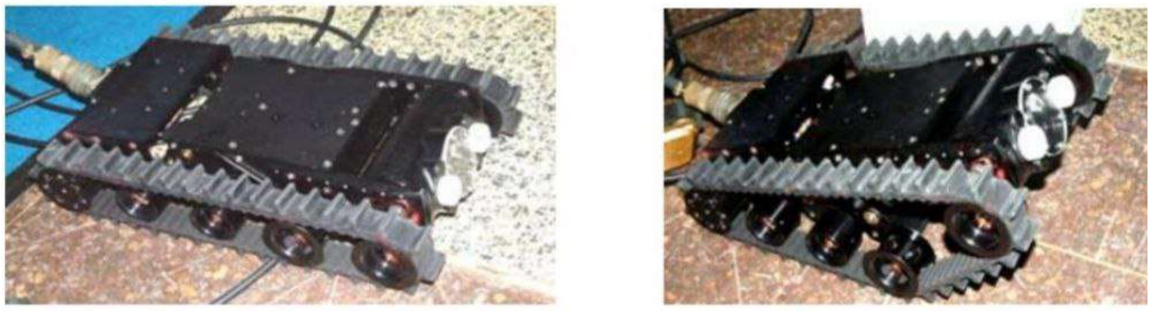


Figure 1.2.4 The Polymorphic Tracked Robot

1.2.5 Comparison Between Different Robot Types:

Table 1 the different between robot types

Robot Type	Stability	Maneuverability	Terrain Adaptability	Complexity	Speed	Cost
Two-Wheeled Robot	Low (needs balancing)	High (can rotate in place)	Low (suitable for flat surfaces)	Moderate (requires control)	High	Low
Three-Wheeled Robot	Moderate	Moderate	Low to Moderate	Simple	Moderate	Low
Four-Wheeled Robot	High	Moderate	Moderate (limited on rough terrain)	Simple to Moderate	High	Moderate
Tracked Robot	Very High	Low (wide turning radius)	Very High (excellent off-road)	High (mechanical design)	Low to Moderate	High

1.3 Mechanical Design :

1.3.1 Locomotion Systems :

There are various methods of moving across solid surfaces, from crawling to hopping. For robots, the three most common systems are wheels**,** tracks, and legs**.** [14]

Features and Limitations:

1.3.1.1 Wheeled Systems:

Advantages:

- Mechanically simple and easy to manufacture.

- Favorable payload-to-weight ratio.

- Adaptable (e.g., modified cars for robots).

Disadvantages:

- Poor performance on uneven terrain.

- Difficulty overcoming obstacles close to the wheel’s radius.

- Large wheels as a solution may be impractical.

1.3.1.2 Tracked Systems :

Advantages:

- Better at negotiating large obstacles.

- Less affected by environmental hazards (e.g., loose soil).

- **Disadvantages**:

- Energy loss due to track friction.

- Dead-reckoning errors during turns (tracks slip).

- Example: NASA’s **Sojourner** rover used pivoting wheels to climb Martian rocks (Figure 1.3.1).

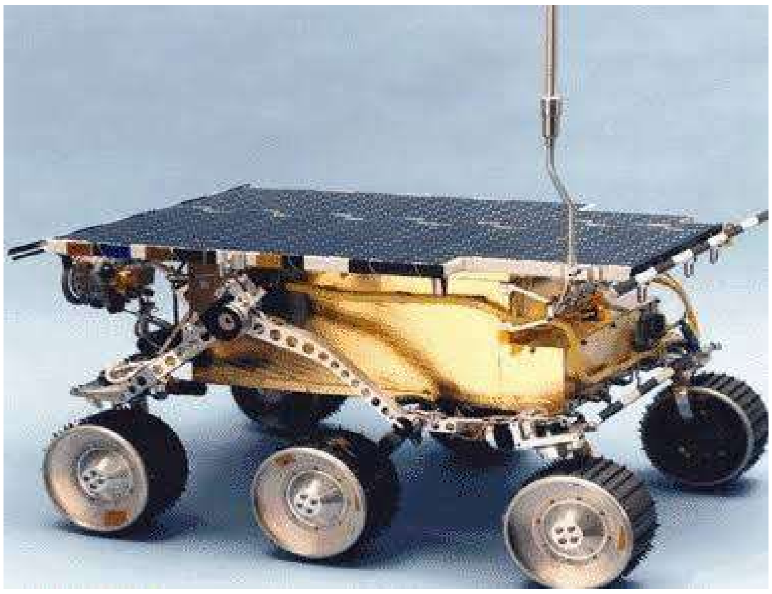


Figure 1.3.1 NASA's Sojourner

2.7.2.4 Legged Systems :

- **Advantages**:

- Ideal for rugged terrain.

- **Disadvantages**:

- High mechanical complexity.

- Higher manufacturing costs.

- Greater susceptibility to failure.

- Example: **Genghis** robot with six servo-controlled legs (Figure 1.3.2).

Figure 1.3.2 Genghis robot

1.3.2 Wheel Design:

Wheeled systems can be optimized through configurations like:

1.3.2.1 Synchro Drive Mechanism:

The synchro drive is a robotic system where all wheels (typically three) function both as steering and driving units. These wheels are mechanically synchronized to always face the same direction. To alter the robot's movement path, all wheels pivot simultaneously around a vertical axis, enabling directional changes while the chassis maintains its original orientation. This design avoids the need for the body to rotate when turning.

However, if the robot requires a designated front section (e.g., for sensor placement), extra mechanical linkages must integrate the chassis's orientation with the wheels' direction, ensuring the body aligns with the movement. While this mechanism addresses limitations seen in differential, tricycle, or car-type drives—such as maneuverability and precision—it introduces higher mechanical complexity. The trade-off thus lies in enhanced operational flexibility versus more intricate engineering. [14]

1.3.2.2 Tricycle Steering:

Car-type drive (Ackerman steering) and tricycle drive both offer good stability due to their suspension setups. The car drive has four suspension points, while the tricycle drive is mechanically simpler, as it doesn't require a linkage between two steerable wheels like the car does. In both systems, the fixed wheels are usually connected to the drive motor, while the steerable wheels are not driven. However, in some robots, the steerable wheels can also be powered. A key advantage of these systems is that the robot can move in a straight line without needing to monitor wheel velocity—positioning the steerable wheel in its neutral position is enough. This simplicity, however, comes with certain trade-offs, which will be discussed in the next section. [14]

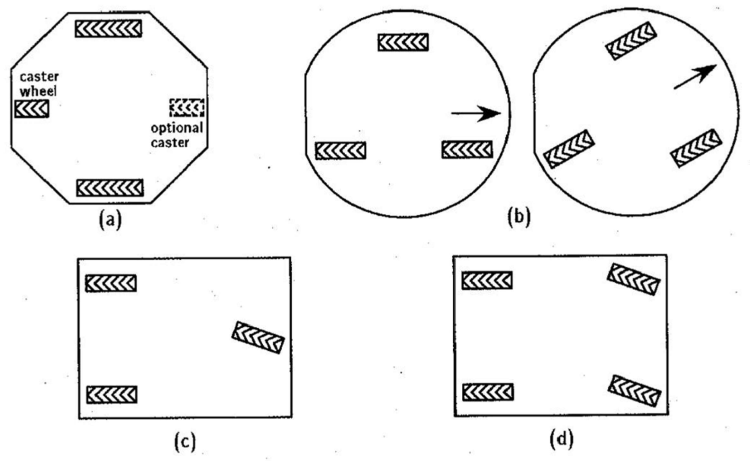


Figure 1.3.3 (a) Differential drive (b) Synchro drive (c) Tricycle drive (d) Car-type drive

1.3.3 Legged Locomotion:

Legged systems are complex but can be simplified using:

- **Model Airplane Servos**:

For insect-like legs (Figure 6.12).

- **Stepping Mechanism**:

1. Swing leg outward to clear obstacles.
2. Rotate leg forward.
3. Lower leg to ground contact.
4. Push body forward via synchronized motion of six legs.

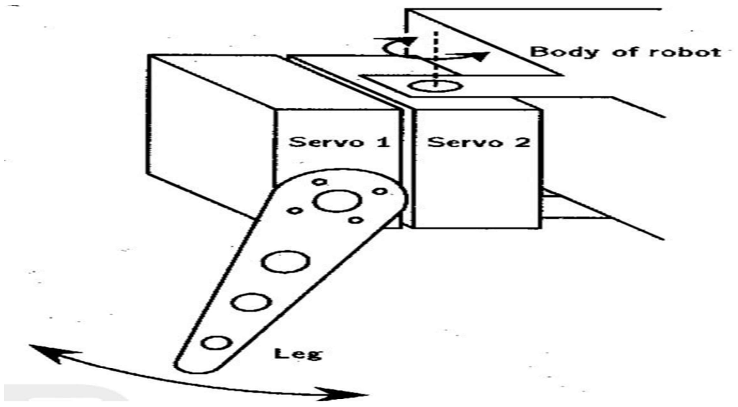


Figure 1.3.4 Model Airplane Servos

1.4 Robot Kinematics and Dynamics:

1.4.1 kinematics :

Kinematics forms the foundational framework for analyzing the motion of mobile robots, addressing their workspace constraints, controllability, and interaction with dynamic environments. Unlike manipulators—whose fixed-base architecture allows direct end-effector localization via joint encoders—mobile robots lack an inertial reference, necessitating pose estimation through incremental odometric integration. This process, however, is inherently susceptible to slippage-induced inaccuracies, complicating precise localization. Mobile robot kinematics models derive from the synthesis of individual wheel contributions, each governed by non-holonomic constraints (e.g., prohibition of lateral slip in differential-drive systems). These constraints are formalized within global (world) and local (robot-centric) coordinate systems, enabling the construction of forward kinematic models that aggregate wheel velocities to predict whole-body motion. Controllability is further shaped by dynamic factors, such as center-of-gravity effects on stability during high-speed maneuvers. The interplay of kinematic constraints (e.g., minimum turning radius) and dynamic limitations defines the robot's maneuverability envelope, informing feasible trajectories and design trade-offs between agility, stability, and actuation efficiency. Synthesizing these principles permits systematic evaluation of navigation strategies, emphasizing the interdependence of mechanical design, sensor-based localization, and motion planning in unstructured environments [15]

1.4.2 kinematic models and constraints :

Kinematic modeling of mobile robots employs a bottom-up approach, deriving system dynamics from individual wheel contributions and constraints. Each wheel influences motion while imposing kinematic limitations, which are integrated via chassis geometry to form holistic constraints. Precise reference frames—global and local—are critical to map self-contained mobility, ensuring consistent spatial representation. The methodology formalizes these frames to analyze wheel kinematics and synthesize robot-level behavior. Structured annotation aligns with established terminology, emphasizing systematic integration of components. This process ensures accurate, coherent models by addressing interdependent mechanical and geometric factors.

1.4.2.1 Represeting robot position

The robot is modeled as a rigid body on wheels moving in a horizontal plane, with its chassis having three degrees of freedom: **x** and **y** positions in the plane and **θ** orientation (rotation about the vertical axis). Internal degrees of freedom (e.g., wheel steering joints) are ignored, focusing solely on the chassis as a single rigid structure. To define the robot's **pose** (position and orientation), a global reference frame $\{X_I, Y_I\}$ (fixed inertial frame) and a local frame $\{XR, YR\}$ (attached to the robot's reference point **P**) are used. [15]

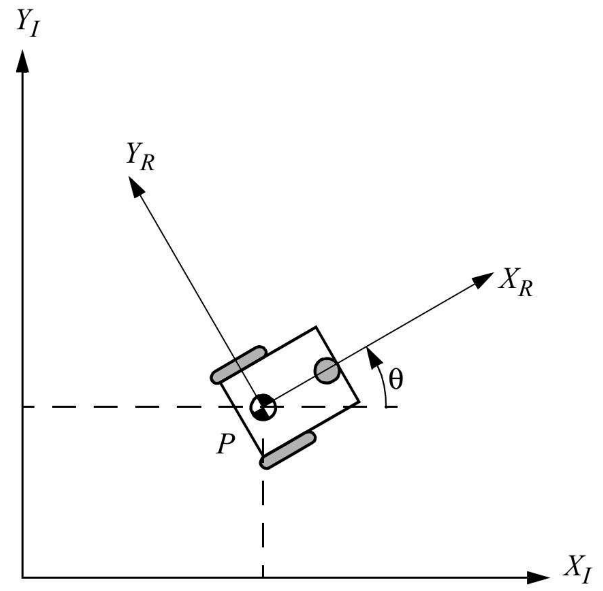


Figure 1.4.1 robot position x and y positions in the plane and θ orientation

The robot's pose is represented as a vector *ξ* **= [x, y, θ]^T**, where **x** and **y** are the coordinates of **P** in the global frame, and **θ** is the angular difference between the global and local frames. This abstraction simplifies analysis by treating the chassis as a single rigid entity, independent of its internal mechanics. [15]

$$\varepsilon = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \qquad (1.1)$$

To decompose a robot's movement into its constituent parts, one must transform the motion relative to the global coordinate system into the robot's own local frame. This conversion hinges critically on the robot's instantaneous orientation and position (its pose). Such a transformation is mathematically achieved via an orthogonal rotation matrix, which preserves geometric properties like distances and angles. The matrix dynamically adjusts based on the robot's orientation—such as its yaw, pitch, or roll—to project global-axis motions onto the robot's local axes. This ensures accurate representation of velocity, acceleration, or displacement in the robot's evolving frame of reference.

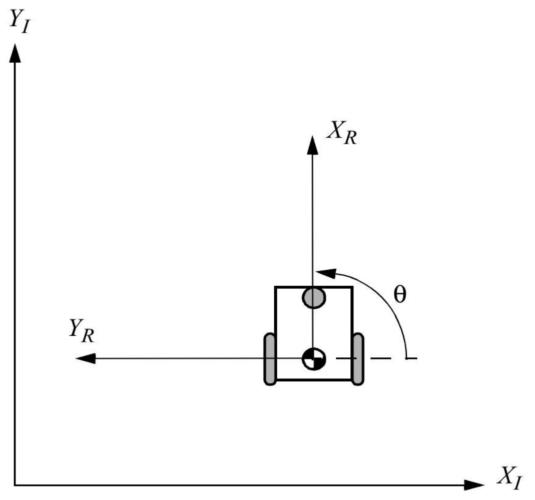


Figure 1.4.2 Pose of a mobile relative to globle and local frame

The mobile robot aligned with aaxis

$$R(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1.2)$$

Where the first and second columns correspond to motion in the global reference frame {X1, Y₁}, and the motion (in the global frame) are in terms of the local reference frame {XR, YR}. This operation is expressed by $R\ (\theta)\ \varepsilon 1$ because the output of this operation depends on the input of only θ:

$$\varepsilon_{R=}R(\frac{\pi}{2})\varepsilon_1 \quad (1.3)$$

For example, consider the robot in figure 1.4.2. For this robot, because $\theta = \frac{\pi}{2}$ we can easily compute the instantaneous rotation matrix R:

$$R\left(\frac{\pi}{2}\right) = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1.4)$$

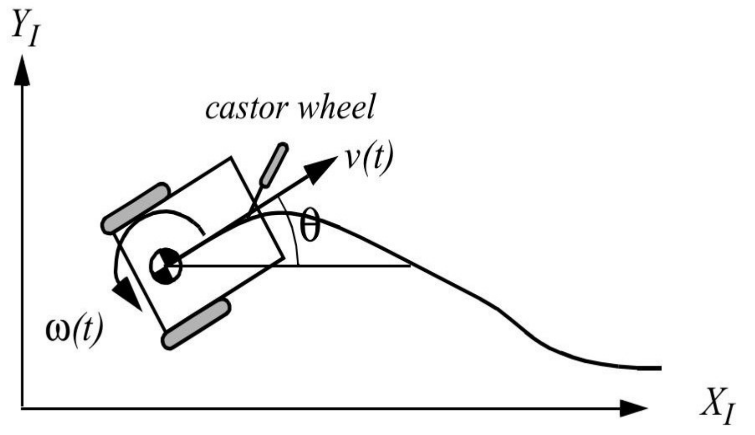


Figure 1.4.3 A differential –drive robot in global refernce frame

Given some velocity $(\dot{x}, \dot{y}, \dot{\theta})$ in the global reference frame we can compute the components of motion along this robot's local axes X_R and Y_RIn this case, due to the spe-cific angle of the robot, motion along X_R is equal to y and motion along Y_R is $-\dot{x}$:

$$\dot{\varepsilon_R} = R\left(\frac{\pi}{2}\right)\dot{\varepsilon_I} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{y} \\ -\dot{x} \\ \dot{\theta} \end{bmatrix} \tag{1.5}$$

1.4.3 Dynamics :

Robot dynamics are the relationship between the forces acting on a robot and the resulting motion of the robot

The dynamics of a robot manipulator can be obtained through the Euler-Lagrange methodology. The kinetic energy of a robot manipulator is

$$K(q.\dot{q}) = \frac{1}{2}\dot{q}^{\,T} M(q)\dot{q} = \frac{1}{2}\dot{q}(\sum_{i=1}^{n}[m_i J_U^T(q)J_U + J_\omega^T(q)R_i(q)l_i R_i^T(q)j_\omega(q)])\dot{q} \tag{2.1}$$

where mi is the mass of link I, Jυi and Jωi are the linear and angular Jacobian of link I respect to the other links, respectively, □I is the inertia tensor of the link I, and Ri (q) is the orientation matrix of each link with respect to the global inertial frame. M(q) ∈ ℝn×n is the inertia matrix, which is symmetric and positive definite for all q ∈ ℝn [16]

1.5 Mobile robot sensors :

Mobile robots rely on sensors for operating as well as autonomy. A sensor is a device that converts a physical parameter (or environment characteristic like temperature, distance, and speed) and creates a signal to be measured and processed digitally. Analyzing the navigation feature of mobile robots, sensors are crucial for understanding the environment around them. By allowing a robot to navigate safely, make complex decisions based upon sensory inputs, communicate with other agents, and adjust to changing conditions. Because robotics requires sensors to function autonomously without needing human intervention and react to events happening in their environment, sensors are essential spears of robotic systems. [17]

type of mobile robot sensors :

There are many types of sensors used in mobile robots; however, in this section, we will focus on the most important ones**.**

1.5.1 Proximity sensors

- **Ultrasonic sensors :**

  Ultrasonics is one of the sensing technologies that can be used by a mobile sensor robot to perceive and interact with its surroundings. Ultrasonics means vibrations that have frequencies above the upper audible limit of human hearing—generally greater than 20 kilohertz. Mobile robots often use such high-frequency waves for vehicle and orientation detection. This term is used when the amplitude of these waves is extremely high. At even higher frequencies, more than 10^{13} hertz, the term hypersound (or praetersound or microsound) is used. At such high frequencies though, sound waves can no longer move effectively. For that matter, longitudinal waves cannot propagate so much as in solids or liquids, either, at frequencies greater than roughly 1.25×10^{13} hertz, because the molecular architecture of the medium is incapable of carrying [18]

Figure 1.5.1 Ultrasonic sensors

- **Infred (IR) sensors :**

  An Infrared (IR) sensor is an electronic device that uses light to detect nearby objects. This kind of sensor is used to measure the heat emitted by an object and motion detection. Every object gives off thermal energy in the infrared spectrum. This radiation is not visible to our eyes, however, an IR sensor can detect it. It is also used in most mobile robots for object detection and motion tracking, and to aid smart [19]

The (figure-1-5-2) shows more of the sensor technology that is widely used in Arduino

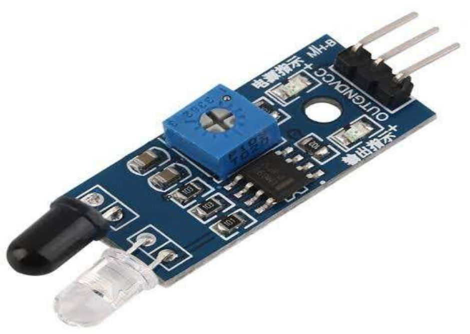


Figure 1.5.2 Infred (IR) sensors

1.5.2 Lidar :

LiDAR (Light Detection and Ranging) is an active mobile sensor used in robots and automated vehicles to perceive their environment. It emits laser pulses and measures their reflections to generate precise 3D spatial data. This enables real-time obstacle detection, mapping (SLAM), localization, and motion estimation, critical for navigation and autonomy in dynamic settings**.**

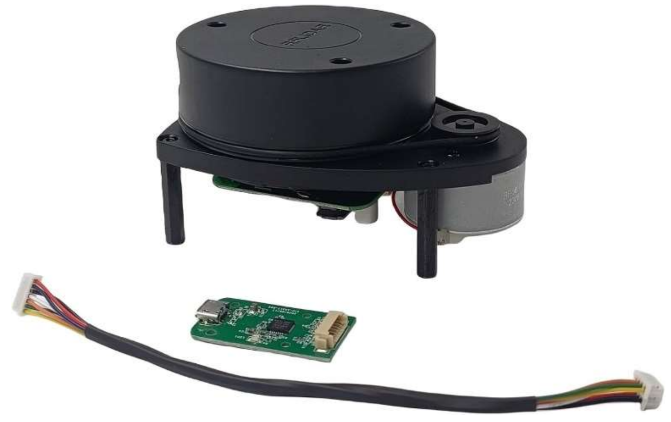


Figure 1.5.2 Light Detection and Ranging(lidar)

1.5.3 Wheeled encoders :

Wheeled encoders are important sensors used in mobile robots for measuring wheel rotation by means of rotary encoders. The encoders may be either mechanical brushes or optical systems such as infrared light beams that capture incremental wheel rotations. As a wheel rotates, the encoder generates electrical pulses, such as revolution times, that are processed to construct other motion parameters such as linear speed, angular velocity, and heading. By combining data on the diameter of the wheel or circumference with revolution times the system can easily convert raw encoder signals to motion parameters that can then be integrated into the movement of the robot. In sensor fusion applications, data obtained with wheel encoders can be combined with inertial measurements (e. g. from accelerometers or gyroscopes) to improve navigation accuracy. This fusion typically treats wheel-derived speed and heading as components of a measurement vector, thereby providing robust state estimation for the robot's localization and motion control

1.5.4 camera

In robotics, the camera functions as the robot's eye, capturing visual data that is essential for perceiving and understanding the environment. Applications include object recognition, facial recognition, depth estimation using stereo vision (two cameras), monocular visual SLAM (Simultaneous Localization and Mapping), color detection, and object tracking—all of which rely on image data. There are various types of cameras used in image processing, but this context focuses on USB cameras. A USB camera refers to any camera that connects via a USB interface, commonly categorized under the USB Video Class (UVC) standard [20]. The following(figure -1-6-1-4) shows one of these types of cameras

Figure 1.6.1.4 Camera Module with Omnivision OV5648 sensor for Robotics,

1.6 Mobile robot motors:

The mechanics mobile robot motors serve as a pivotal form of innovation as they enable the movement, flexibility, and systems operation of autonomous structures. Motors primarily acts as the source of mechanical power therefore these forms needs to provide resolve within fuel efficiency, and resource consumption. Energy powered motors offer an extensive list of selection from brushed and brushless DC motors to servo and stepper motors which rely on power, temperature, settings, and deliverable accuracy defined by the environment. Industrial, service, exploratory and complex robotic structures enables deep navigation and perform with industrial grade standards with self expression in terms of creativity design with traits like divergent versatile accuracy. Knowing aids from unlocking boundless complex task and unlock endless features.

1.6.1 Type of motors

1.6.1.1 DC motors :

DC motors are widely used in mobile robots due to their versatility in size, power, torque, voltage, and RPM, which allow them to meet diverse application requirements. These motors are categorized into two main types: brushed DC motors and brushless DC motors. Brushed DC motors operate by rotating an armature (with coils) within a stationary magnetic field. The brushes and commutators physically reverse the current direction through the coils as the armature spins, enabling continuous rotation and mechanical power generation. Brushless DC motors, in contrast, eliminate brushes and commutators, relying instead on electronic controllers to synchronize the voltage applied to stator coils, which drives the rotor efficiently and with reduced wear. (Figure 1.6.1) illustrates the basic structures of these two motor types.

In mobile robots, precise control of motor speed and direction is critical. This is typically achieved using an H-bridge circuit

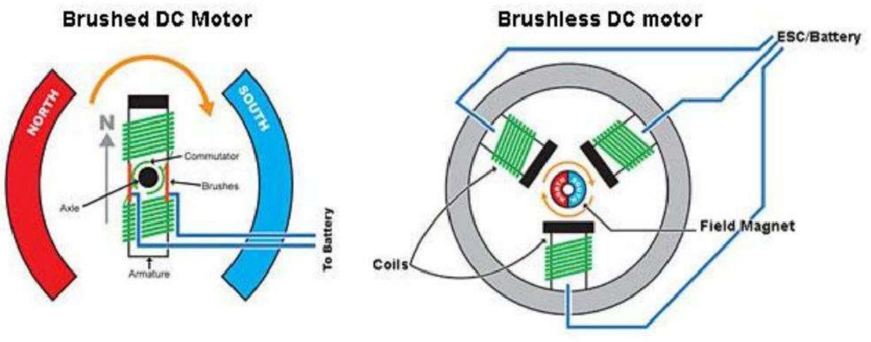


Figure 1.6.1 The difference between brushed DC motor and brushless DC motor

1.6.1.2 servo motors :

Servo motors are widely used in mobile robots for their exceptional precision in controlling angular or linear displacement. These motors incorporate a feedback control system that continuously monitors the shaft's position, enabling accurate adjustments to achieve specific angles or distances—critical for tasks like steering, joint articulation, or sensor positioning in robotics. Comprising a standard motor coupled with a servo mechanism, servo motors adjust their operation based on real-time feedback from sensors such as encoders or potentiometers. They are categorized as DC servo motors (powered by direct current) or AC servo motors (powered by alternating current), with compact, efficient DC variants being preferred in battery-driven mobile robots due to their compatibility with portable power systems. [21]

The(figure-1-6-2) shows the basics and technology of DC servo motors

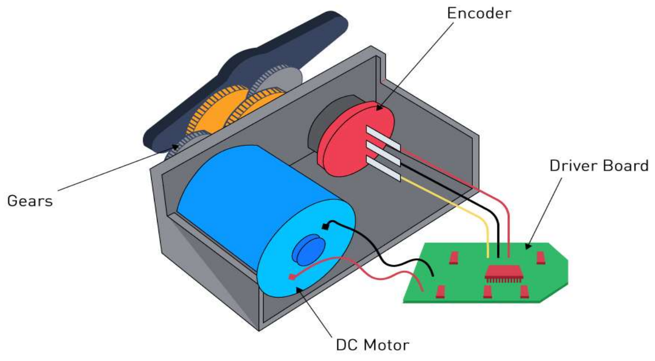


Figure 1.6.2 Basics of Servo Motor Technology

1.1.1.1.1 stepper motors :

Stepper motors, in simple terms, are DC brushless motors. They are capable of self-positioning, so they usually do not need an encoder for position feedback. An encoder can be used in some specific cases to detect stalls during operation.

