A Project Report on

Autonomous Navigation using Wheel Odometry

Submitted by

Mohd Farhan Haroon

Enrollment No.

2000100298

Department of Computer Science and Engineering

Integral Robotics Lab

Integral University, Kursi Road, Lucknow, Uttar Pradesh, India - 226026

Acknowledgement

The completion of this thesis represents a significant milestone in my academic journey. I extend my deepest gratitude to those whose guidance, encouragement, and support made this achievement possible.

Firstly, I express my sincere gratitude to the Executive Director of External Affairs and the CIED, Integral University, Mr. Syed Adnan Akhtar, whose expert knowledge, insightful feedback, and unwavering support helped to shape and refine my research. His guidance has been indispensable in the successful completion of this thesis.

My gratitude extends to my family, especially my parents, who have been an unwavering source of support and encouragement throughout my academic journey. Their steadfast love, guidance, and encouragement have been the driving force behind my success.

I would also like to acknowledge the support and camaraderie of my friends and colleagues. Their shared experiences, knowledge, and encouragement have been invaluable in shaping my academic and personal growth.

Last but not the least, I am extremely grateful to the Almighty for none of this would have been possible without His blessings and guidance in unimaginable ways.

Contents

List of Figures

Chapter 1

Introduction

1.1 About the Project

Autonomous navigation has been a rapidly growing field of research and development in recent years, with applications ranging from self-driving cars to warehouse automation. One popular platform for experimenting with autonomous navigation is the Turtle Bot 3, a small and affordable robot equipped with sensors and actuators.

In this project, we aim to explore the capabilities of the Turtle Bot3 for autonomous navigation using wheel Odometry. Wheel Odometry is a method of measuring the robot's movement by tracking the rotation of its wheels. By integrating this information with data from sensors such as LIDARS and Inertial Measurement Units, the robot can build a map of its surroundings and navigate autonomously to a desired location.

We will begin by understanding all the underlying concepts of kinematics and robotics, setting up the Turtle Bot3 and calibrating its sensors and wheel encoders. Then, we will develop a software pipeline to process sensor data and control the robot's movements. We will test the robot's navigation

capabilities in a variety of environments and scenarios, evaluating its performance and identifying areas for improvement. Through this project, we hope to gain a deeper understanding of autonomous navigation and contribute to the development of more advanced robotics systems.

Chapter 2

Differential Wheel Drive

2.1 What is Differential Drive

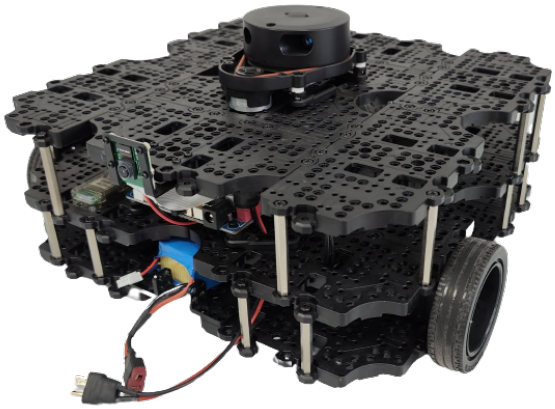


Figure 2.1: Differential Drive Robot

Differential drive is a type of propulsion system commonly used in robotics, particularly in mobile robots. It involves using two separate wheels that are independently driven by individual motors. By varying the speed and direction of the two wheels, the robot can move in any direction within a two-dimensional plane. [1] This type of drive is relatively simple and easy to implement, making it popular among hobbyists and researchers alike. However, one of the challenges of dif-

ferential drive is maintaining stability, especially when navigating uneven terrain or changing directions quickly. Nevertheless, with proper control algorithms and sensor feedback, differential drive