Compared to their weight and size, these motors provide exceptional torque. Although, they do lose a lot of torque at higher speeds, and their complex torque-speed characteristics make it difficult to choose suitable motors. But stepper motors can achieve speeds of approximately 5000 RPM with almost no torque.

The range of power extends to several hundred watts, but it rarely exceeds this level

Conclsion

The first chapter serves as the cornerstone for understanding the world of mobile robotics, covering a range of foundational concepts and structural components that form the infrastructure of any robotic system. Starting with defining key terms such as robot, autonomy (autonomous), and degrees of freedom (DOF) , and progressing to mechanical design and control systems, this chapter provides an integrated vision of the elements of mobile robots and their interactions.

The types of wheeled robots (such as two-wheeled, four-wheeled, and tracked systems) were clarified, along with an analysis of their advantages and challenges. This highlights how a robot's physical design determines its ability to adapt to diverse environments. The mechanical design section bridged engineering principles with practical performance, emphasizing the importance of balancing durability, efficiency, and flexibility.

Through robot kinematics and dynamics, the theoretical framework explaining how software commands translate into tangible motion was introduced, which is central to motion control and navigation. Meanwhile, sensors and motors were presented as vital components that enable a robot to interact with its surroundings (by detecting obstacles and localizing itself) and convert electrical energy into mechanical action.

Overall, this chapter combines theory and application, demonstrating that developing an effective autonomous robot requires precise integration of physical components (such as wheels and motors) and software systems (including control algorithms and models). This comprehensive vision serves as a launching point for understanding future challenges, such as enhancing autonomy and advancing artificial intelligence, which will be explored in subsequent chapters.

Chapter 2

Robotic Operating System ROS2 – Humble

2 Introduction :

The advent of the Robotics Operating System (ROS2) Humble marks a paradigm shift in the development of intelligent robotic systems, particularly in the realm of autonomous mobile robots. As robotics transitions from static industrial applications to dynamic, real-world environments, the need for modular, scalable, and interoperable frameworks has become paramount. ROS2 Humble, with its enhanced middleware and distributed architecture, addresses these challenges by providing a unified ecosystem for designing, simulating, and deploying robotic behaviors—most notably through Behavior Trees, a powerful paradigm for orchestrating complex decision-making processes

This chapter serves as an academic gateway to the core principles of ROS2 Humble, tailored for researchers, engineers, and students seeking to integrate simulation-driven development with autonomous behavior design. Unlike traditional robotic frameworks, ROS2 Humble embraces a decentralized communication model, enabling seamless interaction between software components while maintaining robustness in unpredictable environments. Its compatibility with physics-based simulators like Gazebo and visualization tools such as RVIZ further bridges the gap between theoretical algorithms and real-world applicability, offering a sandbox for testing mobile robot navigation, obstacle avoidance, and task execution.

Through this academic lens, the chapter underscores the transformative potential of ROS2 Humble in advancing robotic autonomy, offering a foundational primer for those aspiring to pioneer next-generation mobile robotics solutions.

2.1 Packages:

In ROS 2 Humble, a package serves as the fundamental organizational and distribution unit for software components. Packages enable modular code structuring, facilitating installation, reuse, and collaborative development within the ROS ecosystem. Adhering to a standardized packaging paradigm ensures compatibility with ROS 2 tooling and simplifies the sharing of functionality across projects and communities.

The packaging infrastructure in ROS 2 Humble leverages the ament build system and colcon build tool to automate compilation, dependency resolution, and deployment processes. The *ament* build system provides a framework for configuring, building, and installing ROS 2 components, while *colcon* operates as the meta-toolchain responsible for orchestrating these operations across interconnected packages.

Officially supported build types include Cmake (for C++-based packages) and Python (via the setuptools framework), which integrate natively with ROS 2 workflows. These build systems enable developers to define package metadata, dependencies, and build instructions through standardized configuration files (e.g., package.xml, CmakeLists.txt, or setup.py). While third-party build systems may be adapted for specialized use cases, Cmake and Python remain the recommended approaches for ensuring compatibility with ROS 2 Humble's core tooling.

By encapsulating code, resources, and dependencies within a well-defined package structure, developers streamline the distribution and deployment of robotic applications. Properly configured packages allow seamless integration with ROS 2 tools for dependency management, version control, and cross-platform compilation, thereby promoting reproducibility and scalability in complex robotic systems. [22]

2.1.1 Workspaces as Packages :

The intention behind a ROS 2 Humble workspace is to allow multiple packages to function independently and cohesively as single entities allowing modular code organization within a singular development environment. All packages within a workspace require their own dedicated directories. This ensures logical separation of functionalities and of build configurations, which is of utmost importance. ROS 2 does impose restrictions on rules of organizational structure with respect to packages, forbidding the use of hierarchical containment of packages (also known as *nested packages*), since every package is an autonomous unit with its own metadata and build independencies.

The existence of multiple types of building engines is inherently supported within the workspace. All packages using Cmake, Python, or other supported building engines can co-exist within a single workspace. This enables developers to plug in a variety of software components, like C++ nodes and Python scripts, within a single workspace and simplifies development and testing across multiple components.

Their suggests organizational practice is to place every package within a **src directory** in the very root of the workspace. This helps maintain a clean top-level workspace where building, installing, and log artifact overwrites done during compilation do not tangentially interfere with sourced code. Developers are ensured compatability with ROS 2 tooling like colcon, which rely on assuming a directory hierarchy fo

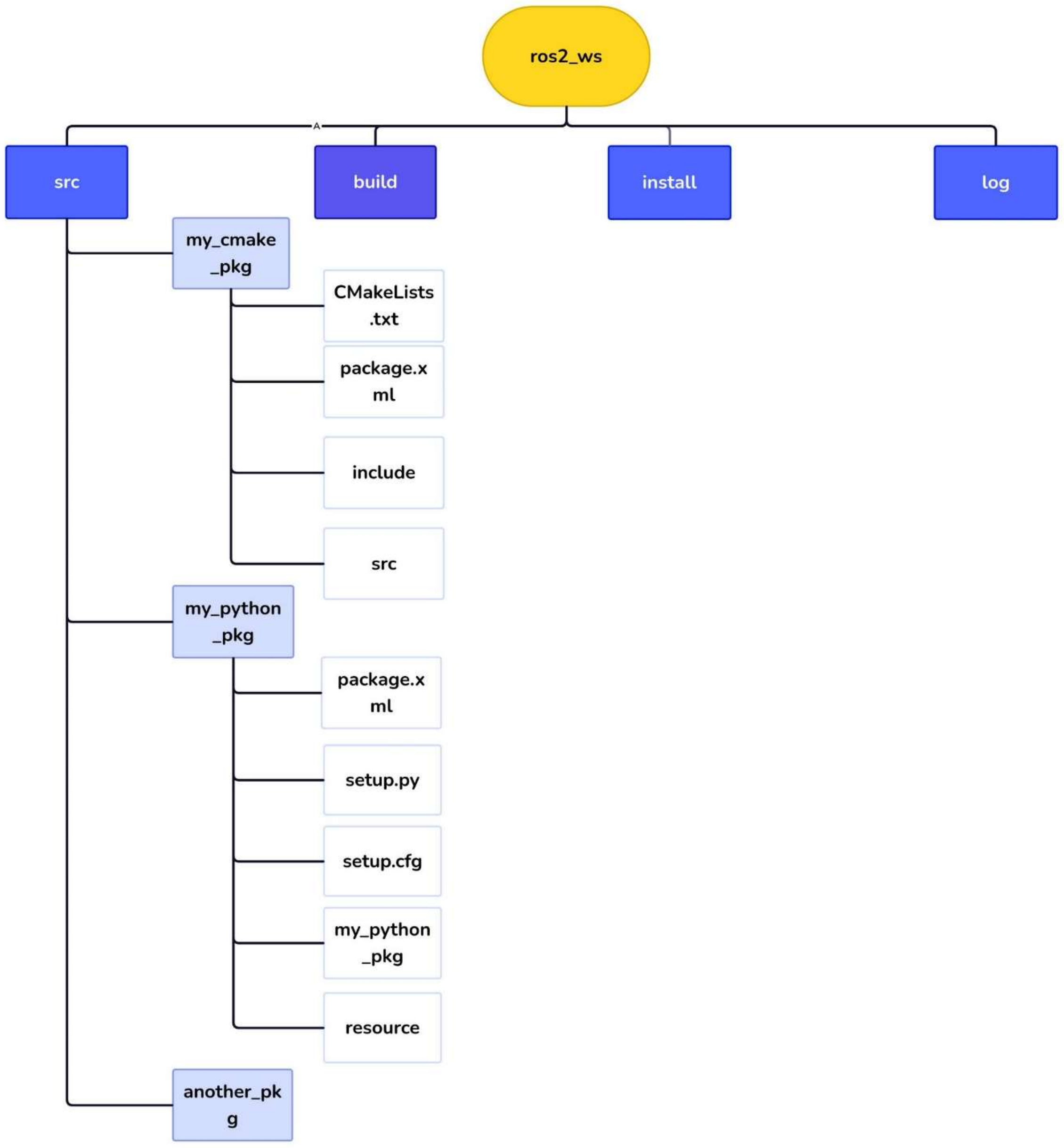


Figure 2.1.1 diagram for package in ros2

2.2 Launch files:

Launch files in ROS 2 are employed to launch several nodes and set their behavior from a single file. Rather than launching each node individually, launch files allow you to specify how your system should launch, which parameters to employ, and what interaction is required between the nodes. In ROS 2, launch files normally consist of Python code and offer an effective means of launching complex robotic applications automatically. Figure 2.2.3 explain that more

2.2.1 **Creating a Launch File** :

This chapter defines a systematic approach to ROS 2 launch file construction in Python, XML, and YAML formats. Employing the *turtlesim* package as a teaching resource, it illustrates key methods of setting up multiple nodes, remapping topics, and handling dependencies. An analysis of the syntax and use case applicability of the three formats is provided to inform format choice in accordance with the level of project complexity and required maintainability. [23]

2.2.2 **Integrating Launch Files into Packages** :

This chapter codifies best practice in embedding launch files into ROS 2 package environments. It specifies filesystem conventions (e.g., /launch directory organization) and build-system setup in both Python (setup.py entry points) and C++/Cmake (CmakeLists.txt directives). Platform-independent installation routines and verification protocols are prioritized to promote uniform deployment to environments. [24]

2.2.3 **Advanced Launch Concepts:**

This lesson examines higher-level capabilities of the ROS 2 launch system in industrial and research contexts. These include dynamic parameter substitutions (for example, LaunchConfiguration), event-driven execution semantics (for instance, OnProcessExit handlers), and compositional design patterns in large-scale systems. This part connects theory to practice, citing ROS 2 launch API documentation to add extra technical detail. [25]

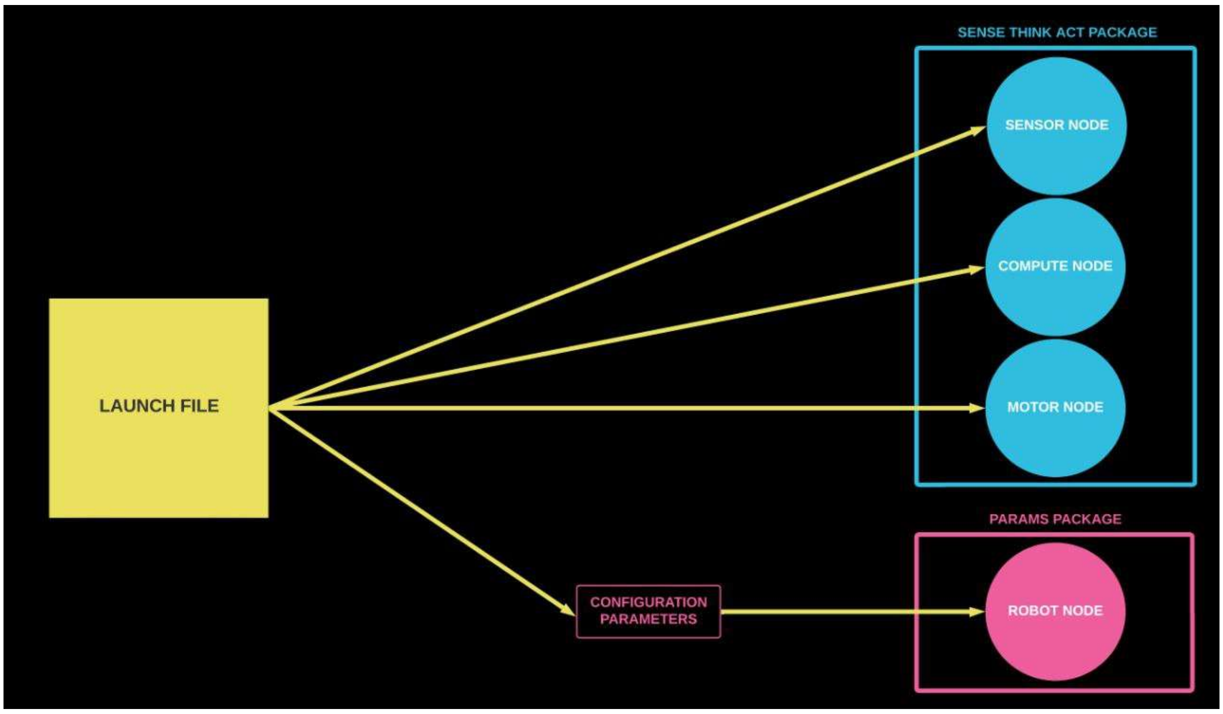


Figure 2.2.3 Launch files in ROS 2

2.3 Nodes:

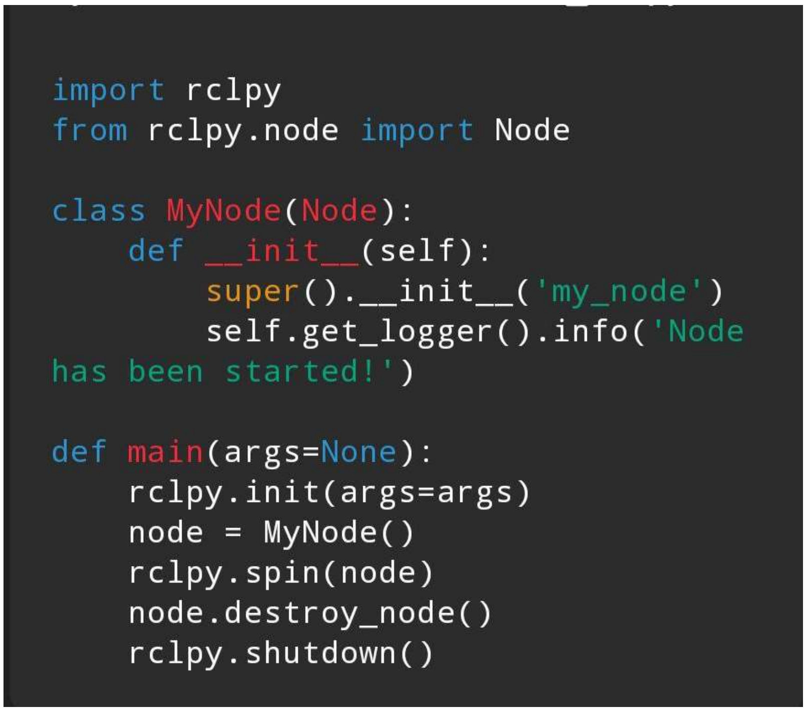
In ROS2, nodes are designed to serve specialized functions within an application. For instance, a node might operate a robot arm's joint motors, process sensor data, or perform other singular tasks. Each executable in ROS2 can host multiple nodes or a single node, depending on the system's architecture. [26]

2.3.1 Programming languages :

Nodes can be written in python (using rclpy) or c++ (using rlcpp)

The rclpy and rclcpp libraries both include a Node class, which acts as a foundational wrapper for integrating your code with ROS 2 functionalities. To leverage these capabilities, you structure your code by creating subclasses derived from the Node class, granting access to its public members (methods and properties). Organizing your application into multiple nodes allows parallel processing of distinct code segments, enhancing efficiency. Additionally, the methods inherited from the Node class facilitate

communication—both between nodes (inter-node) and within a single node (intra-node)—through ROS 2's built-in mechanisms like topics, services, and actions [27]



```
import rclpy
from rclpy.node import Node

class MyNode(Node):
    def __init__(self):
        super().__init__('my_node')
        self.get_logger().info('Node
has been started!')

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

Figure 2.3.1 example basic to using rclpy

2.3.2 communication of nodes :

Nodes interact with each other through various communication mechanisms, including topics (asynchronous data streams), services (synchronous request-response interactions), actions (long-running tasks with feedback), and parameters (runtime configuration settings). These communication frameworks form the next layer of components to examine in ROS2's structure.

This version maintains the core information but alters sentence structure, vocabulary, and emphasis while ensuring technical accuracy.

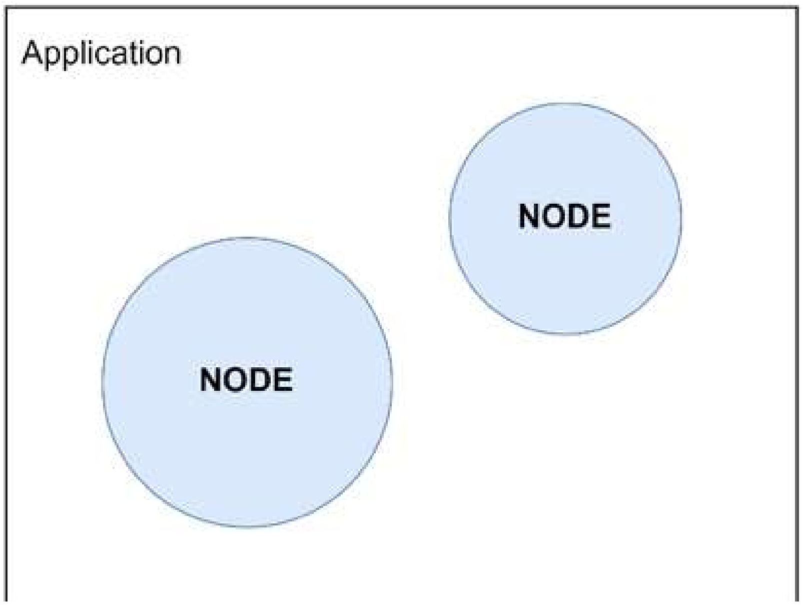


Figure 2.3.2 . Representation of nodes into an application

2.3.3 Publisher nodes:

Publisher nodes function as computational entities responsible for generating and distributing data. These nodes disseminate messages of a designated type through a communication channel (topic), thereby facilitating asynchronous data accessibility for subscriber nodes within distributed systems."

This revision emphasizes technical precision, formalizes key concepts (e.g., "computational entities," "disseminate," "asynchronous"), and clarifies the structural role of topics as communication channels. [28]

2.3.4 Subscribers nodes :

Subscribers, functioning as nodes within a distributed system, are components designed to receive and process data by subscribing to messages or information disseminated through designated topics."

This version emphasizes their role in a system, uses formal terminology (e.g., "disseminated," "designated topics"), and integrates the two original sentences into a cohesive statement [28]

2.4 Topics:

Topics serve as the primary communication mechanism between nodes in distributed robotic systems, operating as an asynchronous messaging interface. Functioning as a communication channel, a topic enables unidirectional data transmission from publisher objects (message producers) to subscriber objects (message consumers), both of which are components of nodes. A single topic may be associated with an arbitrary number of publishers and subscribers [27], facilitating a many-to-many communication architecture (Figure 2.4).

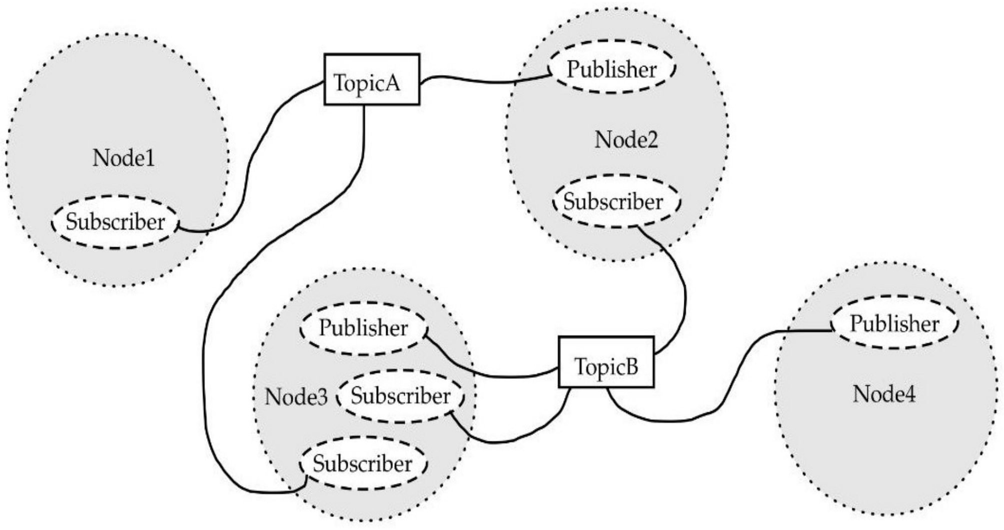


Figure 2.4 Node Communication via Topics A and B (with Self-Message Possibility)

The Robot Operating System (ROS) client libraries, namely *rclpy* (Python) and *rclcpp* (C++), implement this paradigm by providing application programming interfaces (APIs) to manage topic-based communication. Specifically, these libraries offer functionality for instantiating publisher objects equipped with a publish() method for message transmission to designated topics, while subscriber objects utilize callback functions that are automatically invoked upon message receipt. This design adheres to an event-driven model, where incoming messages asynchronously trigger predefined subscriber callbacks

2.4.1 Decoupling of Publishers and Subscribers:

One of the interesting features of the ROS 2 communication model, especially when working with topics, is its built-in anonymity. In simple terms, subscribers usually don't need to know exactly which publisher sent the data they receive. While it's possible to find out the source if needed, it's not something the system relies on for normal operation. This design choice helps keep things flexible and modular—developers can easily add, remove, or swap out publishers and subscribers without breaking the rest of the system.

It's a smart approach that fits well with the goals of building scalable and adaptable robotic systems, especially as they grow more complex over time. [29]

2.5 Services :

In ROS 2, the process of creating a new service starts by defining the structure of the request and response in a .srv file. Unlike message definitions, a service definition file contains two sections: inputs and outputs, separated by a line with three dashes (---). For example, if we want to create a service that counts the number of words in a string, the input would be a string and the output an integer. You can use basic data types or any existing ROS message types in this definition

After defining the service, two nodes are created

The first node is called the Service Server, which is responsible for implementing the logic of the service (such as counting the words) and responding to requests.

The second node is the Service Client, which sends the request to the server and waits for the response.

Communication between the nodes happens through a shared service name used by both nodes, and ROS 2 automatically manages this connection when both nodes are running within the same system [30]

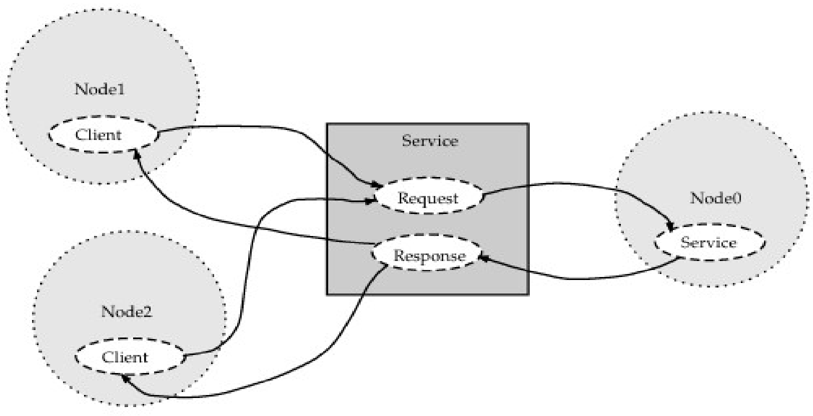


Figure 2.5 Communication between nodes through a shared service

2.5.1 Service Server :

A service server functions as computational unit mediating interactions between clients and servers through highly structured message exchange protocols obviously. It receives formalized service requests and executes predefined operations like data processing or resource allocation very swiftly returning outcome-based responses. Server resides on network node painstakingly configured rather quietly to interpret incoming requests somewhat creatively according to various service specs. It generates response messages quickly after executing domain-specific logic that adhere to formats defined by protocol ensuring interoperability between various distributed components. Decoupled communication gets emphasized in this architecture allowing service-oriented systems to scale and stay maintainable with transactional consistency preserved. [31]

2.5.2 Service Client :

Client components operate by submitting requests to servers and processing responses exchanged as structured messages very quickly nowadays. A node-hosted service client initiates specific commands and subsequently handles returned output from server fairly quickly in most cases normally. Client submits service request and server executes operation; outcome gets relayed back through designated message channel subsequently.

2.5.3 Using a Service:

The easiest way to use a service is by calling it with the rosservice command. For example, to use our word-counting service, you would run:

```
user@hostname$ rosservice call word_count ‘one two three’ count: 3
```

This command uses the call subcommand, followed by the service name and its arguments. Although this approach helps verify that the service is functioning correctly, it’s not as practical as invoking it from within another active node

2.6 Actions :

Asynchronous communications are where a node sends a request to another node and can take some time.

The requesting node may also obtain interim feedback while the read operation is performed as well as a final message of success or error. For example, an avigation request —which can be a costly operation—prevents the requesting node from getting stuck waiting for the request to complete and rather enables the node to perform other operations simultaneously [32]

The following figure (2.6) shows exactly how the process takes place

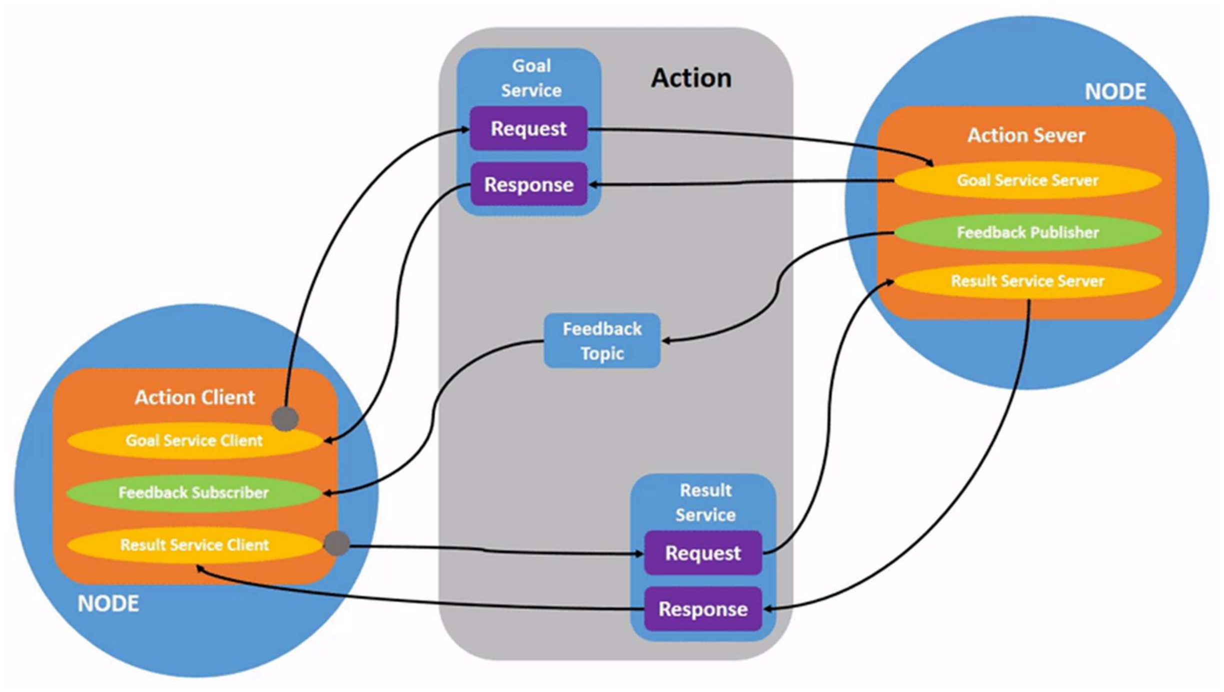


Figure 2.6 Asynchronous communications

2.6.1 Action Server

An action server performs a particular action, and like topics and services, it also has a name and a type. The name is allowed to / must contain namespaces and be unique among all action servers. This enables multiple action servers to be active at the same time that belong to the same server, in different namespaces. [33]

It is responsible for:

- advertising the action to other ROS entities
- accepting or rejecting goals from one or more action clients
- executing the action when a goal is received and accepted
- optionally providing feedback about the progress of all executing actions
- optionally handling requests to cancel one or more actions
- sending the result of a completed action, including whether it succeeded, failed, or was canceled, to a client that makes a result request.

2.6.2 Action Client

An action client sends either a single or a group of goals—each a task to be performed—and monitors their execution. Although a single action server may have multiple clients communicating with it, it will decide how it handles and executes goals from various clients simultaneously

. It is responsible for:

- sending goals to the action server
- optionally monitoring the user-defined feedback for goals from the action server
- optionally monitoring the current state of accepted goals from the action server (see Goal States)
- optionally requesting that the action server cancel an active goal
- optionally checking the result for a goal received from the action server

2.6.3 Action Interface Definition

Actions in ROS are defined using the ROS Message Interface Definition Language (IDL). Each action is composed of three distinct message specifications, organized as follows:

2.6.3.1 Goal :

Defines the objective and parameters of the action. This message is sent by a client to an action server to initiate execution.

2.6.3.2 Result :

Describes the final outcome (success or failure) of the action. This message is sent from the action server to the client once the action completes.

2.6.3.3 Feedback :

Provides periodic updates on the action's progress during execution. These messages are sent from the action server to the client while the action is ongoing.

2.6.3.4 Formatting Rules:

- The three sections are separated by a line containing three hyphens: ---.
- Any section (Goal, Result, or Feedback) may be empty if unnecessary for the action.
- Actions are stored in files with the .action extension.

- One action definition is included per .action file.

2.6.4 Middleware implementation

This summary outlines the middleware implementation of actions in ROS 2, focusing on the communication between the action client and server. It explains the purpose and structure of the three main services and two topics used to manage and monitor goals in a robotic system.

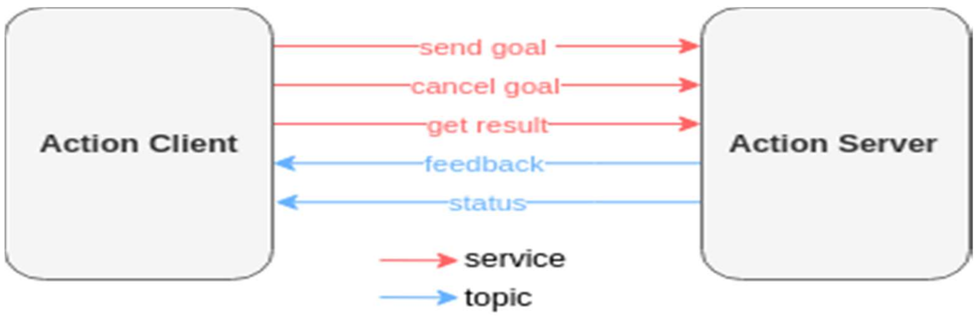


Figure 2.6.4 communication between the action client and server.

Send Goal Service:

- Used by the client to send a goal to the server.
- Request: goal description and UUID.
- Response: whether the goal was accepted or rejected, with timestamp.

Cancel Goal Service:

- Used to cancel goals.
- Request: goal ID and timestamp.
- Response: result code and list of canceled goals.
- The policy follows ROS 1 rules for goal cancellation.

Get Result Service:

- Retrieves the final result of a goal.
- Request: goal ID.
- Response: result status and user-defined result.
- Results can be cached until introspection tools access them.

Goal Status Topic:

- Published by the server.
- Contains the status of all active goals.
- Not used by clients but useful for monitoring.
- Possible states: Accepted, Executing, Canceling, Succeeded, Aborted, Canceled.

Feedback Topic:

- Published by the server.
- Contains goal ID and user-defined feedback message.
- Used to provide progress updates on goals.

2.7 RVIZ :

For Rviz, it permits users to take 3D visualization setting of robots, sensors or any sensor data pertaining to an environment which helps in debugging and development. Rviz is a 3D visualization tool to the ROS framework.

2.7.1 startup rviz

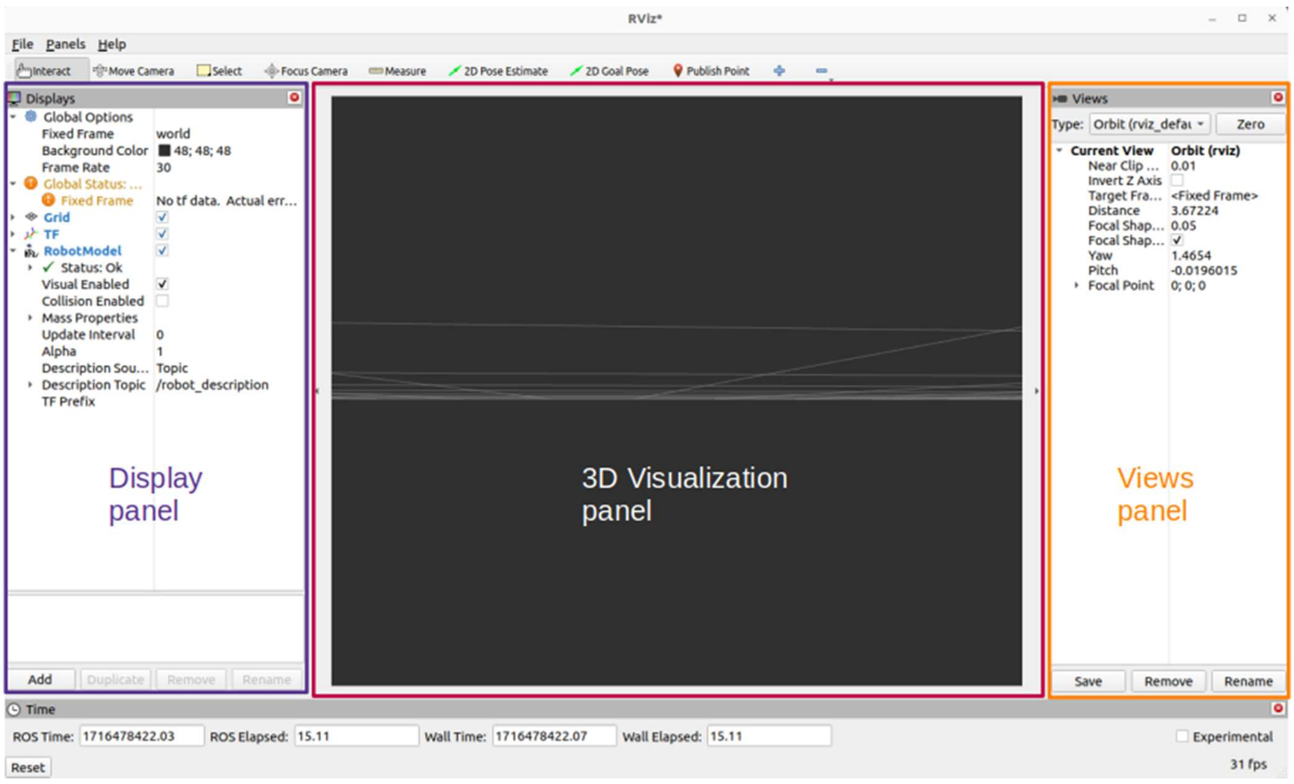
Before launching Rviz, source the ROS 2 setup file:

```
source /opt/ros/humble/setup.bash
```

Then, start Rviz with the following command:

```
ros2 run rviz2 rviz2
```

2.7.2 Rviz Screen Components



Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
Global Status: ...	
Fixed Frame	No tf data. Actual err...
Grid	✓
TF	✓
RobotModel	✓
Status: Ok	
Visual Enabled	✓
Collision Enabled	☐
Mass Properties	
Update Interval	0
Alpha	1
Description Sou...	Topic
Description Topic	/robot_description
TF Prefix	

Type: Orbit (rviz_defau

Current View	Orbit (rviz)
Near Clip ...	0.01
Invert Z Axis	☐
Target Fra...	<Fixed Frame>
Distance	3.67224
Focal Shap...	0.05
Focal Shap...	✓
Yaw	1.4654
Pitch	-0.0196015
Focal Point	0; 0; 0

ROS Time: 1716478422.03 ROS Elapsed: 15.11 Wall Time: 1716478422.07 Wall Elapsed: 15.11

31 fps

Figure 2.7.2 Rviz Screen

2.7.2.1 Display Panel

- This panel shows a list of all the visual elements currently being displayed in the 3D view.
- You can add, remove, or configure different display types such as Grid, TF, RobotModel, etc.
- Each element has parameters that can be adjusted (e.g., color, topic, visibility).
- It's useful for managing what information is shown in the visualization. [34]

2.7.2.2 3D Visualization Panel

- This is the main area where all the visual elements are rendered in 3D.
- You can rotate, zoom, and pan the view to explore the robot and environment.
- Displays added in the Display Panel appear here visually (like robot models, sensor data, maps, etc.).

2.7.2.3 Views Panel

- This panel allows you to manage camera views and control how the 3D scene is displayed.
- You can set the camera type (e.g., Orbit, FPS), modify the view angle, distance, focal point, etc.

2.7.3 Rviz Displays :

The most frequently used menu when using Rviz will probably be the Displays5 menu. This Displays menu is used to select the message to display on the 3D View panel, and descriptions on each item are explained in table 2-7-3 [35]

Table 2 Rviz Displays

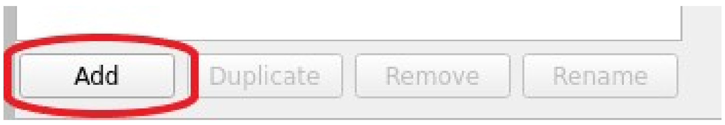
Name	Description	Messages Used
Axes	Displays a set of Axes	
Effort	Shows the effort being put into each revolute joint of a robot	sensor_msgs/msg/JointStates
Camera	Creates a new rendering window from the perspective of a camera, and overlays the image on top of it.	sensor_msgs/msg/Image, sensor_msgs/msg/CameraInfo
Grid	Displays a 2D or 3D grid along a plane	
Grid Cells	Draws cells from a grid, usually obstacles from a costmap from the navigation stack.	nav_msgs/msg/GridCells
Image	Creates a new rendering window with an Image. Unlike the Camera display, this display does not use a CameraInfo	sensor_msgs/msg/Image
InteractiveMarker	Displays 3D objects from one or multiple Interactive Marker servers and allows mouse interaction with them	visualization_msgs/msg/InteractiveMarker

Name	Description	Messages Used
Laser Scan	Shows data from a laser scan, with different options for rendering modes, accumulation, etc.	sensor_msgs/msg/LaserScan
Map	Displays a map on the ground plane.	nav_msgs/msg/OccupancyGrid
Markers	Allows programmers to display arbitrary primitive shapes through a topic	visualization_msgs/msg/Marker, visualization_msgs/msg/MarkerArray
Path	Shows a path from the navigation stack.	nav_msgs/msg/Path
Point	Draws a point as a small sphere.	geometry_msgs/msg/PointStamped
Pose	Draws a pose as either an arrow or axes.	geometry_msgs/msg/PoseStamped
Pose Array	Draws a “cloud” of arrows, one for each pose in a pose array	geometry_msgs/msg/PoseArray
Point Cloud(2)	Shows data from a point cloud, with different options for rendering modes, accumulation, etc.	sensor_msgs/msg/PointCloud, sensor_msgs/msg/PointCloud2
Polygon	Draws the outline of a polygon as lines.	geometry_msgs/msg/Polygon
Odometry	Accumulates odometry poses from over time.	nav_msgs/msg/Odometry

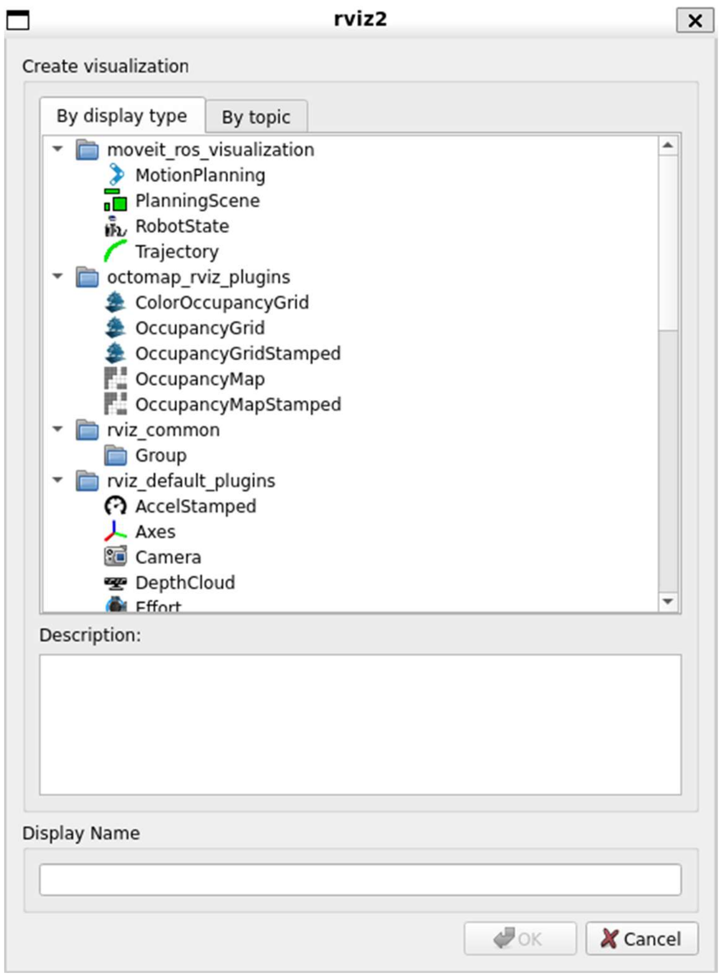
Name	Description	Messages Used
Range	Displays cones representing range measurements from sonar or IR range sensors. Version: Electric+	sensor_msgs/msg/Range
RobotModel	Shows a visual representation of a robot in the correct pose (as defined by the current TF transforms).	
TF	Displays the tf2 transform hierarchy.	
Wrench	Draws a wrench as arrow (force) and arrow + circle (torque)	geometry_msgs/msg/WrenchStamped
Twist	Draws a twist as arrow (linear) and arrow + circle (angular)	geometry_msgs/msg/TwistStamped

2.7.4 Adding a new display

To add a display, click the Add button at the bottom:



This will pop up the new display dialog:


rviz2

Create visualization

By display type | By topic

- moveit_ros_visualization
  - MotionPlanning
  - PlanningScene
  - RobotState
  - Trajectory
- octomap_rviz_plugins
  - ColorOccupancyGrid
  - OccupancyGrid
  - OccupancyGridStamped
  - OccupancyMap
  - OccupancyMapStamped
- rviz_common
  - Group
- rviz_default_plugins
  - AccelStamped
  - Axes
  - Camera
  - DepthCloud
  - Effort

Description:

Display Name

OK | Cancel

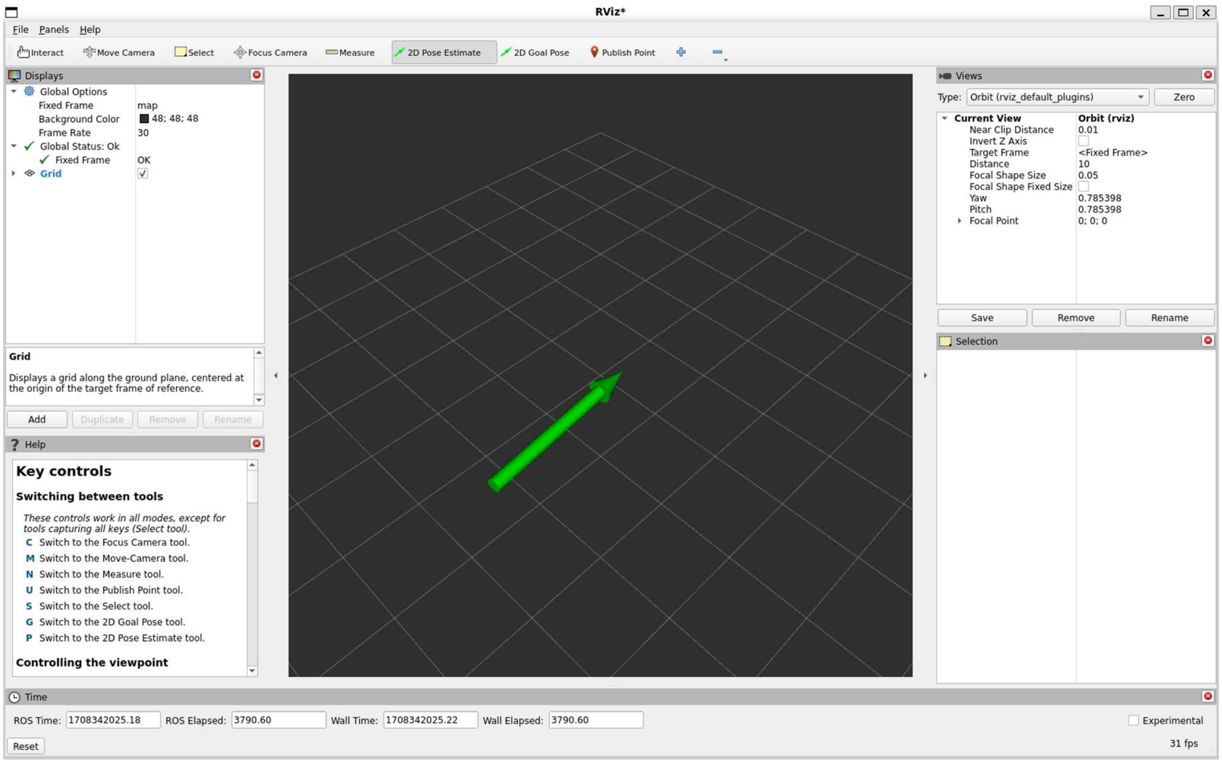
Figure 2.7.4 display dialog

At the top of the list, you'll find the display type, which indicates the kind of data the display will show. The text box in the center provides a description of the chosen display type. Lastly, you need to assign a unique name to the display. For instance, if your robot has two laser scanners, you might name them “Laser Base” and “Laser Head” to distinguish between them.

2.7.5 Tools :

2.7.5.1 2D Pose Estimate:

With this tool, you can set an initial pose to start the localization system (published needed in the initialpose ros topic). The ground plane can be clicked and dragged to set a desired orientation. The output topic can be altered in the tool properties pane



Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Fixed Frame OK
Grid ✓

Grid

Displays a grid along the ground plane, centered at the origin of the target frame of reference.

Key controls

Switching between tools

These controls work in all modes, except for tools capturing all keys (Select tool).

C Switch to the Focus Camera tool.
M Switch to the Move-Camera tool.
N Switch to the Measure tool.
U Switch to the Publish Point tool.
S Switch to the Select tool.
G Switch to the 2D Goal Pose tool.
P Switch to the 2D Pose Estimate tool.

Controlling the viewpoint

Type: Orbit (rviz_default_plugins)

Current View Orbit (rviz)
Near Clip Distance 0.01
Invert Z Axis
Target Frame <Fixed Frame>
Distance 10
Focal Shape Size 0.05
Focal Shape Fixed Size
Yaw 0.785398
Pitch 0.785398
Focal Point 0; 0; 0

ROS Time: 1708342025.18 ROS Elapsed: 3790.60 Wall Time: 1708342025.22 Wall Elapsed: 3790.60

Figure 2.7.5.1 2D Pose Estimate

2.7.5.2 The 2D Nav Goal button :

The 2D Nav Goal button is used to give a goal position to the move_base node in the ROS Navigation stack through RViz. We can select this button from the top panel of RViz and can give the goal position inside the map by left clicking the map using the mouse. The goal position will send to the move_base node for moving the robot to that location. [36]

- **Topic**: move_base_simple/goal
- **Topic Type**: geometry_msgs/PoseStamped

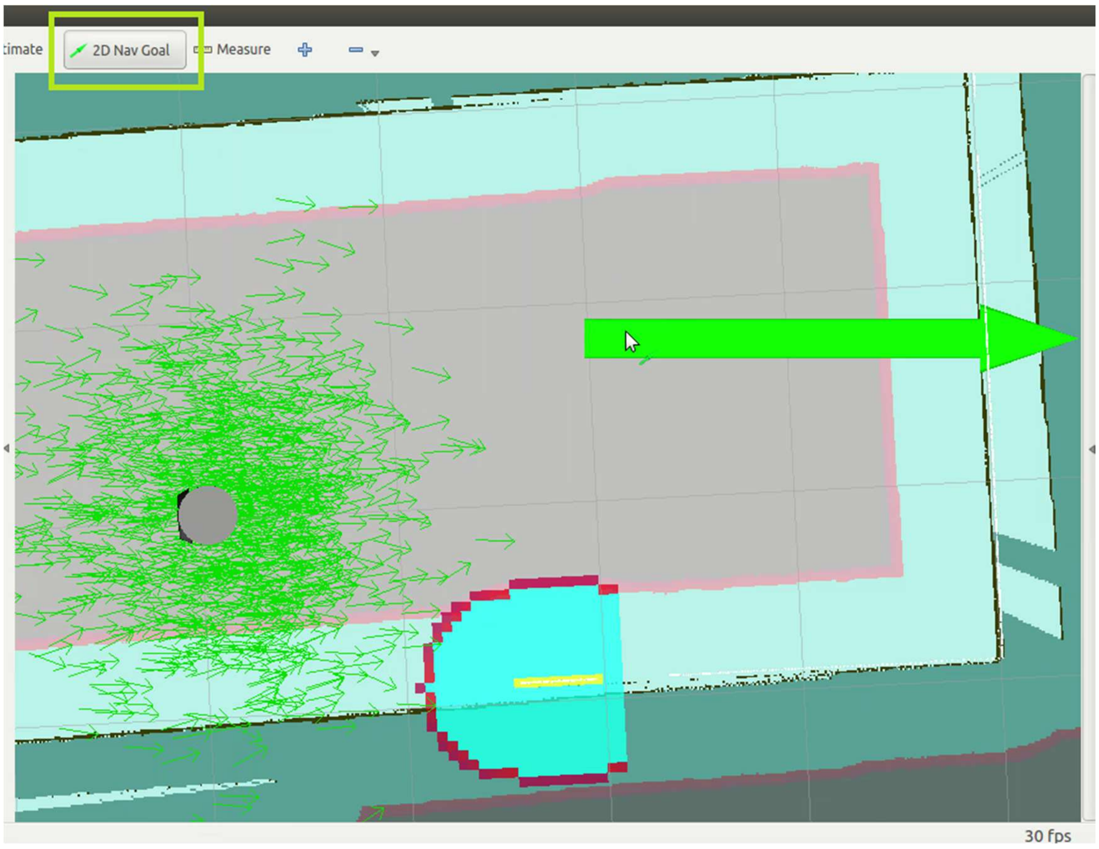


Figure 2.7.5.2 The 2D Nav Goal

2.7.5.3 . Publish Point

The publish point tool lets you select an object in the visualizer and the tool will publish the coordinates of that point based on the frame. The results are shown at the bottom like with the measure tool but are also published on the `clicked_point` topic.

2.8 Gazebo :

Gazebo is an open-source 3D simulation environment used for developing and testing robots. It is developed and maintained by Open Robotics and serves as a powerful tool for simulating physics, generating sensor data, and providing both graphical and programmatic interfaces to interact with robots in realistic environments. [37]
Gazebo is known for its ability to accurately and efficiently simulate robots in various applications such as warehouse logistics, autonomous driving, and space exploration. It offers a strong physics engine, high-quality graphics, and integrated APIs, including support for the ROS framework.
Development of Gazebo began in 2002, and in 2017, it was split into two versions: "Gazebo Classic," which retains the original architecture, and "Ignition," which later evolved into a modern set of modular libraries.
Gazebo is widely used in both academic and industrial fields and is considered an essential tool in the development and testing of robotics and intelligent systems. [38]

2.8.1 Gazebo Interface Overview

Gazebo is a powerful simulation tool used to simulate robots in complex indoor and outdoor environments

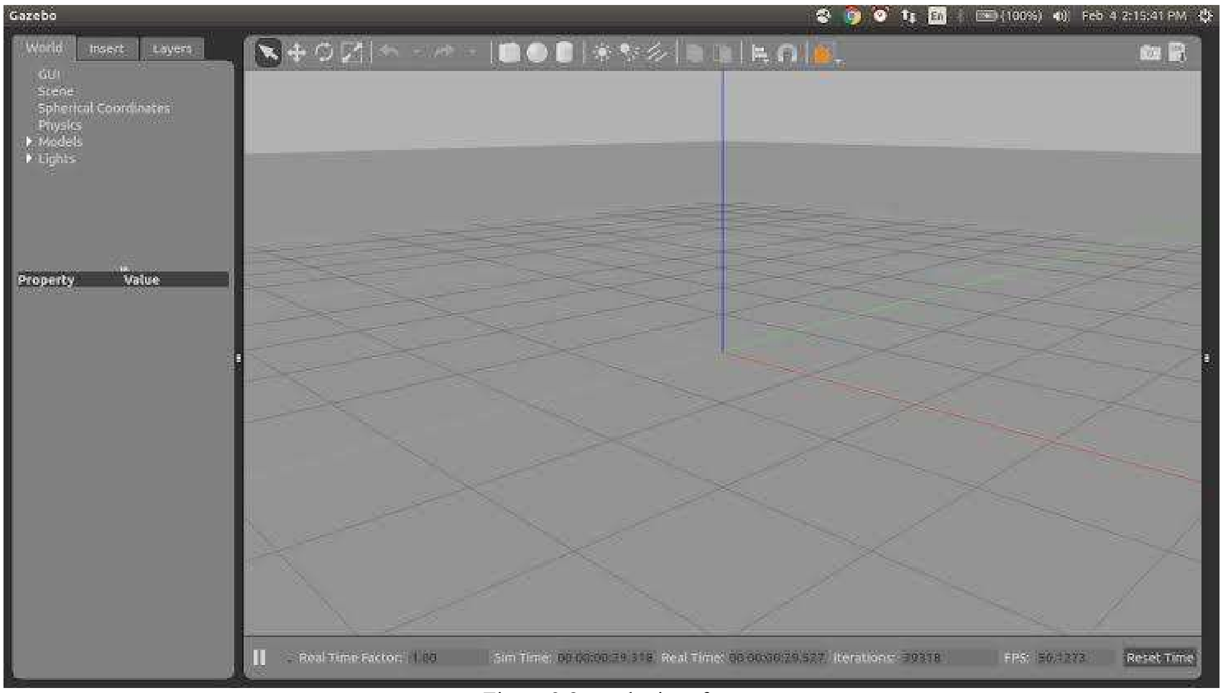


Figure 2.8 gazebo interface

2.8.1.1 Top Toolbar (Horizontal Gray Bar at the Top)

This contains the main control buttons:

- Pause / Play / Step: Control the simulation (play, pause, single step).
- Reset: Reset the simulation or world.
- Camera Tools: Orbit, pan, zoom, view options.
- Model Tools: Insert, select, move, rotate, scale models.
- Light and Object Tools: Add lights, modify objects.
- Screenshot/Recording Buttons: Capture images or record video of the simulation.
- Undo/Redo: For reversing or reapplying actions.

2.8.1.2 Left Sidebar (World / Insert / Layers Tabs)

This is a key area to manage world settings and elements:

a- World Tab
Shows the hierarchy of the simulation world.
Includes:

- GUI: Visual settings.
- Scene: Background, shadows, etc.
- Spherical Coordinates: GPS-style world positioning.

- Physics: Gravity, step size, update rate.
- Models: Inserted robots or objects.
- Lights: Manage lighting in the world.

b- Insert Tab
Used to insert models like robots, tables, sensors, etc.
You can drag-and-drop models into the simulation scene.

c- Layers Tab
Controls visibility of different simulation layers (e.g., terrain, models, sensor data).

2.8.1.3 3D Viewport (Center Area)

This is the simulation world view.

We can:

- Rotate the view (right-click drag).
- Zoom in/out (scroll wheel).
- Pan the camera (middle-click drag).
- The colored axes (red = x, green = y, blue = z) help you understand orientation.

2.8.1.4 Property Viewer (Bottom Left - Property/Value Table)

Shows properties of selected items (models, lights, etc.).

Allows you to see and possibly edit values like position, rotation, color, etc.

2.8.1.5 Simulation Status Bar (Bottom Bar)

Displays real-time simulation information:

- Real Time Factor: Ratio between simulated time and real-world time (ideal is ~1.0).
- Sim Time: Total simulated time.
- Real Time: Time elapsed in the real world.
- Iterations: Number of simulation steps.
- FPS: Frames per second.
- Reset Time Button: Resets the time counter**.**

2.8.2 Integration of ROS 2 Humble with Gazebo Simulator

The connection between ROS 2 Humble and the Gazebo simulator is performed using a bridging interface called ros_gz (formerly ros_ign). Communication between the ROS 2 nodes and the Gazebo simulator is facilitated by the bridge that translates messages, services and actions between both systems.

This configuration enables developers to experiment with robotic algorithms using a more lifelike simulated environment, decreasing the reliance on physical hardware in early development. The integration process includes installation of the right Gazebo version (e.g., Fortress or compatible), setting up the ros_gz_bridge package, and configuration of robot description files in either URDF or SDF formats. A simulator such as this is indispensable for testing of navigation, sensor fusion and control on the real robots. [39]

Conclsion

In summary, this chapter has provided a comprehensive exploration of the Robot Operating System 2 (ROS2) Humble Hawksbill distribution, emphasizing its architecture, tools, and core functionalities that underpin modern robotic system development. Through an academic lens, the chapter systematically dissected the foundational elements of ROS2 Humble, beginning with an overview of its modular framework designed for scalability, real-time performance, and cross-platform interoperability. The discussion progressed to granular components, including packages as the fundamental units for code organization and reuse, launch files for automating complex system configurations, and nodes as the atomic executables enabling distributed computation.

The chapter further elucidated ROS2's communication paradigms: topics for asynchronous publisher-subscriber interactions, services for synchronous request-reply mechanisms, and actions for long-running, interruptible tasks with feedback. These communication models collectively empower developers to design responsive, modular, and fault-tolerant robotic systems. The integration of visualization tools such as RVIZ2 and simulation environments like Gazebo was also highlighted, underscoring their critical roles in prototyping, debugging, and validating robotic algorithms in virtualized and real-world scenarios.

By bridging theoretical concepts with practical implementation strategies, this chapter underscores ROS2 Humble's significance in advancing robotics research and industrial applications. The framework's enhanced security features, support for real-time systems, and compatibility with diverse hardware platforms position it as a pivotal tool in addressing contemporary challenges in autonomous systems. Future work may explore deeper integration with AI/ML frameworks, optimization for edge computing, and expanding interoperability with emerging robotic standards. Ultimately, ROS2 Humble exemplifies a robust, community-driven ecosystem that continues to drive innovation in robotics, fostering the development of intelligent, adaptive, and scalable solutions for tomorrow's technological demands.

Chapter III

Mobile robot simulation

3.1 Introduction

In the advanced robotics generation, simulation plays a vital role in designing and developing smart structures before testing them in real-world environments. Simulation provides a safe environment for testing algorithms, understanding performance, and identifying and fixing errors without compromising hardware or requiring expensive physical resources. Simulating mobile robots is an essential step in the development of autonomous robotic systems, especially for tasks requiring autonomous navigation, orientation, and interaction with the environment. Building a simulation environment begins with installing Ubuntu 22.04, which provides a robust platform for robot development. ROS2 Humble, a version of the Robot Operating System (ROS 2), is then installed, providing an integrated framework for developing flexible and scalable robotic applications. Next, Gazebo, one of the most advanced robotics simulators, is installed, providing realistic 3D physics and a range of sensors for enhanced virtual reality. Blender is used to optimize 3D designs of robotic models, producing exportable working models for use in Gazebo. On the software side, developing software applications using Python and C++ enables controlling and managing the robot's behavior, ensuring dynamic interaction with the environment. Robot models are then described using URDF and SDF files. URDF (Robot Description Format) is used to define the robot in ROS 2, while SDF (Simulation Description Format) provides a more detailed description for the Gazebo simulator. These files specify the shape, movements, joints, and sensors, making the simulation more accurate. For the robot's movement and interaction with hardware and the environment, the Wood procedure is a modern approach to controlling robots. They provide a scalable technique for dealing with robotic behavior in an prepared and efficient manner, permitting flexible and reusable selection-making. All these additives are then blended to create a entire simulation of a mobile robot. The robot's motion, interaction with the environment, and execution of obligations are tested in a digital environment that accurately mimics real-world situations. This improves the performance of sensible structures and reduces the fees associated with actual-global checking out

3.2 Professional Installation of Ubuntu 22.04

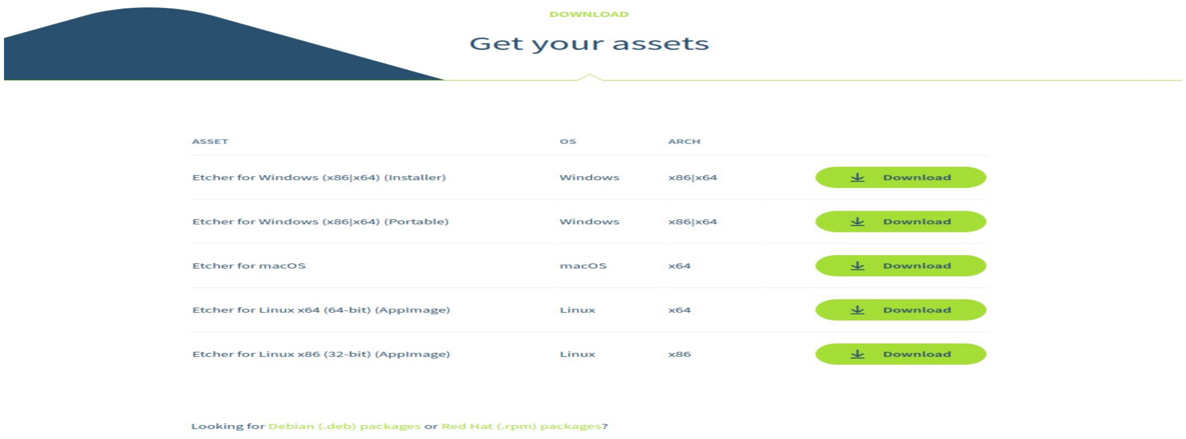
Installing Ubuntu 22.04 is an important step in growing an integrated development environment for robotics simulation. This manual affords an easy way to put in the machine**.**

3.2.1 Prerequisites

Befor.e starting, ensure you have the following:
A compatible computer (preferably with a modern processor and at least **4GB RAM**).
A USB drive (8GB or more) to create a bootable installation disk.
An internet connection to download the ISO file and necessary updates.
Backup of important data if you plan to install Ubuntu alongside another OS.
Download the official Ubuntu 22.04 LTS release from the official
[40]

3.2.2 Creating a Bootable USB Drive

To create a bootable USB drive, we use the following tool: balenaEtcher, which is available in the following versions: (Windows/Mac/Linux) [41]


DOWNLOAD

Get your assets

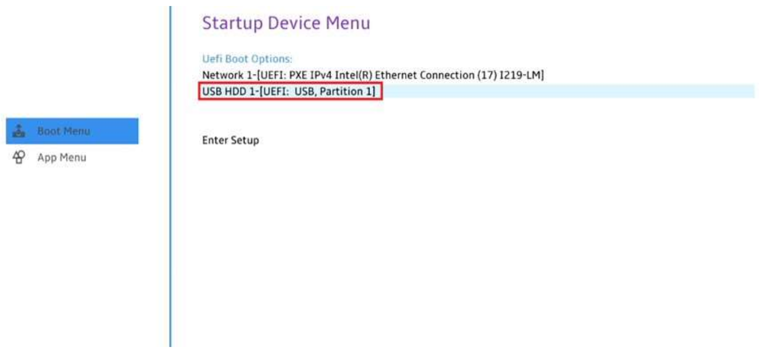
ASSET	OS	ARCH	
Etcher for Windows (x86|x64) (Installer)	Windows	x86|x64	Download
Etcher for Windows (x86|x64) (Portable)	Windows	x86|x64	Download
Etcher for macOS	macOS	x64	Download
Etcher for Linux x64 (64-bit) (AppImage)	Linux	x64	Download
Etcher for Linux x86 (32-bit) (AppImage)	Linux	x86	Download

Looking for Debian (.deb) packages or Red Hat (.rpm) packages?

Figure 3-2-1:Escher program distributions for different operating systems

This image(3-2-1) shows the available Escher download options for the main operating systems (Windows, macOS, Linux), indicating the installable and portable versions. It also shows the classification by infrastructure type (32-bit and 64-bit), and indicates the availability of additional options, including Debian (.Deb) and Red Hat (.Rpm) packages.

Create a bootable USB using Rufus (Windows): This picture(3-2-2) indicates the UEFI boot settings interface, displaying the available boot options: Network Boot [UEFI: PXE IPv4 Intel(R) Ethernet Connection] or USB Boot [UEFI: USB, Partition 1]

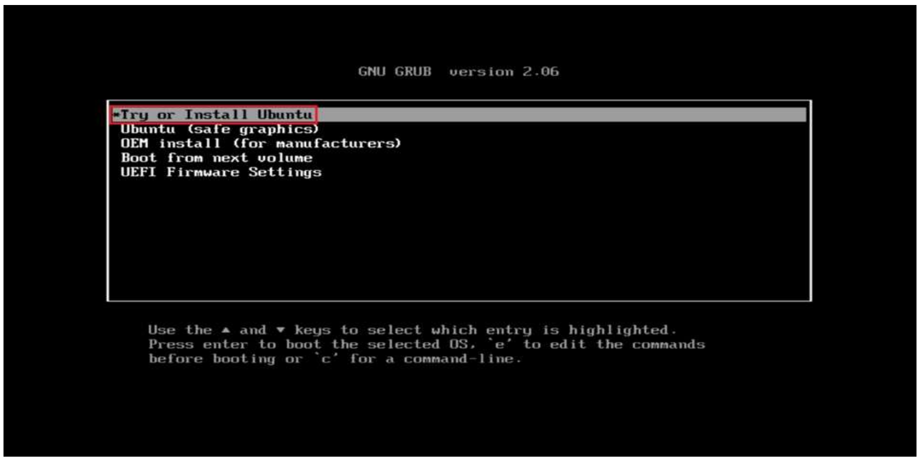


```
Startup Device Menu

Uefi Boot Options:
Network 1-[UEFI: PXE IPv4 Intel(R) Ethernet Connection (17) I219-LM]
USB HDD 1-[UEFI:  USB, Partition 1]

Enter Setup
```

Figure 3-2-2: Boot Settings Menu and Boot Options

3.2.3 Booting from USB and Installing Ubuntu

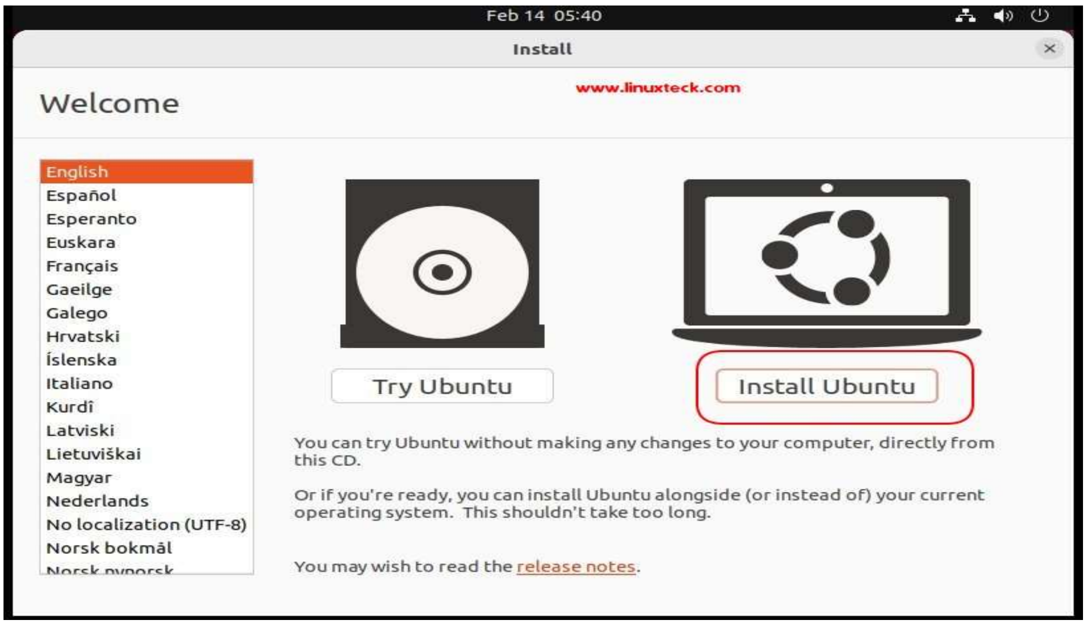
Step 1 To enter BIOS and enable USB booting, reboot your computer and press F2, F12, ESC, or DEL to enter BIOS/UEFI**.** This screen displays the Ubuntu boot menu using GRUB 2.06



```
GNU GRUB  version 2.06

*Try or Install Ubuntu
 Ubuntu (safe graphics)
 OEM install (for manufacturers)
 Boot from next volume
 UEFI Firmware Settings

   Use the ▲ and ▼ keys to select which entry is highlighted.
   Press enter to boot the selected OS, `e' to edit the commands
   before booting or `c' for a command-line.
```

Figure 3-2-3: the GRUB main boot menu that appears when the computer starts, displaying the options

Step 2 When your computer begins, choose Try Ubuntu (to test it with out set up) or Install Ubuntu to proceed with the installation.



You can try Ubuntu without making any changes to your computer, directly from this CD.

Or if you're ready, you can install Ubuntu alongside (or instead of) your current operating system. This shouldn't take too long.

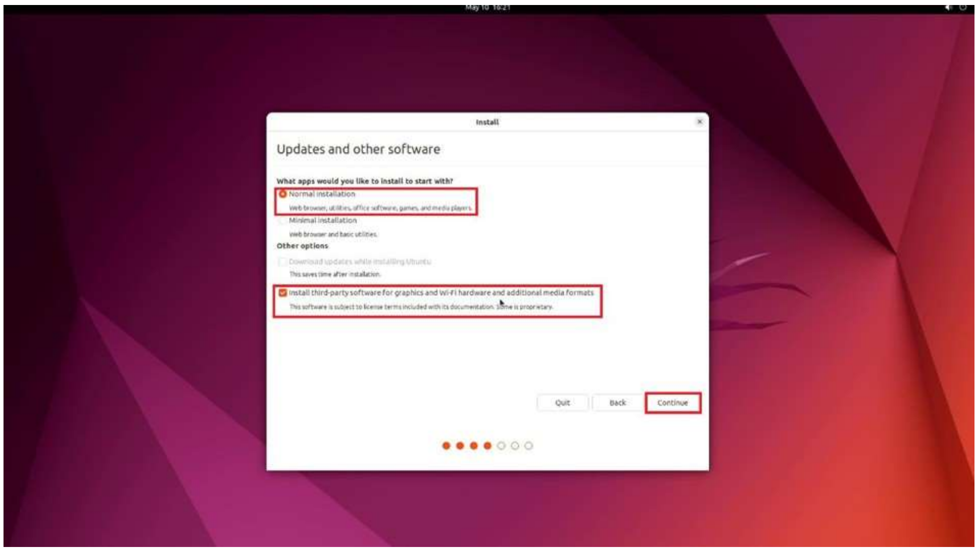
You may wish to read the release notes.

Figure 3-2-4 the opening screen for installing Ubuntu.

This interface is the basic starting point for anyone who wants to try or install Ubuntu, and it offers multiple options to suit people's specific needs. [40]

- **Step 3:**

At this stage, you can choose between a "Normal Installation," which provides a complete laptop environment with essential software such as a browser and desktop suite, and a "Simple Installation," which includes only the essential packages. A normal installation is recommended for the full Ubuntu experience. It's also recommended to enable the option to download updates and third-party software if you have a good internet connection.



Updates and other software

What apps would you like to install to start with?

[✓ Normal installation]
Web browser, utilities, office software, games, and media players.

Minimal installation
Web browser and basic utilities.

Other options

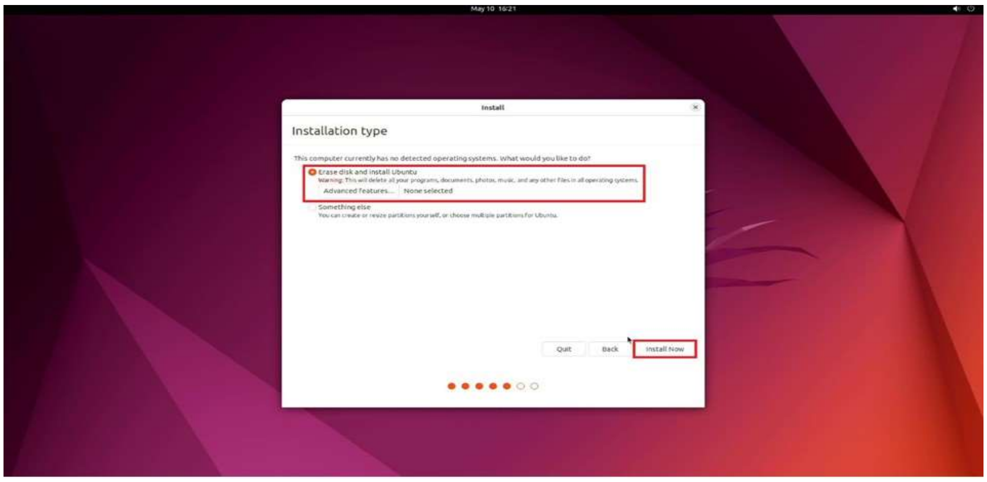
Download updates while installing Ubuntu
This saves time after installation.

[✓ Install third-party software for graphics and Wi-Fi hardware and additional media formats]
This software is subject to license terms included with its documentation. Some is proprietary.

Figure 3-2-5: the software package and update selection screen during the installation process.

- **Step 4:**

Choosing the Installation Type in Ubuntu The "Erase disk and installation Ubuntu" option offers an automated disk partitioning technique, where all current data is deleted and the Ubuntu device is set up without delay. It is supposed for novices, with a strong recommendation to create a backup ahead. On the other hand, the "Something else" alternative allows users to manually partition and personalize the disk, making it appropriate for intermediate or superior customers. In this sensible work, the first option, "Erase disk and set up Ubuntu," become selected to simplify the installation manner and keep away from capacity partitioning mistakes.. [40]



Installation type

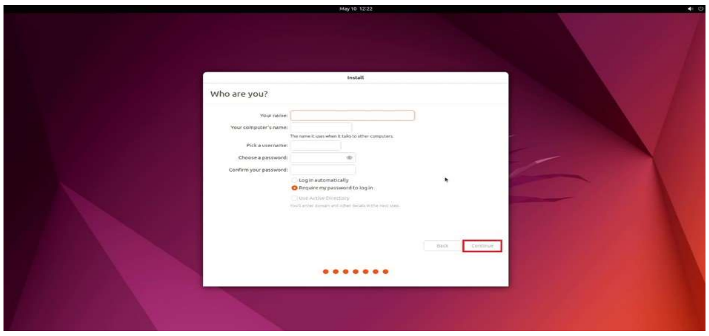
This computer currently has no detected operating systems. What would you like to do?

[✓ Erase disk and install Ubuntu]
Warning: This will delete all your programs, documents, photos, music, and any other files in all operating systems.
Advanced features... None selected

Something else
You can create or resize partitions yourself, or choose multiple partitions for Ubuntu.

Figure 3-2-6: the basic installation type selection screen when installing Ubuntu .

- **Step 5:** This interface is a crucial stage in the installation process where the main user identity is established and settings are set.



Who are you?

Your name:
Your computer's name:
The name it uses when it talks to other computers.
Pick a username:
Choose a password:
Confirm your password:
Log in automatically
[✓ Require my password to log in]
Use Active Directory
You'll enter domain and other details in the next step.

Figure 3-2-7: The user account creation screen during system installation

- **Step 6** : These interfaces are considered essential components of the Ubuntu system, as they represent the first entry point to the system (the login screen) and the main center for managing tasks and applications (the activities interface.([40]

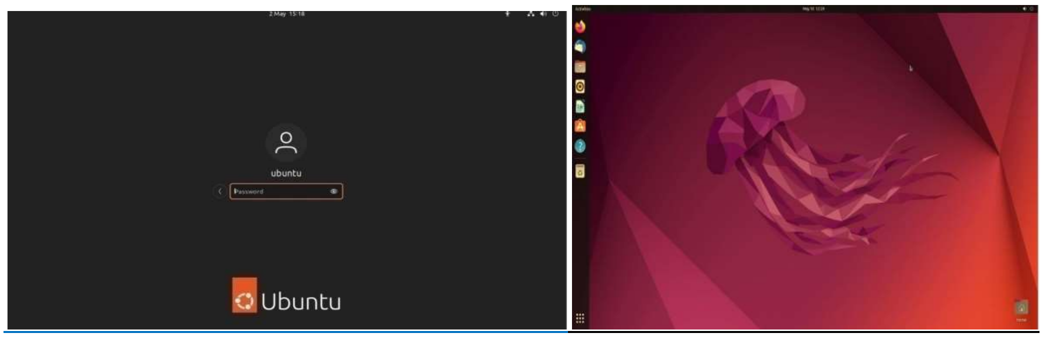


Figure 3-2-9:Ubuntu login interface and operating system.

3.2.4 Install essential software VS Code

3.2.4.1 Install VS Code

We must have a development environment that matches the explanation, so we will download a specific version of Visual Studio.

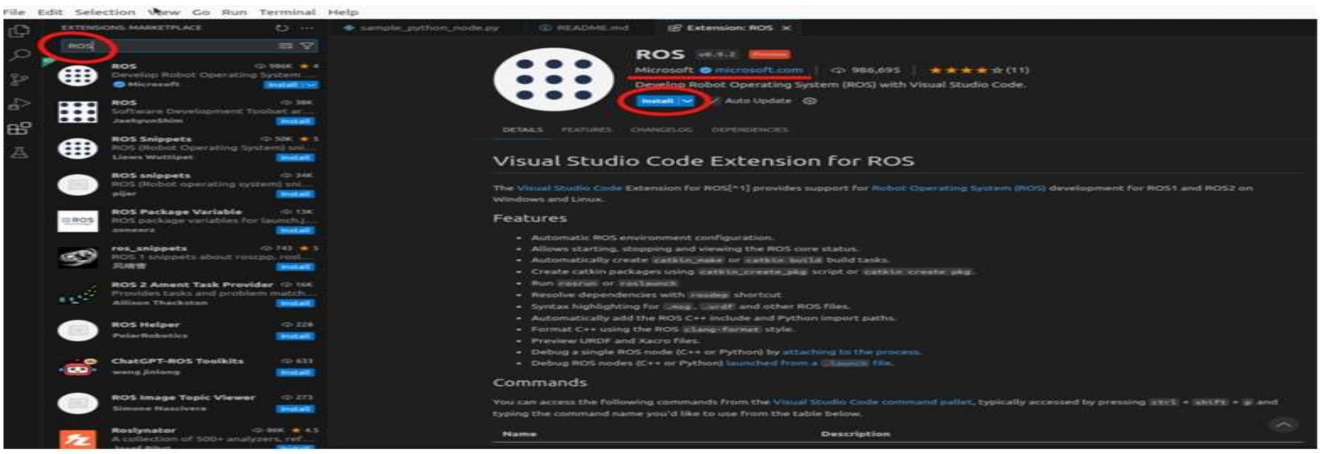
```
cd ~mkdir ros2_workspace_vscode
cd ros2_workspace_vscode
wget -O code_1.93.0-stable_amd64.deb https://update.code.visualstudio.com/1.93.0/linux-deb-
x64/stable
sudodpkg -i code_1.93.0-stable_amd64.deb
```

The cd ~ command is used to navigate to the user's home directory, and a new folder named ros2_workspace_vscode is created with the mkdir command. We then access this folder using cd . The VS Code installation file with the specific version is loaded with wget -O and finally installed with sudodpkg -i as administrator.

3.2.4.2 Add ROS plugin to VS Code

Open the Extensions window using the shortcut: CTRL + SHIFT + X

Search for "ROS" and click "Install." Please choose your Microsoft add-on carefully. There are many other add-ons with the same name, but developed by others.



ROS
Microsoft microsoft.com | 986,695 | ★★★★☆ (11)
Develop Robot Operating System (ROS) with Visual Studio Code.

Visual Studio Code Extension for ROS

The Visual Studio Code Extension for ROS[^1] provides support for Robot Operating System (ROS) development for ROS1 and ROS2 on Windows and Linux.

Features

- Automatic ROS environment configuration.
- Allows starting, stopping and viewing the ROS core status.
- Automatically create catkin_make or catkin build build tasks.
- Create catkin packages using catkin_create_pkg script or catkin create pkg.
- Run rosrun or roslaunch
- Resolve dependencies with rosdep shortcut
- Syntax highlighting for .msg, .urdf and other ROS files.
- Automatically add the ROS C++ include and Python import paths.
- Format C++ using the ROS clang-format style.
- Preview URDF and Xacro files.
- Debug a single ROS node (C++ or Python) by attaching to the process.
- Debug ROS nodes (C++ or Python) launched from a .launch file.

Commands

You can access the following commands from the Visual Studio Code command pallet, typically accessed by pressing ctrl + shift + p and typing the command name you'd like to use from the table below.

Figure 3-2-10 basic information about the ROS add-on provided by Microsoft

3.3 Installing ROS 2 Humble on Ubuntu 22.04

ROS 2 (Robot Operating System) Humble Hawksbill is one of the today's stable releases, presenting a powerful framework for robotics development. Before installing ROS 2, make sure you have:

Ubuntu 22.04 (LTS) installed and updated.
Admin (sudo) privileges for package installation.

3.3.1 Set Up the Sources List

These commands help set up your system to install additional software easily First command does two things at once - it updates your package lists, then installs an essential tool that lets you add new software sources. Second command adds the universe repository which contains tons of extra software you might need. The last command updates everything again so all the changes take effect and your system can see all the new available software. Now your machine [42] is ready to install whatever extra programs you want

```
$ sudo apt install software-properties-common
$ sudo add-apt-repository universe
$sudo apt update
```

3.3.3 Add Humble repository to the sources list:

This command quietly configures your system to access the official ROS 2 software repository by automatically creating a secure connection file tailored to your Ubuntu version, using pre-installed encryption keys to verify all packages while running silently in the background (only showing errors if any occur), essentially enabling safe installation and updates of authentic ROS 2 software directly from the developers' servers with built-in security checks for every package.

```
echo"deb[signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]  http://packages.ros.org/ros2/ubuntu
$(lsb_release -cs) main" |
 sudotee /etc/apt/sources.list.d/ros2.list > /dev/null
```

3.3.4 Install ROS 2 Humble:

This command first updates your package lists to ensure you get the latest versions, then automatically installs the complete ROS 2 Humble desktop package (including all core tools, simulators, and GUI applications) without requiring manual confirmations, giving you a ready-to-use ROS 2 development environment with everything needed to start building and running robotics applications right away [43]

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

3.3.5 Set up the Environment

To use ROS properly, the environment configuration must be loaded every time a terminal window is opened by executing the following command:

```
source /opt/ros/humble/setup.bash
```

To avoid manually executing this command each time, you can add it to the .bashrc file so that it is automatically checked when a new terminal session is started. This is done with the following two commands:

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

This approach ensures that the ROS environment is automatically initialized every time a new terminal window is started.

3.3.5 Testing ROS 2 Installation

Let's verify everything is working with some simple demos

3.3.5.1 Start the C++Talker

In the first terminal, post messages every second

```
ros2 run demo_nodes_cpp talker
[ INFO] [1659572397.676857656] [talker]: Publishing: 'Hello World: 1'
[INFO] [1659572398.076482393] [talker]: Publishing: 'Hello World: 2
[INFO] [1659572399.077283784] [talker]: Publishing: 'Hello World: 3
```

3.3.5.2 Run the Python Listener

In a second terminal: It should echo the messages:

```
$ ros2  run demo_nodes py listener
[INFO] [1659572582.135432432] [Listener]: I heard: [Hello World: 106]
[INFO] [1659572563.860000028] [listener]: I heard: [Hello World: 167]
[INFO] [1659572564.868385852] [listener]: I heard: [Hello World: 106]
```

3.4 Installing Gazebo and Ensuring Compatibility with ROS 2 Humble

GAZEBO

3.4.1 Verifying ROS 2 and Gazebo Compatibility

ROS 2 documentation recommends the usage of Gazebo Fortress as the well matched model with ROS 2 Humble. However, Gazebo Harmonic can also be used with caution.

ROS 2 Version	Compatible Gazebo Version
Humble	Gazebo Fortress (Recommended)
Rolling	Gazebo Harmonic (Use with Caution)

Table 3ROS 2 and Gazebo Compatibility

3.4.2 Installing Gazebo Fortress

3.4.2.1 Install some necessary tools:

This command installs the lsb-release (for system version identification) and gnupg (for cryptographic security and software validation) packages as prerequisites before installing any additional software on an Ubuntu/Debian system. [44]

```
sudo apt-get update
sudo apt-get install lsb-release gnupg
```

3.4.2.2 Install Ignition Fortress:

This command adds the official Ignition Fortress repository, then installs the emulator package with all its dependencies after updating the package lists. The encryption key is first imported, the source is prepared, and the system is updated before installation. [44]

```
sudo curl https://packages.osrfoundation.org/gazebo.gpg --output /usr/share/keyrings/pkgs-osrf-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/pkgs-osrf-archive-keyring.gpg] http://packages.osrfoundation.org/gazebo/ubuntu-stable $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/gazebo-stable.list> /dev/null
sudo apt-get update
sudo apt-get install ignition-fortress
```

3.4.3 Verifying Installation

To launch Gazebo:

```
gazebo
```

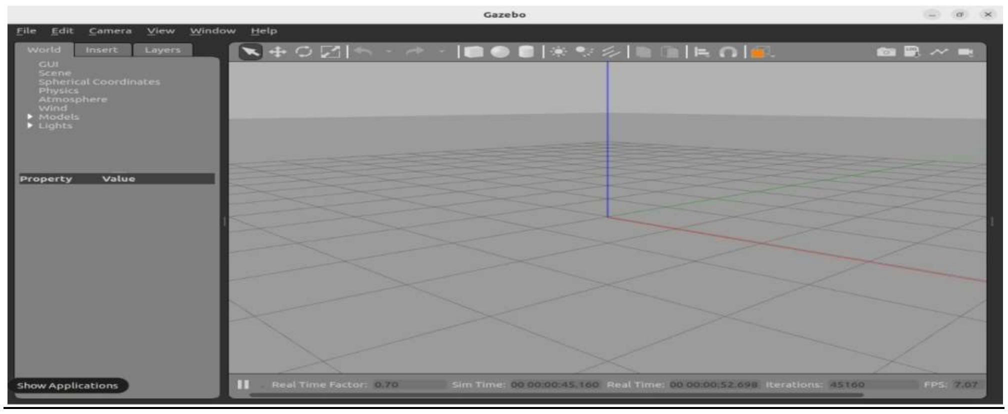


Figure 3-4-1: the Gazebo emulator interface.

3.5 Installing Blender on Ubuntu 22.04

3.5.1. Introduction

Blender is a effective open-source three-D modeling software program extensively used for animation, rendering, and robotics simulation in Gazebo. This manual gives step-by means of-step commands to install Blender on Ubuntu 22.04

3.5.2 Installation Methods

There are two primary ways to install Blender on Ubuntu:

3.5.2.1 Install Blender via APT (Recommended)

The easiest and most stable way to install Blender is through the **APT package manager**. [45]

- **Install Blender**:

```
sudo apt update

sudo apt install  blender
```

3.5.3 Verify Blender Installation:

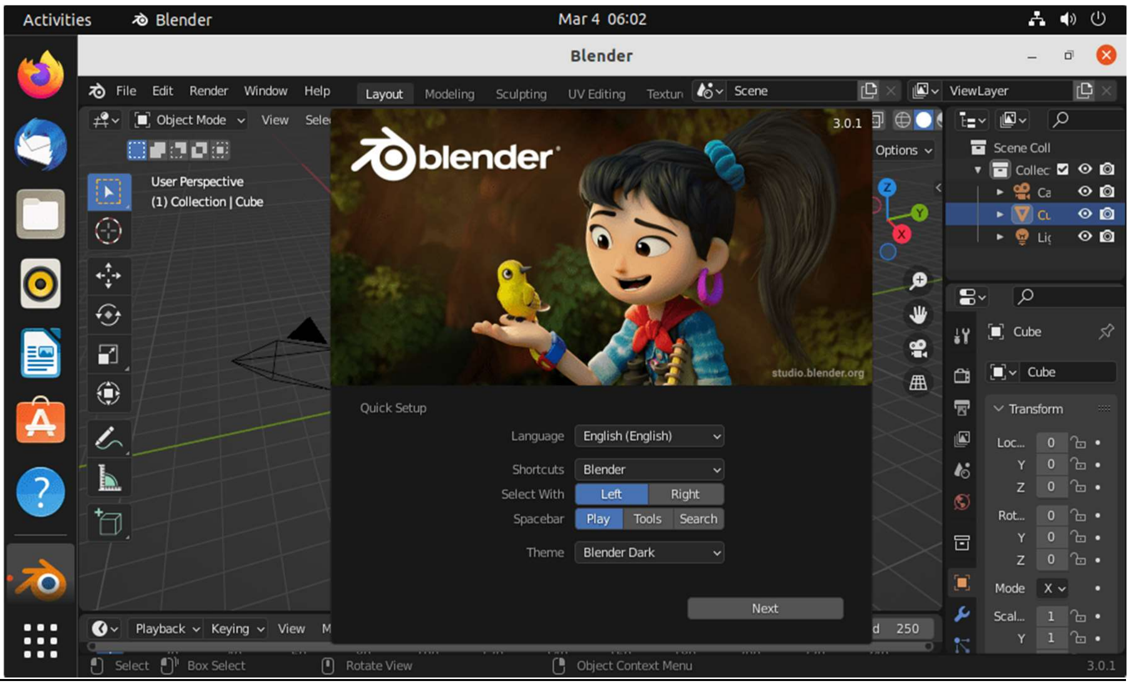
After installation, Blender can be verified by running

```
Blender
```

This command will launch Blender if the installation was successful.

ExpectedOutput:

Blender should launch with its graphical user interface.



Quick Setup
Language: English (English)
Shortcuts: Blender
Select With: Left | Right
Spacebar: Play | Tools | Search
Theme: Blender Dark
Next

Figure 3-5-1 **blender interface**

3.6 General Description of the Turtlebot3_ws_bt Environment

The turtlebot3_ws_bt environment is an integrated system designed within the ROS 2 framework. It aims to develop an advanced simulation of the TurtleBot3 robot, enabling autonomous navigation and behavior control using a behavior tree. This environment is organized according to a central structure, consisting of a main folder containing a subfolder, src/, which contains three main packages developed and integrated into the simulation and visualization environment using standard ROS 2 tools such as Gazebo and RViz.

3.6.1. Package 1: turtlebot3_gazebo

This package configures the physical simulation environment using the Gazebo simulator. It contains launch files necessary to set up the simulation, including:
robot_state_publisher.launch.py: Responsible for publishing the robot's kinematic changes using sdf data.
spawn_turtlebot3.launch.py: Inserts the robot model into the simulation environment.
turtlebot3_world.launch.py: Assembles the simulation components, including the model and map, to run the entire environment in Gazebo.

3.6.2. Package 2: tb3_sim

This package configures the robot's autonomous navigation environment through a set of launch files that activate the following nodes:
amcl: An iterative localization algorithm using sensors.
map_server: Provides a map of the environment.
planner_server and controller_server: For path planning and execution via the Nav2 navigation system.
nav2.launch.py: Assembles and launches these nodes together.
turtlebot3_world.launch.py: Reinitializes the simulation environment to match the autonomous navigation settings.
It also includes a Python script titled **set_init_amcl_pose.py**, which is used to set the robot's initial pose within the AMCL algorithm, contributing to precise localization.

3.6.3. Package 3: tb3_autonomy

This package represents the advanced level of robot behavioral control. It relies on the Behavior Tree technique to define the sequence of action nodes and condition nodes. This package includes:
The **tree.xml file:** defines the structure of the behavior tree.
The **autonomy.launch.py** launch file: launches the master node, autonomy_node, which implements the tree using the BehaviorTree.CPP library.

The **navigation_behaviors.cpp** file: contains the implementation of custom behaviors used as implementation nodes within the tree to control the robot's movements and make decisions based on its state and surrounding environment.

3.6.4 General Integration

These additives are integrated within the ROS 2 architecture, which is based on message exchange between nodes. This enables bodily simulations to be activated via Gazebo, with a live display of the robotic's function, nation, and trajectory in the RViz display device. This organizational structure displays a systematic method to developing an self reliant robotic machine capable of intelligently interacting with dynamic environments, with the aid of integrating simulation equipment, localization and navigation algorithms, and behavioral manipulate mechanisms within a unified, scalable platform.

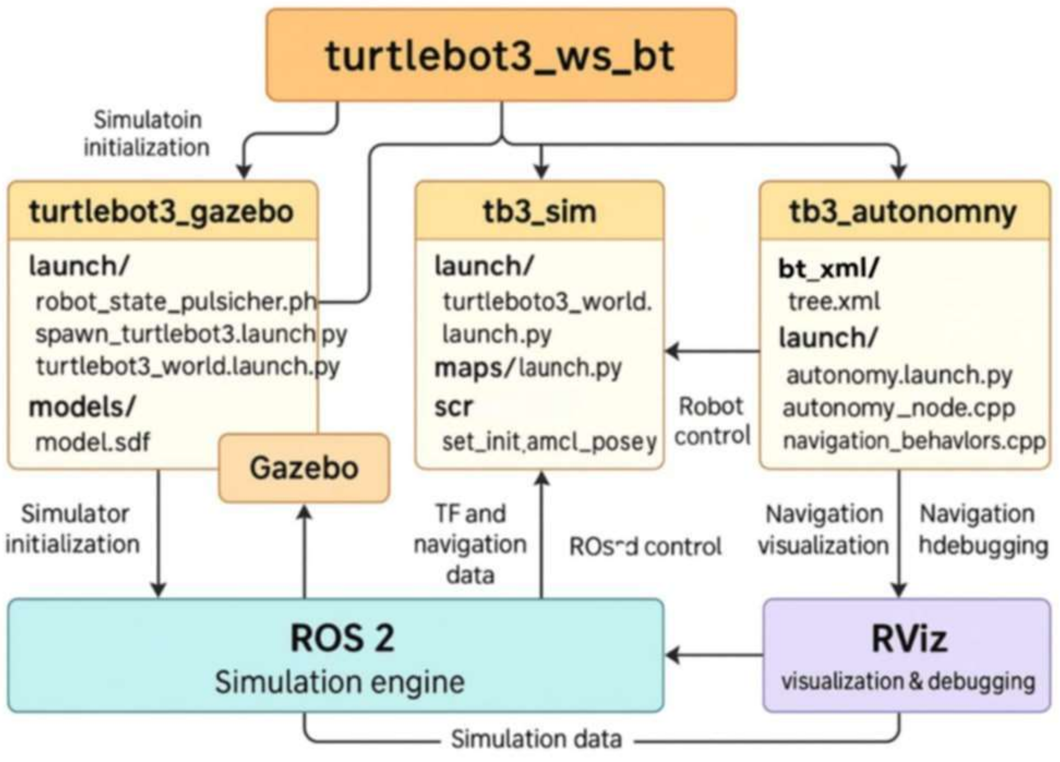


Figure 3.6.1 Structuring a TurtleBot3 project in ROS 2 with Gazebo simulation and autonomous navigation

3.7 Creating a ROS 2 Package

In ROS 2 structures, a workspace represents the infrastructure that includes all of the programs had to develop, take a look at, and simulate robotic applications. It is a first-rate folder organized in a way that permits ROS 2 tools to understand, construct, and run the programs inside it seamlessly. In the context of simulating a mobile robot, a workspace is the place to begin for assembling the diverse software additives into a potential, included environment. In this task, we can create a workspace for simulating a TurtleBot3 cellular robotic, named turtlebot3_ws_bt. Within this space, we are able to create 3 sub-packages:

A Python package deal: This is used to area the robotic within the initial role in the simulation environment, i.E., it initializes the robot's function at startup.
A C++ package deal named tb3_autonomy: This package deal is answerable for implementing the robotics' behaviors the usage of a conduct tree, permitting the robot to act dynamically in keeping with its surrounding environment.
A C++ package referred to as turtlebot3: used to address SDF (Simulation Description Format) files, which describe the geometry, sensors, and actuators within the simulation surroundings. [46]

3.7.1 Install the necessary development tools

The tools must be installed to facilitate working with the software packages.

- **Development tools C++:** These commands are used to compile code using colcon and create binary files.

```
sudo apt install build-essential
```

- **Development tools python:** These commands facilitate and add a number of useful additions to the Colcon build tool, such as support for automatic building of Python packages, detailed reports on builds, and other features that speed up the development process. [46]

```
sudo apt install python3-colcon-common-extensions python3-pip
```

3.7.2 Create a Workspace:

We first create a workspace folder, then navigate to the src subfolder dedicated to creating packages inside it:

```
mkdir -p ~/ turtlebot3_ws_bt /src
cd ~/turtlebot3_ws_bt
```

This creates a new home folder called turtlebot3_ws_bt containing a src subfolder, which is where all packages for the project are created.

3.7.2.2 Build a workspace:

Although the workspace does not contain any programs at this stage, it is nice to carry out the preliminary construct the usage of the default ROS 2 construct device, colcon. This step configures the infrastructure, creates the vital construct directories, and guarantees that the environment is well configured.

```
cd ~/ turtlebot3_ws_bt
colcon build
```

3.7.2.3 Activate the workspace:

After successful construction, we activate the workspace:

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

3.7.3 Creating a Python package:

In ROS 2, all packages must be created within a validated workspace. In this project, we have already created the workspace named turtlebot3_ws_bt, and now we will create a ROS 2 package in Python within this path, with the goal of writing a node

```
$ cd ~/ turtlebot3_ws_bt/src/
```

Take a look at how to build a package:This command creates a new ROS2 Python package with the essential dependencies (rclpy and std_msgs) specified, and then displays the contents of the created directory to examine the package structure

```
ros2 pkg create tb3_sim --build-type ament_python --dependencies rclpy std_msgs geometry_msgs
transforms3d
```

3.7.3.1 Contents of the my_turtlesim package in ROS 2

After creating a ROS 2 package named tb3_sim in the turtlebot3_ws_bt workspace, we create extra folders in the package deal to organize the special documents in step with their capabilities. This company enables facilitate challenge management and increases clarity at some point of development. The following instructions are used to create those folders**:**

```
cd  tb3_sim
 mkdir -p tb3_sim
mkdir -p launch
mkdir -p maps
mkdir -p config
ls
package.xml  resource  setup.cfg  setup.py   launch  maps  config
```

- **Package.xml :**

The package deal.Xml file is a key file used to outline a ROS 2 bundle and record its metadata. This document contains statistics together with the bundle call, model, short description, developer name, electronic mail, license, and the dependencies required to run the bundle. This record is study with the aid of the colcon build machine and different ROS tools to decide how the package ought to be built and run. It is also used to percentage packages across systems or in reliable repositories.

- **Resource :**

The resource/ folder is used to outline a package-precise useful resource, an empty file with the identical name because the bundle. This record is important for the set up machine to apprehend the bundle and link it to the best paths at some stage in set up. Although it does now not include executable code, its presence is important as part of the package deal shape and serves as an internal reference for identifying package deal-particular resources.

- **Setup.Cfg :**

This file is used to configure the bundle's set up settings the usage of setuptools. It commonly includes a [develop] section to specify the direction to hyperlink to whilst putting in the bundle in development mode, and an [install] segment to specify the statistics to put in. In the ROS 2 context, this report works together with Setup.Py to specify how executables, together with launch and configuration files, are established**.**

- **Setup.Py :**

Setup.Py is the primary record for installing a Python package in a ROS 2 surroundings. It specifies the package deal call, model, executables (console_scripts), related records (which include launch and config documents), and required dependencies. This report is known as inside the Colcon build manner and compiled into an executable package in the ROS 2 construct. It is likewise used to specify where configuration documents and maps are placed inside the installed report system**.**

- **Release:**

The launch/ folder is used to save launch documents that manage the operation of the package's diverse nodes. These files are commonly written in Python (.Py) or XML (.Launch.Xml). Launch documents permit more than one nodes to be combined and run inside the preferred sequence. They also can skip parameters and specify configuration settings. In the case of a cellular robot simulation, these documents are used to release the conduct tree, deploy the initial position, release navigation nodes, and extra.

- **Maps:**

This folder includes the environment maps utilized by the robot for navigation and positioning. It usually consists of .Pgm files (surroundings bitmaps) and an associated .Yaml record, which specifies the map's location in space and the desired coordinates. These maps are utilized by AMCL or Nav2 systems for self-positioning. These maps are typically loaded in the release files.

- **Config:**

This folder includes package deal-precise configuration documents, which include .Yaml documents used to configure navigation or positioning algorithms (e.G., places.Yaml). These documents can incorporate parameters passed to nodes, including AMCL or DWB Planner settings. This folder is critical for separating facts from code, making it less complicated to alter and configure without rebuilding the package.

- **tb3_sim/ tb3_sim/:**

This is a subfile within the package that takes the package name. It serves as the source directory and contains the Python files that implement the nodes**.**

3.7.4 Creating a C++ Package

Creating a C++ package is very just like developing a Python package deal, but there are a few differences in structure and content due to one-of-a-kind improvement environments and construct requirements. To get started, first navigate in your workspace folder after which create the brand new package the usage of the proper ROS 2 gear. We'll adopt the equal naming sample because the Python bundle, naming the package deal tb3_autonomy, and specifying the build type as ament_cmake, which is the construct kind for C packages in ROS 2.

```
cd ~/turtlebot3_ws_bt/src
ros2 pkg create tb3_sim --build-type ament_cmake --dependencies rclcpp std_msgs
geometry_msgs behaviortree_cpp_v3 yaml-cpp rclcpp_action nav2_msgs tf2
tf2_geometry_msgs
```

3.7.4.1 Explaining the role of each file or folder:

```
cd ~/ turtlebot3_ws_bt /src/ tb3_autonomy
ls
CMakeLists.txt   Include  package.xml  src
```

- **CMakeLists.Txt:**

This report specifies how to build a bundle the usage of the ament_cmake build machine. It carries the instructions for compiling C ++ documents, creating libraries or executable nodes, and specifying package deal dependencies. In the case of the tb3_autonomy package deal, this file is changed to outline the autonomy_node.Cpp node, specify the location of the header documents, and consist of dependencies together with rclcpp and behaviortree_cpp_v3.

- **Include folder:**

In C++ projects, it is good practice to separate class and function definitions (headers) from their implementations (implementations). Header files with the .hpp or .h extension are placed in the include/ folder, while implementations are placed in .cpp files in the src/ folder. In our example, the include/tb3_autonomy/autonomy_node.h file is created to include the AutonomyNode class definition.

- **Package.Xml:**

This file is essential for every ROS 2 package. It contains metadata about the package, such as its name, version, author, and license, as well as identifying dependencies on other packages. In tb3_autonomy, you must ensure that the file contains dependencies such as rclcpp, std_msgs, and ament_cmake, in addition to any other dependencies associated with the behavior tree.

- **Src folder:**

This folder represents the place of the package deal's supply files. It's wherein the node logic is written and carried out the use of .Cpp files. In our case, the autonomy_node.Cpp report is positioned in src/ and incorporates the AutonomyNode class definition that creates the behavior tree and updates it periodically.

3.7.5 Building a simple system using ROS 2

Here, we build a simple autonomous mobile robot control system based on the ROS 2 environment.

The syste misbased on
the idea of work distribution between two standalone nodes executing together: one written in C++ to control the robot behavior using a behavior tree, and the other written in Python
to provide the initial position needed to execute the Automated Local Positioning (AMCL) system.

3.7.5.1 Create a C++ node that controls turtlebot3 based on BT file

- **Step 1**:

To implement a simple, distributed control system for a mobile robot in the ROS 2 world, a standalone C++ node is created to specify the behavior of the robot through a behavior tree. The node is created in the tb3_autonomy package, which is built under the ament_cmake build system, and source files are placed in a special path under the src directory.

```
cd ~/ turtlebot3_ws_bt /src/ tb3_autonomy /src
nano autonomy_node.cpp
```

- **Step 2**:

Writing the code

The presented code represents a ROS 2 C++ node known as AutonomyNode, used to run a conduct tree aimed toward guiding a robotic to a selected area. Upon startup, the node masses an

XML conduct tree document from the tb3_autonomy package folder and registers a custom conduct node referred to as GoToPose answerable for moving the robot. A timer is then installation to periodically (each 500 milliseconds) update the conduct tree through executing tickRoot. If the task succeeds, a message is printed indicating the crowning glory of the navigation; if it fails, the timer is canceled and execution is halted. The code is based on the BehaviorTree.CPP library to manage the behavior tree and demonstrates the combination of the ROS 2 structure with the behavior tree in a structured and scalable manner for self sufficient robotics applications.

```
#include "autonomy_node.h"
using namespace std::chrono_literals;
const std::string bt_xml_dir =
   ament_index_cpp::get_package_share_directory("tb3_autonomy") + "/bt_xml";
AutonomyNode::AutonomyNode(const std::string &nodeName) : Node(nodeName)
{
 this->declare_parameter("location_file","none");
 RCLCPP_INFO(get_logger(), "Init done");
}
void AutonomyNode::setup()
{
 RCLCPP_INFO(get_logger(), "Setting up");
 create_behavior_tree();
 RCLCPP_INFO(get_logger(), "BT created");
 const auto timer_period = 500ms;
 timer_ = this->create_wall_timer( timer_period,
    std::bind(&AutonomyNode::update_behavior_tree, this));
 rclcpp::spin(shared_from_this());
 rclcpp::shutdown();
}
void AutonomyNode::create_behavior_tree()
{
 BT::BehaviorTreeFactory factory;
 // register bt node
 BT::NodeBuilder builder =
    [=](const std::string &name, const BT::NodeConfiguration &config) {
   return std::make_unique<GoToPose>(name, config, shared_from_this());};
 factory.registerBuilder<GoToPose>("GoToPose", builder);
 RCLCPP_INFO(get_logger(), bt_xml_dir.c_str());
 tree_ = factory.createTreeFromFile(bt_xml_dir + "/tree.xml");
 RCLCPP_INFO(get_logger(), "3");}
void AutonomyNode::update_behavior_tree()
{
 BT::NodeStatus tree_status = tree_.tickRoot();
 if (tree_status == BT::NodeStatus::RUNNING)
 {return; }
 else if (tree_status == BT::NodeStatus::SUCCESS)
 { RCLCPP_INFO(this->get_logger(), "Finished Navigation"); }
 else if (tree_status == BT::NodeStatus::FAILURE){
   RCLCPP_INFO(this->get_logger(), "Navigation Failed"); timer_->cancel(); }}
int main(int argc, char **argv)
{rclcpp::init(argc, argv);
```

```
  auto node = std::make_shared<AutonomyNode>("autonomy_node");
  node->setup();
  return 0;}
```

- **Step 3:**

Open the CMakeLists.txt file inside the **tb3_autonomy** and **package.xml** package folder using the command:

```
cd ~/ turtlebot3_ws_bt /src/ tb3_autonomy
nano CMakeLists.txt
```

CMakeLists.txt file This file, CMakeLists.txt, represents the basic structure for building a ROS 2 C++ package known as tb3_autonomy. The process begins by specifying the minimum required CMake version and then declaring the project name. Compilation options such as -Wall and -Wextra are enabled to ensure potential warnings are detected during compilation, which helps improve code quality.Next, the packages necessary for building this project are searched, such as rclcpp for programming using the ROS 2 C++ API, rclcpp_action for supporting actions, nav2_msgs for interacting with navigation messages, behaviortree_cpp_v3 for implementing complex behavior trees, yaml-cpp for parsing YAML files, and tf2 and tf2_geometry_msgs for applying transformations in 3D space. The file also includes instructions for installing important files such as behavior tree files (BT XML) and launch files within the package folder upon installation. The source of custom behaviors is specified in the navigation_behaviors.cpp file, which is merged with the main autonomy_node.cpp file to produce the autonomy_node executable.The ament_target_dependencies command is used to add compile-time dependencies to the executable node, and the target_link_libraries command is used to link the YAML library. Installation instructions for the autonomy_node node are included at the end of the file under the appropriate path within the package.
Finally, if the BUILD_TESTING option is enabled, the package is prepared for use by automated quality testing tools using ament_lint_auto. The process concludes with a call to ament_package(), which formally defines the package in the ament build system.

```
cmake_minimum_required(VERSION 3.8)
project(tb3_sim)
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 17)
endif()
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(rclcpp_action REQUIRED)
find_package(nav2_msgs REQUIRED)
find_package(behaviortree_cpp_v3 REQUIRED)
find_package(yaml-cpp REQUIRED)
find_package(tf2 REQUIRED)
find_package(tf2_geometry_msgs REQUIRED)
include_directories(include)
install(DIRECTORY  bt_xml   launch  DESTINATION share/${PROJECT_NAME})
```

```
set(BEHAVIOR_SOURCES src/navigation_behaviors.cpp)
add_executable(autonomy_node src/autonomy_node.cpp ${BEHAVIOR_SOURCES})
set(TARGET_DEPENDS rclcpp behaviortree_cpp_v3 yaml-cpp rclcpp_action nav2_msgs tf2
  tf2_geometry_msgs)
ament_target_dependencies(autonomy_node ${TARGET_DEPENDS}))
target_link_libraries(autonomy_node ${YAML_CPP_LIBRARIES})
install(TARGETS autonomy_node DESTINATION lib/${PROJECT_NAME})
if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
endif()
ament_package()
```

- **Update package.xml**

Open the **package.xml** file inside the **tb3_autonomy** package folder using the command:

```
cd ~/ turtlebot3_ws_bt /src/ tb3_autonomy
nano package.xml
```

The package deal.Xml file is a essential component of ROS 2 programs, containing descriptive and organizational facts that permits the construct machine to system the package deal successfully. In the tb3_autonomy package, the record defines the package deal name, model, protection and license records, and specifies that the build tool is ament_cmake, that's for C programs. The document additionally includes important dependencies including rclcpp for developing C nodes, behaviortree_cpp_v3 for creating conduct bushes that alter robotic behavior, and yaml-cpp for studying YAML configuration documents. The file additionally references non-obligatory dependencies such as rclcpp_action and nav2_msgs, which may additionally later be used with the navigation machine. It additionally includes trying out dependencies for code best warranty, which include ament_lint_auto and ament_lint_common. Finally, the <export> section indicates which build tool is used, making it less complicated for ROS 2 gear to deal with the package deal for the duration of the construct and set up technique.

```
<?xml version="1.0"?>
<?xml-model                                    href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>tb3_autonomy</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="sharadmaheshwari19@gmail.com">sharad</maintainer>
  <license>TODO: License declaration</license>
  <buildtool_depend>ament_cmake</buildtool_depend>
  <depend>rclcpp</depend>
  <!-- <depend>rclcpp_action</depend>
  <depend>nav2_msgs</depend> -->
  <depend>behaviortree_cpp_v3</depend>
  <depend>yaml-cpp</depend>
  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>
  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

3.7.5.2 Create a Python node to configure initial pose

This node plays a key role in the robot's positioning system, publishing a PoseWithCovarianceStamped message to the /initialpose topic. This message contains information about the robot's initial pose and the direction from which it will start within the operating environment, which is what the Adaptive Monte Carlo Localization (AMCL) system needs to initiate

- **Step 1 Create a node file inside the package**

To create the node file, we first created the necessary directories within the tb3_sim package. In ROS 2, it is customary for the package folder to contain an internal folder with the same name, so that the package structure matches the settings in the setup.py file.

```
cd ~/ turtlebot3_ws_bt /src/ tb3_sim / tb3_sim
nano set_init_amcl_pose.py
```

- **Step 2 Explanation of the code for the node**

This code represents a ROS 2 node written in Python used to publish the robot's initial position to the /initialpose topic using a PoseWithCovarianceStamped message. This allows AMCL to determine the robot's location on the map at startup. The node begins by declaring itself as init_amcl_pose_publisher and then declares several parameters (x, y, theta, cov) representing the initial position coordinates, the orientation angle, and the variance associated with the position uncertainty. The node then creates a publisher on the /initialpose topic and waits for an actual subscriber (usually AMCL) to connect before sending the message. After confirming the existence of a subscriber, the node generates a message containing the initial position and transforms the theta angle into a quaternion representation using the transforms3d library to accurately represent the robot's orientation in 3D space. The covariance matrix is also populated to reflect the degree of uncertainty in the position and orientation. Finally, the message is published once when the node starts, allowing the robot's position to be automatically initialized upon takeoff.

```
import time
import rclpy
from rclpy.node import Node
import transforms3d
from geometry_msgs.msg import PoseWithCovarianceStamped
class InitAmclPosePublisher(Node):
  def __init__(self):
    super().__init__("init_amcl_pose_publisher")
    self.declare_parameter("x", value=0.0)
    self.declare_parameter("y", value=0.0)
    self.declare_parameter("theta", value=0.0)
    self.declare_parameter("cov", value=0.5**2)
    self.publisher = self.create_publisher(
        PoseWithCovarianceStamped,
        "/initialpose",
        10, )
    while(self.publisher.get_subscription_count() == 0):
      self.get_logger().info("Waiting for AMCL Initial Pose subscriber")
      time.sleep(1.0)
  def send_init_pose(self):
    x = self.get_parameter("x").value
    y = self.get_parameter("y").value
    theta = self.get_parameter("theta").value
    cov = self.get_parameter("cov").value
    msg = PoseWithCovarianceStamped()
    msg.header.frame_id = "map"
    msg.pose.pose.position.x = x
    msg.pose.pose.position.y = y
    quat = transforms3d.euler.euler2quat(0, 0, theta)
    msg.pose.pose.orientation.w = quat[0]
    msg.pose.pose.orientation.x = quat[1]
    msg.pose.pose.orientation.y = quat[2]
    msg.pose.pose.orientation.z = quat[3]
     msg.pose.covariance = [
        cov, 0.0, 0.0, 0.0, 0.0, 0.0,  # Pos X
        0.0, cov, 0.0, 0.0, 0.0, 0.0,  # Pos Y
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  # Pos Z
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  # Rot X
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0,  # Rot Y
        0.0, 0.0, 0.0, 0.0, 0.0, cov   # Rot Z   ]
    self.publisher.publish(msg)
def main(args=None):
  rclpy.init()
  initAmclPosePublisher = InitAmclPosePublisher()
  future = initAmclPosePublisher.send_init_pose()
  rclpy.spin(initAmclPosePublisher)
  initAmclPosePublisher.destroy_node()
  rclpy.shutdown()
```

Make the file executable

```
chmod +x set_init_amcl_pose.py
```

- **Step 3 Prepare setup.py and package.xml files**

In order to run the node using the ros2 run command, we need to modify the setup.py file to associate the node with an executable name.

```
cd ~/ turtlebot3_ws_bt /src/ tb3_sim
```

```
nano setup.py
```

Add in the entry_points section:

```
 name=package_name,
   version='0.0.0',
   packages=[package_name], data_files=[ ('share/ament_index/resource_index/packages', ['resource/' + 
package_name]),
       ('share/' + package_name, ['package.xml']),
       (os.path.join('share', package_name),
        glob('launch/*launch.[pxy][yma]*')),
       (os.path.join('share', package_name, 'config/'),glob('config/*')),       (os.path.join('share', 
package_name, 'maps/'),
        glob('maps/*')), ],
   install_requires=['setuptools'],
   zip_safe=True,
   maintainer='sharad',
   maintainer_email='sharadmaheshwari19@gmail.com',
   description='TODO: Package description',
   license='TODO: License declaration', tests_require=['pytest'],
   entry_points={'console_scripts': [ 'amcl_init_pose_publisher = 
tb3_sim.set_init_amcl_pose:main',  ],
```

In the setup.Py file, we comprehensively configured the package deal to make sure it installs and runs nicely inside the ROS 2 environment. Important additions protected specifying the package name='tb3_sim' and its preliminary version version='zero.0.0', together with consisting of the inner package listing via applications=[package_name]. Necessary files had been delivered underneath data_files, including the bundle definition files in ament_index, the package deal.Xml file, and the release/, config/, and maps/ directories the use of the os.Course.Be part of and glob() capabilities to dynamically specify documents. To make certain right installation, the developer included the setuptools library underneath install_requires and enabled the set up from zip documents with zip_safe=True. Also, developer identification facts which includes name and e mail had been brought. Most importantly, the developer added an entry factor inside the entry_points segment via the road 'amcl_init_pose_publisher = tb3_sim.Set_init_amcl_pose:predominant', which permits the Python node to be run immediately using the ros2 run tb3_sim amcl_init_pose_publisher command. All of those additions are vital to check in the package with the ROS 2 system and permit its programming nodes to run in a based and preferred way.

- **Update package.xml**

The commands are used to enter the tb3_sim package folder and edit the package.xml file which contains the package information and dependencies required to run properly in ROS2.

```
cd ~/ turtlebot3_ws_bt /src/ tb3_sim
```

```
nano package.xml
```

The package deal.Xml document in the tb3_sim bundle is an crucial aspect of the bundle definition in ROS 2. It specifies general residences which include the package deal call (tb3_sim), model (0.0.0), and outline (incomplete in this case), as well as renovation facts together with the developer's name and e mail. It also shows that the publishing license has no longer yet been decided, so that you can want to be completed later. Programmatically, the package pronounces its operational dependencies through the <depend> tag. It relies at the gazebo_msgs package to have interaction with the Gazebo simulator, and on geometry_msgs to change geometric messages which includes locations and guidelines. For code first-class, the bundle consists of testing dependencies such as ament_copyright, ament_flake8, ament_pep257, and python3-pytest to verify compliance with stylistic standards and software rights. Finally, the build tool kind used in the bundle is described as ament_python below the <export> phase, which instructs build tools like colcon to deal with the package as a Python package for the duration of compilation and set up.

```
<?xml version="1.0"?>
<?xml-model                                   href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>tb3_sim</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="sharadmaheshwari19@gmail.com">sharad</maintainer>
  <license>TODO: License declaration</license>
  <depend>gazebo_msgs</depend>
  <depend>geometry_msgs</depend>
  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>
  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

3.7.5.3 Building a Package

After creating the software nodes of the system, the packages must be built within the ROS 2 environment to be executable. The build is executed through colcon, which is a build tool for ROS 2 packages based on workspace layout. It is very important to ensure that the build commands are executed from the workspace root to avoid configuration and linking problems.

- **Step 1 Building a Package**

Since the system relies on two packages, Python and C++, all packages must be compiled for each change or new node added. To compile all packages, run the following commands from the workspace root:.

```
$ cd ~ turtlebot3_ws_bt / /
$ colcon build
```

- **Step 2 Building selected Packages**

If you want to build only a specific package – such as the C++ tb3_autonomy package – you can use the --packages-select option to specify the target package:

```
cd ~/turtlebot3_ws_bt
colcon build --packages-select tb3_autonomy
source install/setup.bash
```

Run "autonomy_node" node of package "tb3_autonomy":

```
ros2 run tb3_autonomy autonomy_node
[INFO] [autonomy_node]: Init done
[INFO] [autonomy_node]: Setting up
[INFO] [autonomy_node]: /.../tb3_autonomy/bt_xml
[INFO] [autonomy_node]: 3
[INFO] [autonomy_node]: BT created
[INFO] [autonomy_node]: Sent Goal to Nav2
[INFO] [autonomy_node]: [GoToPose] Goal reached
[INFO] [autonomy_node]: Finished Navigation
```

When the autonomy_node grab node is started as a result of executing the ros2 run tb3_autonomy autonomy_node command, the node begins initializing its components. The first message displayed in the terminal is: [INFO] [autonomy_node]: Init completed, indicating that the initial configuration of the node has been completed successfully. This is indicated by the Behavior Tree Setup section, displayed by the message: [INFO] [autonomy_node]: Setting up. The path to the tree's XML file is then displayed: ./tb3_autonomy/bt_xml, which is monitored with the help of an internal numeric message (3), likely used for monitoring and optimization purposes. After the tree is successfully loaded, the message: [INFO] [autonomy_node]: BT created is displayed, confirming that the behavior tree was successfully created. When the GoToPose behavioral node is activated, the Nav2 module sends a navigation goal and end-of-presentation message: [INFO] [autonomy_node]: The goal was sent to Nav2. When the robot reaches the goal coordinates, the node sends two consecutive messages: the first indicates the end of the completed task with [GoToPose], and the second indicates the completion of the navigation task with: [INFO] [autonomy_node]: Navigation completed.These outputs demonstrate the organized order in which the node performs tasks within the behavior tree framework and indicate proper interaction between ROS 2 components, such as the Nav2 navigation module and the specialized GoToPose node. These messages enable developers to track device reputation and diagnose capacity issues efficiently and effectively.

3.8 Describing the Robot Using SDF in ROS 2 and Gazebo

3.8.1 Introduction

A crucial step in designing and simulating a robot is developing a virtual model that may be programmed, tested, and manipulated in a controlled virtual environment. two fundamental formats for describing a robot in ROS 2 and Gazebo are: URDF (Unified Robot Description Format) and SDF (Simulation Description Format). These formats allow the robot's structure and behavior to be defined in a manner that can be simulated and visualized.
URDF is mostly used in the ROS 2 environment to specify the mechanical components of the robot while SDF is utilized in the Gazebo simulation environment for augmented physics and sensor details as well as more realistic environmental modeling.

3.8.2 Introduction to the SDF File

The SDF (Simulation Description Format) file is the most important file for robot simulation in Gazebo. It can be used to describe the physical and visual geometry of the robot and its electronic components such as motors and sensors. This file is used to load the model in the simulation environment. It is derived from the XML format, with the possibility of having elements hierarchically and logically arranged.

3.8 .3 Building ROS 2 Packages Using C++

This process is done in the same way as was done previously in creating C++ packages, but with modifications to the name and the additions that SDF needs in order to work.

```
cd ~/ turtlebot3_ws_bt /src/
ros2 pkg create --build-type ament_cmake turtlebot3_gazebo
 --dependencies rclcpp std_msgs geometry_msgs sensor_msgs tf2 gazebo_ros
mkdir -p ~/ turtlebot3_ws_bt /src/ turtlebot3_gazebo / models
```

The above instructions create a ROS 2 C bundle named turtlebot3_gazebo using the ament_cmake construct machine, specifying the basic dependencies for coping with messages, sensors, and the Gazebo simulator. A models folder is then created within the package deal. This folder houses the robotic's SDF (Simulation Description Format) description files, which can be created the usage of 3-D layout gear Blender. These files describe the robot's bodily and visible shape and are loaded through the Gazebo simulator to realistically simulate the robot's motion and interplay with the surroundings in the ROS 2 surroundings.

3.8.4 Basic file structure

File in the SDF format starts with the **<sdf version=x.x>** root element, which defines the implementation version of the SDF format. Under this tag, the robot explanation is given through the **<model name=...>**; tag that enlists all the mechanical parts. You can clarify where the robot is by the '<pose>' tag, which brings up its location (x, y, z) and the orientation (roll, pitch, yaw) in 3D space

The <static> tag is used to specify whether or not a model is static or dynamic. If it's miles authentic, the version is treated as an item that isn't always difficulty to the laws of physics (which include partitions or floors), that means it does no longer pass and isn't always tormented by forces. If it's miles false, the model is taken into consideration a dynamic entity which can move and engage with its environment, as in mobile robots.

```
<sdf version="1.6">
  <model name="turtlebot3_waffle">
 <pose>0.0 0.0 0.0 0.0 0.0 0.0</pose>
   ...
  </model>
</sdf>
```

In an SDF (Simulation Description Format) record, the <sdf version="1.6"> tag marks the start of the document and indicates that its format follows model 1.6 of the SDF general, making sure compatibility with simulators like Gazebo. Next, the robot or object is defined interior a <model> tag, in which the name="turtlebot3_waffle" characteristic specifies the call of the version being used, in this situation "turtlebot3_waffle," applications. The <pose> tag is used to specify the preliminary role and orientation of the version within the simulated environment, where the collection x y z represents roll pitch yaw

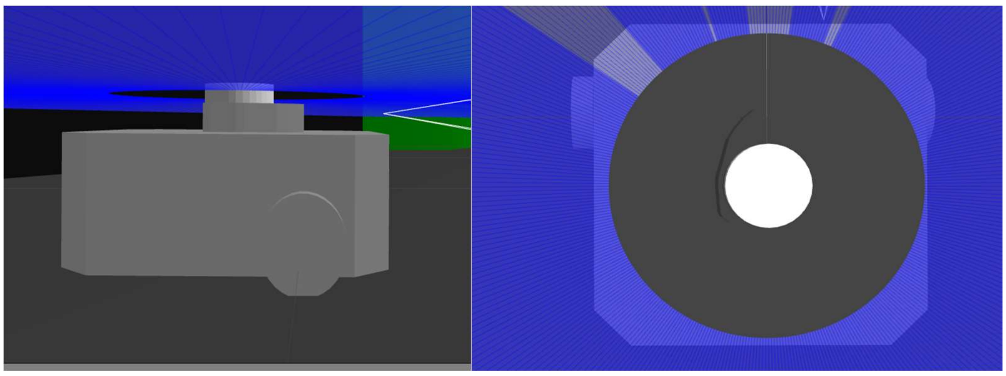


Figure 3.8.3 turtlebot3_waffle

3.8.4 .1 Parent and Child Links in SDF Files

In SDF (Simulation Description Format) files, joints are used to define the kinematic relationship between two robot parts, called:
Parent link: The fixed or reference part to which the other link is attached.
Child link: The moving part affected by the joint.

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <model name="turtlebot3_waffle">
  <pose>0.0 0.0 0.0 0.0 0.0 0.0</pose>
    <link name="base_footprint"/>
    <link name="base_link">
```

The robotic version is described as turtlebot3_waffle and starts through defining its initial role within the simulation area. Next, we be aware two hyperlinks: base_footprint and base_link. In the context of robot layout, base_footprint is usually connected to base_link thru a link that defines the connection among these components. The base_footprint is often the reference factor used to music the robot's function at the ground, while the base_link is the basic structure upon which other components together with sensors and wheels are installed. To entire the kinematic relationship, a <joint> detail is usually delivered that connects these two links and specifies the sort of motion allowed among them (which includes fixed, revolute, or prismatic). This builds the robotic's shape hierarchically and dynamically, allowing for accurate simulation of motion and interactions inside a Gazebo environment or any simulator that helps SDF.

3.8.4 .1 Basic properties of the <joint> link:

When defining joints in an SDF file, a set of factors is used to exactly specify the kinematic houses of the joint. The name element is used to assign a completely unique name to the joint within the model, allowing it to be without problems diagnosed and manipulated. The kind element specifies the kind of joint—the shape of motion allowed between the 2 links connected to it. Common types encompass revolute, which permits constrained rotational motion round a specific axis; continuous, which permits limitless rotation, together with inside the case of wheels; and prismatic, which permits linear movement, along with piston movement. Fixed, but, permits no motion and is used to attach hyperlinks completely. The <parent> and <infant> factors are used to specify the two links attached to a joint, where discern represents the fixed or reference link, and infant is the moving link in an effort to perform the movement relative to parent. To specify the path of movement or rotation, the <axis> tag is used, containing a three-dimensional vector representing the axis of motion the use of the coordinates (x, y, z). Finally, the <limit> element is used to specify the movement limits allowed for a joint, wherein decrease specifies the minimal motion (angular or linear), and higher specifies the maximum movement, which enables nice-music the joint's behavior within the simulation surroundings with realistic precision. In this an SDF file, we see a definition for a pair of joints connecting different parts of a robot, and each joint has specific properties:

```
 <joint name="wheel_right_joint" type="revolute">
     <parent>base_link</parent>
     <child>wheel_right_link</child>
     <pose>0.0 -0.144 0.023 -1.57 0 0</pose>
     <axis>
       <xyz>0 0 1</xyz>
     </axis>
   </joint>
```

```
  <joint name='caster_back_right_joint' type='ball'>
    <parent>base_link</parent>
    <child>caster_back_right_link</child>
  </joint>
```

In this situation SDF report, two joints are defined to connect exclusive parts of a cellular robot. The first joint, named wheel_right_joint, is a revolute joint that connects the robot's major body (base_link) to the right wheel (wheel_right_link). The joint's function and orientation are targeted using the <pose> tag, wherein the position indicates the joint's role relative to the robot's frame, at the same time as the <axis> detail specifies that the joint rotates across the z-axis, that's suitable for wheel rotation. The 2nd joint, named caster_back_right_joint, is a ball joint that connects the base_link to the unfastened proper rear wheel (caster_back_right_link). This type of joint permits free motion in all directions, making it appropriate for small, freely rotating wheels to support the robot without affecting its guidance. This joint does not have a defined axis of movement or position because it naturally lets in unrestricted rotation round all axes.

3.8.4 .5 Sensor Systems<sensor>

When designing robots in SDF files, sensor systems are vital for allowing the robot to intelligently interact with its surroundings. Tags related to sensors are used to exactly discover their properties, determine their region, and the sort of statistics they accumulate.
Type – Sensor Type
Determines the sort of sensor installed on the robot, with types varying depending on the nature of the information to be accrued. Common kinds consist of:

- **Camera:** Used to seize photographs or video.
- **Lidar:** Uses lasers to measure distances and construct 3-d maps.
- **IMU:** Measures acceleration and rotation (inertial size unit).
- **Contact:** Detects collisions among robot elements or with the surroundings.

3.8.4 .6 Main sensor settings

sensor: The main element used to identify any sensor in the SDF. It contains two basic properties:
name: The sensor name (must be unique within the model).
type: The sensor type (such as imu, ray, or camera).

```
<sensor name="sensor_name" type="sensor_type">
  ...
</sensor>
```

pose:Specifies the sensor's position and orientation relative to the issue it's miles set up on (usually a link). It includes 6 values:
Coordinates: x, y, zAngles (radians): roll, pitch, yaw

```
<pose>0 0 0.1 0 0 0</pose>
```

NoiseAll sensors can include a Gaussian <noise> element to simulate fact.
It consists of mean: the imply.

Stddev: the usual deviation.

```
<noise type="gaussian">
  <mean>0.0</mean>
  <stddev>0.01</stddev>
</noise>
```

pluginAll sensors combine with ROS 2 through a plugin loaded from a .So report (along with libgazebo_ros_imu_sensor.So).
This aspect is used to map sensor outputs to ROS subjects.
It regularly contains a remapping of the ROS topic.

```
<plugin name="plugin_name" filename="plugin_library_name.so">
  <ros>
    <remapping>~/out:=ros_topic_name</remapping>
  </ros>
</plugin>
```

3.8.4 .7 Sensors explained

- **Inertia measuring sensor (IMU)** The Inertial Measurement Unit (IMU) is used to measure the robot's acceleration and angular velocity around the 3 axes (X, Y, Z). In the SDF report, this sensor is described the use of the <sensor kind="imu"> tag.

This unit functions Gaussian noise to simulate real-global records and is related to the ROS 2 environment using the libgazebo_ros_imu_sensor.So plugin.
This plugin mechanically publishes facts to a ROS topic called /imu, allowing it for use in navigation and steering systems.

- **LiDAR sensor** LiDAR is a laser-based distance sensor, defined within the SDF using the ray kind.It is established at the robot through a connector called base_scan and scans the encircling surroundings at a 360-diploma horizontal angle.Its properties, consisting of range and size accuracy, may be managed the use of parameters inclusive of variety and samples.

This sensor is connected to ROS 2 through the libgazebo_ros_ray_sensor.So plugin and publishes facts to the /test topic.

- **Front camera** The digicam is used to simulate color computer vision and is diagnosed in SDF by using the sort digital camera.

It is typically set up on the camera_rgb_frame link and simulates a real digital camera such as the Intel RealSense R200 at high decision (e.G., 1080p).
Key settings consist of:
horizontal_fov: The horizontal angle of view.
Clip: The viewing variety (near and a ways).
Noise: Adds noise to simulate a practical photo

of a LiDAR (Ray Sensor)

```
<sensor name="lidar_sensor" type="ray">
  <pose>0 0 0.1 0 0 0</pose>
  <ray>
    <scan>
      <horizontal>
        <samples>360</samples>
        <resolution>1</resolution>
        <min_angle>-3.14</min_angle>
        <max_angle>3.14</max_angle>
      </horizontal>
    </scan>
    <range>
      <min>0.2</min>
      <max>10.0</max>
      <resolution>0.01</resolution>
    </range>
  </ray>
  <plugin name="lidar_plugin" filename="libgazebo_ros_ray_sensor.so">
    <ros>
      <remapping>~/out:=scan</remapping>
    </ros>
  </plugin>
</sensor>
```

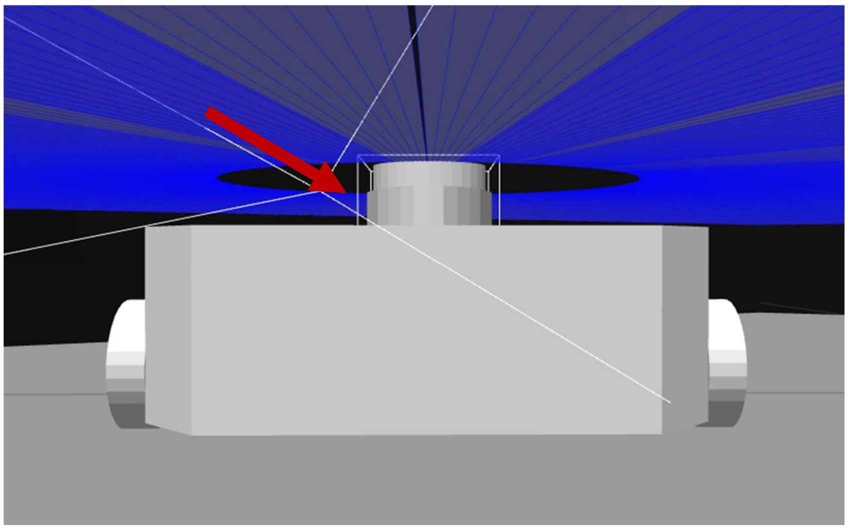


Figure 3.8.4 LiDAR (Ray Sensor)in waffal model

In this case, the <sensor> element is used to define a "ray" LiDAR sensor, which simulates a rotating laser sensor. It is named lidar_sensor and is established at a selected role within the robot, as distinctive by way of the <pose> tag, at a top of zero.1 meters without rotation. The <ray> sub-element includes ray scanning residences, such as a <test> detail that specifies a 360-diploma horizontal experiment range (from -π to π) with a pattern rely of 360 and a scanning resolution aspect of one. Additionally, a <variety> detail specifies a measurable range from zero.2 meters to 10 meters, with a measurement resolution of zero.01 meters. At the cease of the sensor definition, a <plugin> tag is used to attach the libgazebo_ros_ray_sensor.So plugin, which connects this sensor to the ROS 2 system, redirecting the output to a ROS topic referred to as experiment. This lets in LiDAR facts for use in applications which includes impediment avoidance or mapping the robot's surrounding surroundings.

3.8.5 Edit package.xml and CMakeLists.txt

- **CMakeLists.txt**

This document, CMakeLists.Txt, is an important factor of the ROS 2 package deal shape based on ament_cmake in C++ for robotic simulation inside the Gazebo surroundings. The additions included on this record first specify stipulations such as the CMake model and the supported C++ standards (in this situation, C++ 17). Necessary dependencies inclusive of rclcpp, geometry_msgs, sensor_msgs, and tf2, which might be required for interacting with ROS 2, are then covered, at the side of gazebo and gazebo_ros_pkgs, libraries precise to the Gazebo simulator that permit integration with ROS 2. Directories for the Gazebo libraries are also related and delivered to the include paths. In the installation segment, an critical addition is copying the launch, models, urdf, rviz, and worlds directories to the share/$PROJECT_NAME/ direction, allowing simulation files (which include SDFs, URDFs, and scenes) to be deployed in the package. These additions permit the package deal to perform a full simulation of a TurtleBot3 robot in the Gazebo surroundings successfully and combine with the ROS 2 architecture.

```
# Set minimum required version of cmake, project name and compile options
cmake_minimum_required(VERSION 3.5)
project(turtlebot3_gazebo)
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 17)
endif()
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES
"Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()
if(MSVC)
```

```
  add_compile_definitions(_USE_MATH_DEFINES)
endif()
# Find ament packages and libraries for ament and system dependencies
find_package(ament_cmake REQUIRED)
find_package(gazebo REQUIRED)
find_package(gazebo_ros_pkgs REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(nav_msgs REQUIRED)
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(tf2 REQUIRED)
# uild
link_directories(  ${GAZEBO_LIBRARY_DIRS})
include_directories(  include  ${GAZEBO_INCLUDE_DIRS})
set(dependencies  "geometry_msgs"  "nav_msgs"  "rclcpp"  "sensor_msgs"  "tf2")
# Install
install(DIRECTORY launch models rviz urdf worlds DESTINATION
share/${PROJECT_NAME}/)
install(DIRECTORY include/ DESTINATION include/)
# Macro for ament package
ament_export_include_directories(include)
ament_export_dependencies(gazebo_ros_pkgs)
ament_export_dependencies(geometry_msgs)
ament_export_dependencies(nav_msgs)
ament_export_dependencies(rclcpp)
ament_export_dependencies(sensor_msgs)
ament_export_dependencies(tf2)
ament_package()
```

- **package.xml**

The package.xml file defines the core dependencies the package needs to run in the ROS 2 environment, such as rclcpp, geometry_msgs, sensor_msgs, and tf2, as well as gazebo_ros_pkgs, which enables integration with the Gazebo simulator. ament_cmake is used as the main build tool. In the export module, the path to the 3D models is specified via the line <gazebo_ros gazebo_model_path="${prefix}/models"/>, which enables Gazebo to automatically load model files (such as SDFs) from the models folder within the package. This integration is essential for the simulation to run smoothly.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>turtlebot3_gazebo</name>
  <version>2.2.6</version>
  <description>
```

```
    Gazebo simulation package for the TurtleBot3
  </description>
  <maintainer email="willson@robotis.com">Will Son</maintainer>
  <license>Apache 2.0</license>
  <url type="website">http://turtlebot3.robotis.com</url>
  <url type="repository">https://github.com/ROBOTIS-GIT/turtlebot3_simulations</url>
  <url type="bugtracker">https://github.com/ROBOTIS-GIT/turtlebot3_simulations/issues</url>
  <author email="thlim@robotis.com">Darby Lim</author>
  <author email="pyo@robotis.com">Pyo</author>
  <author>Ryan Shim</author>
  <buildtool_depend>ament_cmake</buildtool_depend>
  <depend>gazebo_ros_pkgs</depend>
  <depend>geometry_msgs</depend>
  <depend>nav_msgs</depend>
  <depend>rclcpp</depend>
  <depend>sensor_msgs</depend>
  <depend>tf2</depend>
  <export>
    <build_type>ament_cmake</build_type>
    <gazebo_ros gazebo_model_path="${prefix}/models"/>
  </export>
</package>
```

3.9 Using Behavior Trees in Robot Programming with ROS 2

3.9.1 Introduction

Behavior Trees (BTs) are a flexible structure used to organize decision-making in intelligent systems, especially autonomous robots. This model breaks down complex tasks into connected nodes that follow a specific execution

3.9.2 Setting up a working environment for a behavior tree node in ROS 2

Create a ROS 2 package to support Behavior Trees

A scalable, robust framework is created for a project that uses a behavior tree to control a robot, ensuring seamless integration between the behavior tree and ROS 2 nodes.

```
ros2 pkg create --build-type ament_cmake tb3_autonomy --dependencies rclcpp
behavior_tree_cpp_v3 pluginlib
```

The ros2 pkg create command is used to create a new ROS 2 package named tb3_autonomy with the build type ament_cmake specified to support C++. We add the dependencies rclcpp, behavior_tree_cpp_v3, and pluginlib to enable programming robot behaviors using a behavior tree and loading nodes as dynamic components. The behavior_tree_cpp_v3 library is essential for writing and executing behavior trees, while pluginlib allows manually written nodes to be linked to the tree's XML file.

Go to the package folder and create the folders.

```
cd ~ / turtlebot3_ws_bt /src/ tb3_autonomy
mkdir   bt_xml
mkdir launch
```

Explanation of added folders folder

bt_xml It shops XML documents that describe a conduct tree that uses XML to configure robot behavior in a flexible and easy-to-edit way while not having to recompile the code.

Launch Includes ROS 2 launch files (.launch.py) to start nodes and run the behavior tree within the full system.

3.9.3 Basic types of behavior tree nodes

In the BehaviorTree.CPP library, nodes are classified into four main types, each of which serves a specific role in decision-making logic.

- **Sequence Node** A control node executes its children one by one from left to right. If any child node returns FAILURE or RUNNING, it stops immediately and returns the same result. SUCCESS is returned only if all children succeed.

- **Fallback Node** A control node executes its children from left to right until one succeeds or returns RUNNING. Used to test multiple solutions to the same problem. Returns FAILURE only if all nodes fail.

- **Parallel Node** All subnodes execute simultaneously. They are used when multiple tasks need to be executed simultaneously (such as walking and monitoring). A success condition can be defined as a certain number of successes or failures.
- **Decorator Node** Wraps a single child node and changes its behavior. A popular example is RetryUntilSuccessful, which repeats the execution of the node until it succeeds, or Inverter, which reverses the result.

- **Action Node** Performs a specific task such as moving or grabbing an object. Returns one of three states: SUCCESS, FAILURE, or RUNNING. Typically written in C++ and inherits from BT::SyncActionNode or BT::StatefulActionNode.

- **Condition Node** Checks a specific condition and immediately returns either SUCCESS or FAILURE. It does not perform any execution. Ideal for testing conditions (such as whether the charge is sufficient).

3.9.4 Behavior Tree package explanation

3.9.4.1. BT file

First, we navigate to the folder containing the behavior tree files in the ROS 2 project. Then, using nano tree.xml, we open a text editor to create or edit an XML file containing the robot's behavior tree definition.

```
cd ~ / turtlebot3_ws_bt /src/ tb3_autonomy/ bt_xml
nano tree.xml
```

3.9.4.2 Then we write the behavior tree code

The file starts offevolved with a <root> detail that specifies the name of the principle tree for execution (MainTree). Within it, we define a behavior tree using <BehaviorTree ID="MainTree">, where the tree carries a Sequence node referred to as sequence. A Sequence node executes its children sequentially, shifting from one to the subsequent handiest if the preceding one become a success. In this case, we have 4 GoToPose nodes, every representing a command for the robot to visit a particular region (location1 to location4) in series. Each node is known as by means of a call and a goal vicinity (loc).

```
<root main_tree_to_execute = "MainTree" >
  <BehaviorTree ID="MainTree">
    <Sequence name="sequence">
      <GoToPose name="go_to_location1" loc="location1" />
      <GoToPose name="go_to_location2" loc="location2" />
      <GoToPose name="go_to_location3" loc="location3" />
      <GoToPose name="go_to_location4" loc="location4" />
    </Sequence>
  </BehaviorTree>
</root>
```

3.9.5 Create a Behavior Tree

- **Step 1 Inclusion and nominal spaces**

The autonomy_node.H header file containing the AutonomyNode elegance definitions is blanketed. The using namespace std::chrono_literals; statement additionally lets in you to apply time constants like 500ms directly, making it easier to paintings with timers and time functions.

```
#include "autonomy_node.h"
using namespace std::chrono_literals;
```

- **Step 2 Behavior tree file path (XML)**

The path to the Behavior Tree XML file is determined by the ROS 2 get_package_share_directory function and the bt_xml folder is searched inside the ROS 2 package named tb3_autonomy.

```
const std::string bt_xml_dir =  ament_index_cpp::get_package_share_directory("tb3_autonomy") +
"/bt_xml";
```

- **Step 3 Creator Constructor**

This element represents the constructor of the AutonomyNode magnificence, which is used to initialize a new object of this magnificence upon creation. The constructor gets the node name as a controversy (nodeName) and passes it to the parent Node constructor to initialize the ROS 2 node with the correct name. Next, a parameter named "location_file" is asserted with a default cost of "none," allowing it to be changed later via configuration documents or the command line. Finally, an "Init performed" message is outlined to the device log to confirm that the initialization method changed into successful.

```
AutonomyNode::AutonomyNode(const std::string &nodeName) : Node(nodeName)
{this->declare_parameter("location_file","none");
 RCLCPP_INFO(get_logger(), "Init done");}
```

- **Step 4 Setting function**

This function is used to prepare the behavior tree and create a periodic timer (every 500 milliseconds). Create_behavior_tree() creates the tree from an XML file. Create_wall_timer() creates a timer that calls the update_behavior_tree() function periodically. rclcpp::spin() runs a ROS loop to activate timers and subscriptions. After execution, the node is terminated using rclcpp::shutdown().

```
void AutonomyNode::setup()
{RCLCPP_INFO(get_logger(), "Setting up");
 create_behavior_tree();
 RCLCPP_INFO(get_logger(), "BT created");
 const auto timer_period = 500ms;
 timer_                                   =                                   this-
>create_wall_timer(timer_period,std::bind(&AutonomyNode::update_behavior_tree, this));
 rclcpp::spin(shared_from_this());
 rclcpp::shutdown();}
```

- **Step 5 Create a behavior tree**

The create_behavior_tree() function creates the robot's behavior tree using the BehaviorTree.CPP library. It starts offevolved by creating a BehaviorTreeFactory item, that is used to institution and be a part of nodes inside the tree. A Builder function is then described for a custom node called GoToPose, using Lambda to generate a new object from this node with the important parameters. This node is then registered inside the manufacturing unit under the call "GoToPose" in order that it is able to be identified within the behavior tree record. Information is then published to assist music execution, and eventually, the tree is loaded from an XML document saved at bt_xml_dir "/tree.Xml" to create the tree structure with the intention to be used for periodic updates.

```
void AutonomyNode::create_behavior_tree()
{
  BT::BehaviorTreeFactory factory;
   BT::NodeBuilder builder =[=](const std::string &name, const BT::NodeConfiguration &config)
  {
    return std::make_unique<GoToPose>(name, config, shared_from_this());};
    factory.registerBuilder<GoToPose>("GoToPose", builder);
  RCLCPP_INFO(get_logger(), bt_xml_dir.c_str());
  tree_ = factory.createTreeFromFile(bt_xml_dir + "/tree.xml");
  RCLCPP_INFO(get_logger(), "3");}
```

- **Step 8 Update behavior**

treeThe update_behavior_tree() characteristic is used to periodically execute and update the behavior tree. Each time it is called, tickRoot() is called on the tree, which activates the root node and all its kids and returns the execution repute (RUNNING, SUCCESS, or FAILURE). If the popularity is RUNNING, execution is still in development, so no movement is taken. If the tree leads to SUCCESS, a message is printed indicating that the traversal became a hit. If it fails, a failure message is outlined and the timers are canceled using timer_->cancel(), which stops the periodic tree update.

```
void AutonomyNode::update_behavior_tree(){
  BT::NodeStatus tree_status = tree_.tickRoot();
  if (tree_status == BT::NodeStatus::RUNNING){
    return;}
  else if (tree_status == BT::NodeStatus::SUCCESS) {
    RCLCPP_INFO(this->get_logger(), "Finished Navigation");}
  else if (tree_status == BT::NodeStatus::FAILURE) {
    RCLCPP_INFO(this->get_logger(), "Navigation Failed");
    timer_->cancel(); }}
```

- **Step 9 Main function main**

The primary characteristic represents the place to begin of the program. It initializes the ROS 2 device with the rclcpp::init(argc, argv) function to enable node and message managing. Next, a shared AutonomyNode object named "autonomy_node" is created, representing the ROS node containing the behavioral tree manage good judgment. The setup() function is then known as to initialize the conduct tree and start the replace timer. Finally, zero is back to signify that this system has efficaciously completed.

```
int main(int argc, char **argv){
  rclcpp::init(argc, argv);
  auto node = std::make_shared<AutonomyNode>("autonomy_node");
  node->setup();
```

```
   return 0;}
```

3.10 Create a Launch File in ROS 2

When organizing a cellular robotic simulation undertaking using ROS 2, it's far essential to systematically distribute launch files inside the various applications that make up the device. Each package deal has a selected position within the gadget, so a release folder is allotted within each bundle to comprise its launch documents, ensuring bundle independence and ease of reuse and improvement.In this mission, we created three major applications:

- **tb3_sim/:**

The bundle liable for putting in place the overall simulation environment and running the robotic components.

- **Turtlebot3_gazebo/:**

The package deal that integrates with the Gazebo simulator and presents the virtual world and robot fashions.

- **Tb3_autonomy/:**

The package committed to self sustaining robotic behaviors, consisting of navigation and obstacle avoidance.

3.10.1 Create launch folder:

First, we go into the package we are working with that we created before, tb3_sim, and create a new folder dedicated to the launch files:

```
cd ~ turtlebot3_ws_bt / /src/ tb3_sim/src
mkdir launch
```

- **Create a launch file:**

We create a new file in the format .launch.py, in which the code will write the launch files.

```
cd launch
nano. turtlebot3_world.launch
```

In the same way, launch files are created.

3.10.2 file structure explanation Turtlebot3_world.launch

The turtlebot3_world.Launch.Py release document is used to initialize the simulation environment in Gazebo and launch the TurtleBot3 robot inside it. This includes deploying the robotic's bodily version and including it to the digital global with timing and visible simulation enabled.

- **Step 1 Import libraries**

This code is utilized in ROS 2 to create a launch report to launch a couple of additives inside a simulation or robot environment. It uses features from the release library to dynamically include other release documents using IncludeLaunchDescription and specifies paths through get_package_share_directory. LaunchConfiguration is also used to read values from configuration files or consumer input at launch

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
```

- **Step 2 The basic function**

for generating the launch fileThis is the function that must be defined in any Python launch file in ROS 2. It returns a LaunchDescription object.

```
def generate_launch_description():
```

- **Step 3 Determine package paths turtlebot3_world.launch**

Paths in ROS 2 are used to locate necessary files such as launch files, world files, and models. Packages within the turtlebot3_gazebo package contain TurtleBot3 simulation files such as worlds and launch files. gazebo_ros integrates the Gazebo simulator with ROS 2 to provide interaction between the environment and the simulator. The tb3_sim package often contains files specific to the robot simulator such as user-specific configurations or modifications.

```
 launch_file_dir = os.path.join(
     get_package_share_directory('turtlebot3_gazebo'), 'launch')
  pkg_gazebo_ros = get_package_share_directory('gazebo_ros')
  pkg_tb3_sim = get_package_share_directory('tb3_sim')
 world = os.path.join( get_package_share_directory('turtlebot3_gazebo'),
     'worlds', 'turtlebot3_world.world')
```

- **Step 4 Explanation of the launch description: Turtlebot3_world.launch**

This piece of code is used to insert and run sub-release files at the same time as jogging a simulation in ROS 2. The robot_state_publisher_cmd command runs the robot_state_publisher.Launch.Py record to submit the robotic's nation (along with joint positions) with use_sim_time enabled. The spawn_turtlebot_cmd command runs the spawn_turtlebot3.Launch.Py file to set up the robot into the simulation surroundings at specific coordinates (x_pose and y_pose). These values are handed as launch arguments the usage of launch_arguments.

```
robot_state_publisher_cmd = IncludeLaunchDescription( PythonLaunchDescriptionSource(
os.path.join(launch_file_dir,'robot_state_publisher.launch.py') ),
     launch_arguments={'use_sim_time': use_sim_time}.items() )
  spawn_turtlebot_cmd = IncludeLaunchDescription(
     PythonLaunchDescriptionSource(os.path.join(launch_file_dir, 'spawn_turtlebot3.launch.py')),
     launch_arguments={
         'x_pose': x_pose,
         'y_pose': y_pose
     }.items()
```

- **File 1:**

This code is used to launch the spawn_entity.Py node from the gazebo_ros package, that is responsible for placing the robotic into the Gazebo simulation the usage of an .Sdf document. The robot version is designated thru the TURTLEBOT3_MODEL surroundings variable, and the document is loaded from the fashions folder. The starting coordinates (x .y) are also defined as parameters that may be custom designed upon launch.

```
 TURTLEBOT3_MODEL = os.environ['TURTLEBOT3_MODEL']
    model_folder = 'turtlebot3_' + TURTLEBOT3_MODEL
    urdf_path = os.path.join(get_package_share_directory('turtlebot3_gazebo'),
        'models', model_folder, 'model.sdf' )
```

- **File 2:**

This code is used to run the robot_state_publisher node, which publishes the robotic's nation (including joint positions) based totally on a URDF document. The suitable URDF file is loaded in step with the robotic kind specific inside the TURTLEBOT3_MODEL variable. Simulation time (use_sim_time) is likewise allowed if requested, retaining the node synchronized with the Gazebo simulator's time.

```
TURTLEBOT3_MODEL = os.environ['TURTLEBOT3_MODEL']
    use_sim_time = LaunchConfiguration('use_sim_time', default='false')
    urdf_file_name = 'turtlebot3_' + TURTLEBOT3_MODEL + '.urdf'
    print('urdf_file_name : {}'.format(urdf_file_name))
    urdf_path = os.path.join(
        get_package_share_directory('turtlebot3_gazebo'), 'urdf',
         urdf_file_name)
    with open(urdf_path, 'r') as infp:
        robot_desc = infp.read()
    return LaunchDescription([ DeclareLaunchArgument( 'use_sim_time',
   default_value='false',description='Use simulation (Gazebo) clock if true'),
```

- **Step 5 Create and return the final launch file**

In this phase, a Launch Description object is created, representing the whole release file. Four commands are then delivered to it: launch the Gazebo server, release its graphical interface, launch the robotic kingdom publisher, and sooner or later set up the robot within the simulator. Finally, those configurations are again to be completed robotically while the file is launched.

```
ld = LaunchDescription()
ld.add_action(gzserver_cmd)
ld.add_action(gzclient_cmd)
ld.add_action(robot_state_publisher_cmd)
ld.add_action(spawn_turtlebot_cmd)
return ld
```

3.10.3 file structure explanation tb3_sim nav2.launch

In this phase, we will display the way to implement the TurtleBot3 localization and navigation machine inside a simulation surroundings the use of ROS 2 and the Nav2 system. The nav2.Release.Py launcher document within the tb3_sim package deal calls the nav2_bringup bundle to release all of the important navigation additives, together with positioning the use of the AMCL algorithm and path making plans, as well as showing the visual interplay surroundings the usage of RViz. This record also demonstrates how to organize the internal shape of the folders and files necessary to support the localization and navigation process effectively and smoothly in the simulation.

- **Step 1 Determine package paths: tb3_sim nav2.launch**

In this segment, the trails to the ROS 2 core applications nav2_bringup for the navigation gadget and tb3_sim for the TurtleBot3 simulator are described using get_package_share_directory. The launch variables use_sim_time are then described to enable simulation time and autostart to mechanically begin the system. These settings help manage the conduct of the navigation system all through startup.

```
pkg_nav2_dir = get_package_share_directory('nav2_bringup')
pkg_tb3_sim = get_package_share_directory('tb3_sim')
use_sim_time = LaunchConfiguration('use_sim_time', default='True')
autostart = LaunchConfiguration('autostart', default='True')
```

- **Step 2 Run RViz and the primary positioning node**

In this segment of the code, a node is created to run the RViz2 program to visually display the simulation environment, assisting to reveal the robot's motion and interplay with the surroundings. Another node is created to installation the initial role of the AMCL localization machine by way of putting the robot's beginning coordinates (x and y) within the map, ensuring that the localization system starts from a exactly defined area.

```
rviz_launch_cmd = Node(package="rviz2",
executable="rviz2",name="rviz2",arguments=[...])
```

```
set_init_amcl_pose_cmd                    =                    Node(package="tb3_sim",
executable="amcl_init_pose_publisher",  name="amcl_init_pose_publisher",
   parameters=[{ "x": -2.0,"y": -0.5,}])
```

- **Step 3 Create an object LaunchDescription**

In this segment, a LaunchDescription item is created to organization all of the commands and nodes with a view to be run together. Commands for launching the navigation device, deploying the AMCL preliminary mode, and walking RViz are then delivered to this description. Finally, this description is returned to start the included system release technique.

```
ld = LaunchDescription()
ld.add_action(nav2_launch_cmd)
ld.add_action(set_init_amcl_pose_cmd)
ld.add_action(rviz_launch_cmd)
return ld
```

3.10.4 file structure explanation tb3_autonomy autonomy.launch.py

The autonomy.Launch.Py document within the tb3_autonomy bundle is used to release a couple of additives that manipulate the behavior of a TurtleBot3 robotic autonomously inside a ROS 2 environment. This file coordinates the launch of vital nodes inclusive of publishing the robot's nation, loading the version, and launching the decision-making modules. Its inclusion inside the package deal enhances organization and allows the launch of the whole gadget with a unmarried launch command.

- **Step 1 Import libraries:**

This piece of code is used inside the autonomy.Release.Py release document to prepare a runtime environment for ROS 2 nodes. The get_package_share_directory characteristic is imported to specify paths inside packages, and the LaunchDescription characteristic is imported to bring together the nodes and actions to be launched. Node from launch_ros.Actions is also used to specify and launch ROS 2 nodes, inclusive of control or perception algorithms, within the tb3_autonomy package deal.

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

- **Step 2 Determine package paths   : autonomy.launch**

The two lines are used to specify the root paths of two packages, tb3_sim and tb3_autonomy, using the get_package_share_directory function.

```
pkg_tb3_sim = get_package_share_directory('tb3_sim')
pkg_tb3_autonomy = get_package_share_directory('tb3_autonomy')
```

- **Step 3 Definition of the node to be released: autonomy.launch**

The AutonomyNode is the node responsible for running the behavior tree for autonomous robot navigation. This tree uses a custom behavioral node called GoToPose, which is responsible for sending a navigation command to the robot toward a specific location specified by a key name in a YAML file containing the location coordinates.

```
autonomy_node_cmd = Node( package="tb3_autonomy", executable="autonomy_node",
    name="autonomy_node",
    parameters=[{ "location_file": os.path.join(pkg_tb3_sim, "config", "sim_house_locations.yaml")
}])
```

- **Step 4 Create a launch description object and add the node to it.**

In this part, a LaunchDescription item is created, which is used to organization all the nodes and moves to be released. The autonomy_node_cmd node is then introduced to it, and the ld object is again to be accomplished while the report is administered the usage of the ros2 release command.

```
ld = LaunchDescription()
 ld.add_action(autonomy_node_cmd)
return ld
```

3.11Mobile robot simulation:

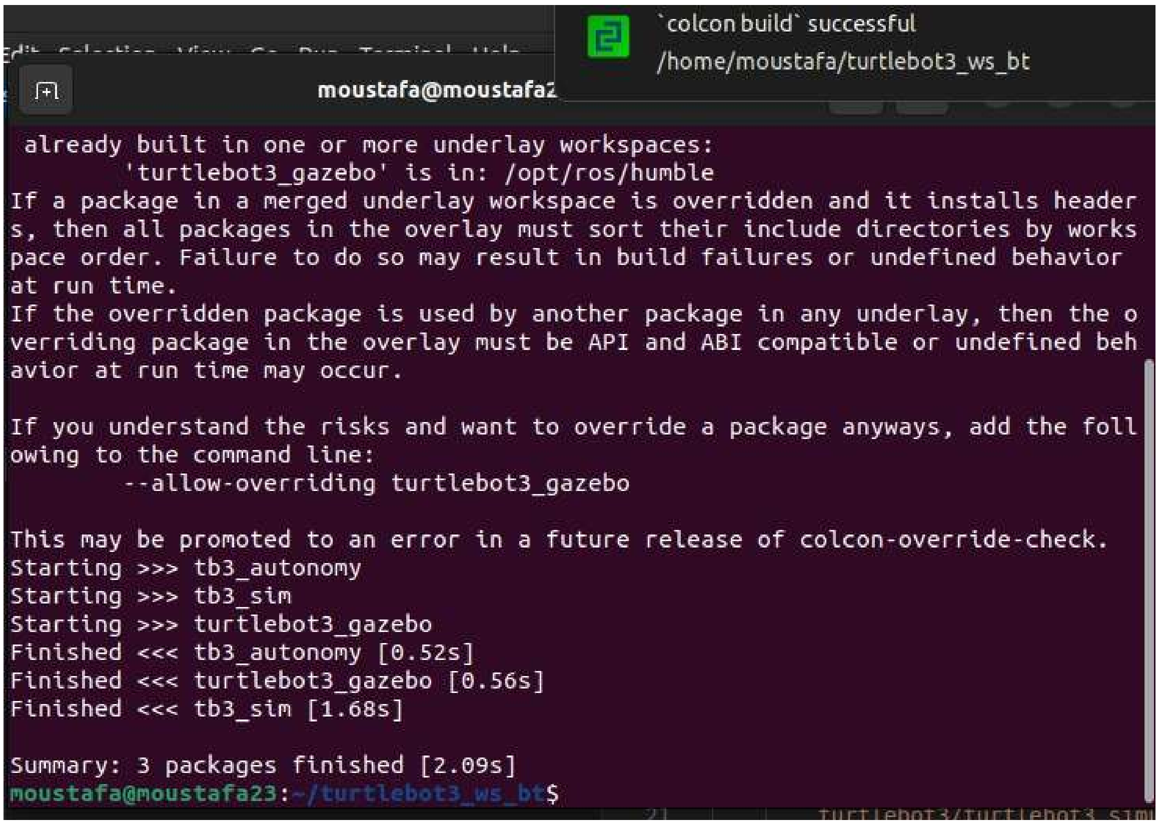
The simulation process is structured into three main stages, each executed in a separate terminal to facilitate modular control of the robot, navigation stack, and autonomous behavior system. These stages are as follows:

3.11.1 Launching the Simulation Environment (Terminal 1):

Command :

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

Output: The figure 3.10.1 following



```
 already built in one or more underlay workspaces:
        'turtlebot3_gazebo' is in: /opt/ros/humble
If a package in a merged underlay workspace is overridden and it installs header
s, then all packages in the overlay must sort their include directories by works
pace order. Failure to do so may result in build failures or undefined behavior
at run time.
If the overridden package is used by another package in any underlay, then the o
verriding package in the overlay must be API and ABI compatible or undefined beh
avior at run time may occur.

If you understand the risks and want to override a package anyways, add the foll
owing to the command line:
        --allow-overriding turtlebot3_gazebo

This may be promoted to an error in a future release of colcon-override-check.
Starting >>> tb3_autonomy
Starting >>> tb3_sim
Starting >>> turtlebot3_gazebo
Finished <<< tb3_autonomy [0.52s]
Finished <<< turtlebot3_gazebo [0.56s]
Finished <<< tb3_sim [1.68s]

Summary: 3 packages finished [2.09s]
moustafa@moustafa23:~/turtlebot3_ws_bt$
```

Figure 3.10.1 colcon build

```
. install/setup.bash

ros2 launch tb3_sim turtlebot3_world.launch.py
```

This launch file is responsible for setting up the Gazebo simulation environment and spawning the TurtleBot3 robot. It performs the following actions:

➢ robot_state_publisher.launch.py

- This launch file initiates the robot_state_publisher node, which publishes the state (transforms) of the robot to the TF tree.

It uses the following parameters:

- use_sim_time: Allows ROS nodes to use the simulation time from Gazebo.

Robot_description: Contains the robot's SDF specifying the robot's physical and kinematic properties.

- spawn_turtlebot3.launch.py
- This component spawns the TurtleBot3 in the Gazebo world using spawn_entity.py from the gazebo_ros package.

It utilizes:

- The .sdf model of the robot.
- Position parameters: x_pose and y_pose to determine the initial location of the robot in the world.
  - Gazebo Server and Client Launch:
- gzserver.launch.py loads the simulation world (e.g., a house environment).
- gzclient.launch.py opens the Gazebo graphical interface for visualization.

The following(figure 3.10.2)shows the Gazebo simulation environment after launching the turtlebot3_world.launch.py file. In this environment, the TurtleBot3 robot is successfully spawned and positioned in the virtual world, ready for navigation and autonomous behavior testing

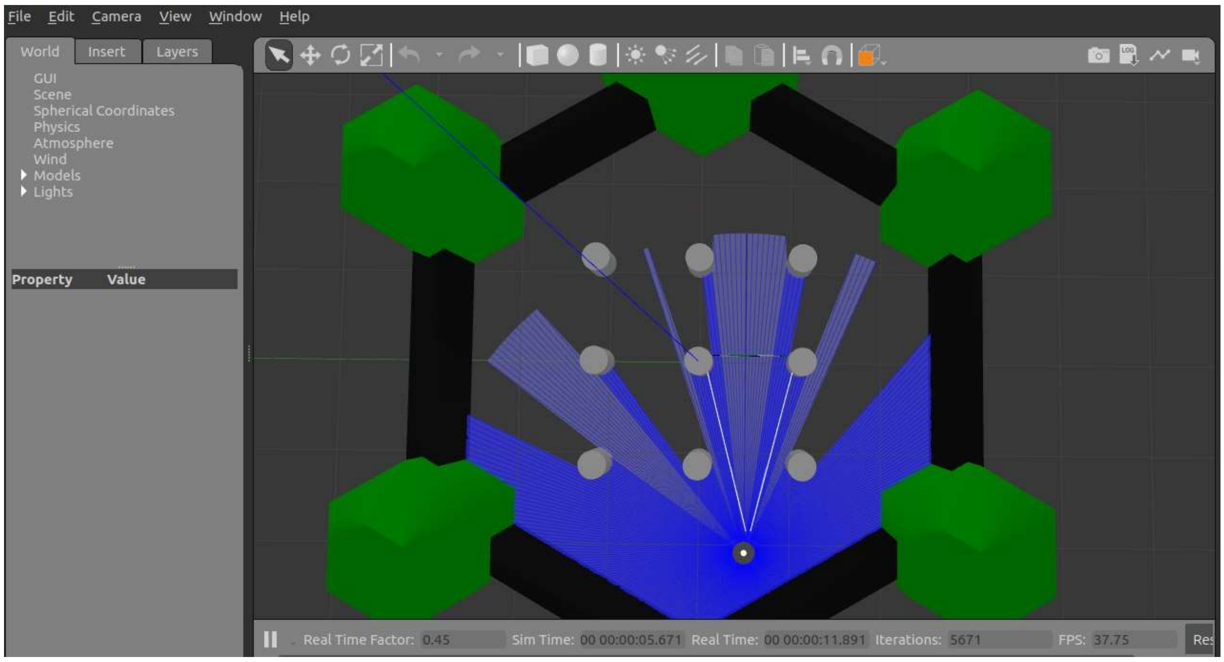


Figure 3.10.2 Gazebo Simulation Environment with Spawned TurtleBot3

3.11.2 Launching the Navigation Stack (Terminal 2):

Command:

```
. install/setup.bash
ros2 launch tb3_sim nav2.launch.py
```

This file initializes the Nav2 stack, enabling autonomous navigation through SLAM, localization, and path planning. It performs the following:

This file initializes the Nav2 stack, enabling autonomous navigation through SLAM, localization, and path planning. It performs the following:

- ➢ bringup_launch.py from nav2_bringup
- Starts all necessary Nav2 nodes (e.g., amcl, planner_server, controller_server, map_server, etc.).

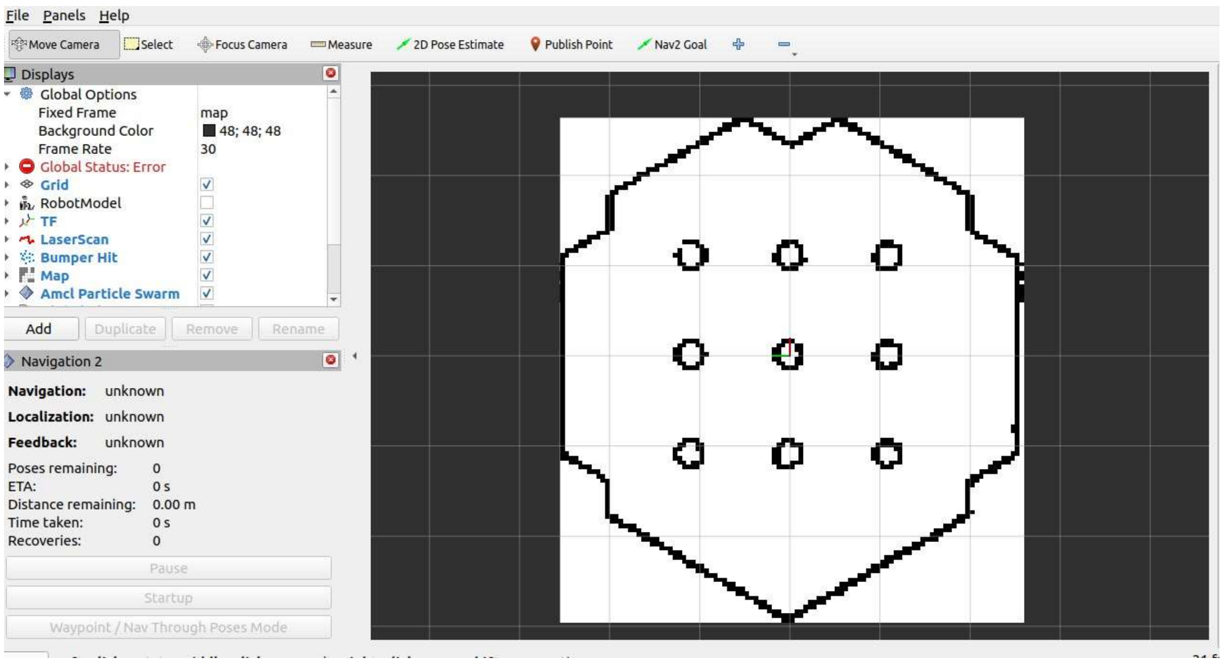
Parameters:

- use_sim_time: Synchronizes the system with Gazebo's simulated clock.
- map: Loads a static map in .yaml format for localization and navigation.

- ➢ Initial Pose Publisher

A custom node (amcl_init_pose_publisher) sends the initial pose of the robot to the amcl localization system using send_init_pose() function.

The figure 3.10.3 below shows the RViz visualization environment after launching the nav2.launch.py file. The loaded map allows the localization and path planning functionalities of the TurtleBot3 robot to operate correctly within the simulation environment



```
Global Options
  Fixed Frame        map
  Background Color   48; 48; 48
  Frame Rate         30
Global Status: Error
Grid                 ✓
RobotModel
TF                   ✓
LaserScan            ✓
Bumper Hit           ✓
Map                  ✓
Amcl Particle Swarm  ✓
```

```
Navigation:    unknown
Localization:  unknown
Feedback:      unknown
Poses remaining:     0
ETA:                 0 s
Distance remaining:  0.00 m
Time taken:          0 s
Recoveries:          0
```

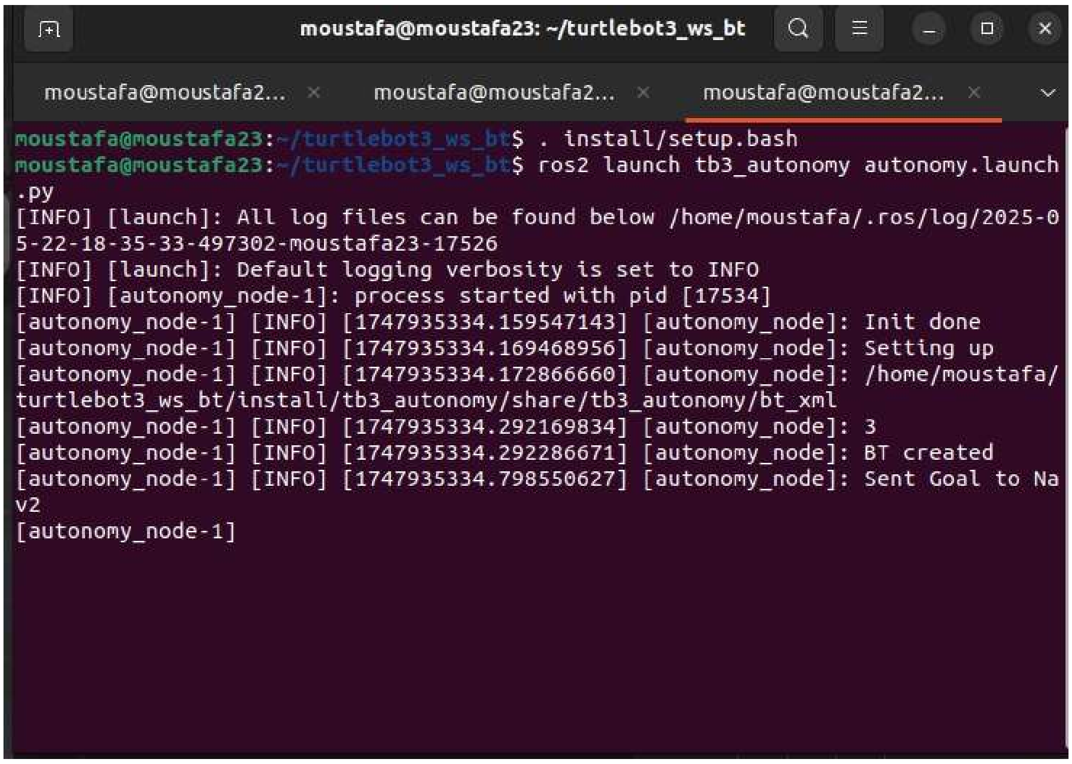
Figure 3.10.3 RViz Interface Displaying the Navigation Map and Robot Position

3.11.3 Launching the Autonomy System (Terminal 3)

Command:

```
 . install/setup.bash
ros2 launch tb3_autonomy autonomy.launch.py
```

Output: The figure 3.10.4 following



```
moustafa@moustafa23:~/turtlebot3_ws_bt$ . install/setup.bash
moustafa@moustafa23:~/turtlebot3_ws_bt$ ros2 launch tb3_autonomy autonomy.launch
.py
[INFO] [launch]: All log files can be found below /home/moustafa/.ros/log/2025-0
5-22-18-35-33-497302-moustafa23-17526
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [autonomy_node-1]: process started with pid [17534]
[autonomy_node-1] [INFO] [1747935334.159547143] [autonomy_node]: Init done
[autonomy_node-1] [INFO] [1747935334.169468956] [autonomy_node]: Setting up
[autonomy_node-1] [INFO] [1747935334.172866660] [autonomy_node]: /home/moustafa/
turtlebot3_ws_bt/install/tb3_autonomy/share/tb3_autonomy/bt_xml
[autonomy_node-1] [INFO] [1747935334.292169834] [autonomy_node]: 3
[autonomy_node-1] [INFO] [1747935334.292286671] [autonomy_node]: BT created
[autonomy_node-1] [INFO] [1747935334.798550627] [autonomy_node]: Sent Goal to Na
v2
[autonomy_node-1]
```

Figure 3.10.4 Launching the Autonomy System in terminal

This stage launches the custom autonomy node, which manages high-level decision-making through Behavior Trees (BT). This node is essential for implementing autonomous tasks such as navigation to predefined points.

➢ Loading Behavior Tree from XML

The system reads a behavior tree structure from an .xml file. This defines the robot's behavior in terms of actions and decisions (e.g., move to point A, wait, return to base, etc.).

➢ Location File

A .yaml file (sim_house_locations.yaml) defines named goal positions (e.g., kitchen, living room, etc.) with their coordinates.

➢ Behavior Tree Execution

The autonomy node:

- Creates the tree using the XML structure.
- Continuously updates the status of the tree.

- Monitors the current state of execution (e.g., RUNNING, SUCCESS, or FAILURE) to make decisions or re-plan accordingly.

The following figure 3.10.5 illustrates the execution of the autonomy node using a behavior tree. This tree defines a sequence of actions and decisions that allow the TurtleBot3 robot to navigate autonomously between predefined locations within the simulated environment

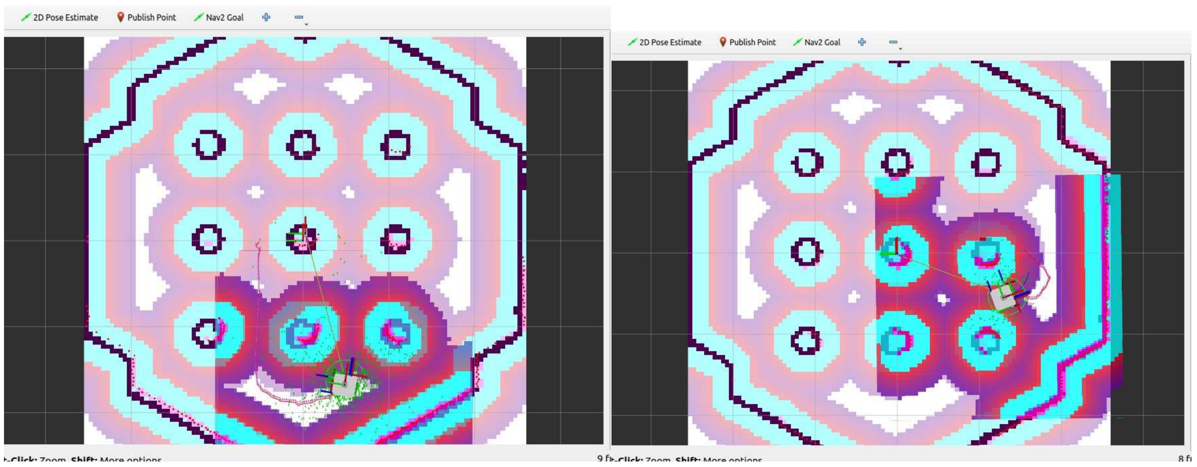


Figure 3.10.5 Autonomy Node Execution with Behavior Tree for TurtleBot3

This three-stage simulation setup creates a complete robotic system within ROS 2 that mimics real-world behavior. The robot is spawned in a simulated environment, navigates using a robust localization and planning stack, and operates autonomously based on high-level behavior trees. This structure allows for scalable testing of various robotic applications such as patrol, delivery, and human-robot interaction in a controlled simulation environment.

3.12 conclusion

his chapter presents a systematic journey toward establishing an advanced robotic simulation environment, spanning from technical foundations to intelligent modeling and control. The initial stages involved installing Ubuntu 22.04 as a stable core for programming and design platforms, followed by configuring the ROS 2 Humble framework—a cornerstone of modern robotics development due to its dynamic architecture and support for distributed computing. The integration of Gazebo as a physics-based simulation environment and Blender as a specialized 3D modeling tool highlighted the synergy between open-source technologies to ensure scalable and precise simulations.

On the software front, the chapter addressed the creation of Python and C++ packages, emphasizing flexibility in algorithm implementation for both rapid prototyping and high-performance applications. Standardization in robot modeling was achieved through URDF/SDF files, ensuring compatibility across simulation platforms and control systems. The exploration of Behavior Trees further underscored the chapter's focus on intelligent autonomy, providing a structured framework for designing complex decision-making processes in robotic systems.

Practically, the chapter bridged theory and application by simulating an integrated mobile robot, from geometric modeling to behavioral testing in virtual scenarios. This holistic approach deepened the understanding of real-world robotics challenges, such as sensor interaction, real-time data processing, and balancing performance with energy efficiency.

Overall, this chapter offers an academic and practical framework for engineering students and robotics researchers, fostering interdisciplinary technical skills through a cohesive sequence of steps—from environmental setup to advanced modeling. It lays a robust foundation for experimentation and innovation, aligning with the demands of the Fourth Industrial Revolution and applied artificial intelligence. By unifying open-source tools, standardized practices, and intelligent control strategies, the chapter equips learners to tackle emerging challenges in autonomous systems and contribute to the evolving landscape of smart robotics.

General Conclusion

The field of robotics has witnessed remarkable advancements in recent decades, emerging as one of the most transformative and interdisciplinary domains of modern technology. Mobile robotics, in particular, represents a pinnacle of innovation, combining mechanical engineering, computer science, and artificial intelligence. However, its complexity lies in the intricate balance required for precise control, navigation, and decision-making in dynamic environments. Designing autonomous systems capable of interpreting sensor data, adapting to uncertainties, and executing tasks efficiently remains a significant challenge, particularly when bridging theoretical models with real-world applications.

To address these challenges, this project leveraged ROS2 Humble, a cutting-edge framework that streamlines robotic development through modular tools such as nodes, topics, and services. ROS2's distributed architecture enabled seamless integration of hardware and software components, while Gazebo provided a powerful simulation environment to model, test, and refine robotic behaviors in risk-free virtual settings. This synergy between ROS2 and Gazebo significantly simplified the prototyping process, allowing for iterative improvements in navigation algorithms, sensor integration, and system coordination.

The practical chapter , while foundational, presented notable challenges. Key difficulties included mastering Gazebo's physics-based simulation parameters, debugging URDF/SDF model discrepancies, and optimizing behavior trees for real-time decision-making. Additionally, aligning simulation results with theoretical expectations required meticulous calibration of sensor models and control loops. These hurdles underscored the steep learning curve associated with advanced simulation tools, emphasizing the need for robust documentation and hands-on experimentation.

To enhance the project's academic and practical impact, future work could prioritize integrating machine learning for adaptive navigation, expanding simulations to multi-robot collaboration scenarios, and validating algorithms on physical hardware. Emphasizing edge computing for resource-constrained platforms and incorporating human-robot interaction models would further bridge the gap between simulation and reality. By fostering open-source contributions and adopting standardized benchmarking practices, this framework can evolve into a versatile platform for next-generation robotics research.

In summary, this thesis underscores the potential of simulation-driven development in overcoming the complexities of mobile robotics. While challenges persist, the combination of ROS2, Gazebo, and systematic experimentation offers a scalable pathway toward intelligent, autonomous systems ready to tackle real-world problems.

References

[1] M. A. D. V. LAXMI, ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING, 2023.

[2] "robotnik," 3 april 2025. [Online]. Available: https://robotnik.eu/what-is-an-industrial-robot-industrial-robot-definition/.

[3] T. Haidegger, Robotics and Autonomous Systems.

[4] "edinformatics," 3 1pril 2025. [Online]. Available: https://www.edinformatics.com/math_science/robotics/military_robots.htm.

[5] J. &. B. J. Enderle, Introduction to Biomedical Engineering* (3rd ed.). Academic Press, 2012.

[6] K. Goris, Autonomous Mobile Robot Mechanical Design, 2005.

[7] S. B. W. &. F. D. Thrun, Probabilistic Robotics, 2005.

[8] R. N. I. R. &. S. Siegwart, Introduction to Autonomous Mobile Robots, 2011.

[9] "Techtarget," [Online]. Available: https://www.techtarget.com/iotagenda/definition/drone.

[10] J. J. Craig, Introduction to Robotics: Mechanics and Control, 2005.

[11] *Robotics: Concepts, Methodologies, Tools, and Applications,* IGI Global,, 2019..

[12] m. Michael, Make an Arduino-controlled robot, 2012.

[13] D. D. S. Y. a. Z. Z. E. H. Liu, Intelligent Robotics and Applications, 2020.

[14] J. L. J. a. A. M. Flynn, Mobile Robots: Inspiration to Implementation, 2nd ed, 1999.

[15] I. R. N. a. D. S. R. Siegwart, Introduction to Autonomous Mobile Robots, 2nd ed., MIT Press, 2011.

[16] W. Y. a. A. Perrusquía, "Robot Kinematics and Dynamics".

[17] "link.springe," [Online]. Available: https://link.springer.com/referenceworkentry/10.1007/978-3-642-41610-1_159-2.

[18] "britannica," [Online]. Available: https://www.britannica.com/science/ultrasonics.

[19] "robu.in," [Online]. Available: https://robu.in/ir-sensor-working/.

[20] T. Developers, ROS Robot Programming, 2017.

[21] "circuitdiges," [Online]. Available: https://circuitdigest.com/article/servo-motor-working-and-basics.

[22] "docs.ros.org," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html.

[23] "docs.ros.org," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Creating-Launch-Files.html.

[24] "docs.ros.org," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Launch-system.html.

[25] "docs.ros.org/," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Launch-Main.html.

[26] :. D. J. P. Tutor, Writer, *Set up and control of a UR3 Robot using ROS2 Humble MASTER'S THESIS.* [Performance].

[27] Matti Kortelainen, A short guide to ROS 2 Humble Hawksbill, Kuopio, Finland, March 22, 2023.

[28] "mathworks," [Online]. Available: https://www.mathworks.com/help/ros/gs/ros2-topics.html.

[29] "docs.ros.org," [Online]. Available: https://docs.ros.org/en/humble/Concepts/Basic/About-Topics.html.

[30] B. G. a. W. D. Quigley, Smart, Programming Robots with ROS: A Practical Introduction to the Robot Operating System., 2015.

[31] H. C. R. J. T. L. YoonSeok Pyo, ROS Robot Programmin, 2017.

[32] F. M. Rico, A Concise Introduction to Robot Programming with ROS2, 2023.

[33] "design.ros2.org," [Online]. Available: https://design.ros2.org/articles/actions.html.

[34] "robotic sunveiled," [Online]. Available: https://www.roboticsunveiled.com/ros2-rviz2/.

[35] "docs.ros.org," [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Intermediate/RViz/RViz-User-Guide/RViz-User-Guide.html#displays.

[36] "oreilly," [Online]. Available: https://www.oreilly.com/library/view/ros-programming-building/9781788627436/192de5c9-e5bd-40b3-a75a-2990bdfa7caf.xhtml.

[38] "openrobotic," [Online]. Available: https://www.openrobotics.org/.

[39] "github," [Online]. Available: https://github.com/gazebosim/ros_gz.

[40] "ubuntu.," [Online]. Available: https://ubuntu.com/download/desktop.

[41] "etcher.balena," [Online]. Available: https://etcher.balena.io/.

[42] "linuxopsys," [Online]. Available: https://linuxopsys.com/install-ros-2-humble-on-ubuntu.

[43] "docs.ros," [Online]. Available: https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html.

[44] "gazebosim.org," [Online]. Available: https://gazebosim.org/docs/fortress/install_ubuntu/.

[45] "blender," [Online]. Available: https://docs.blender.org/manual/en/latest/getting_started/installing/linux.html.

[46] "davesroboshack," [Online]. Available: https://davesroboshack.com/the-robot-operating-system-ros/creating-ros2-packages/.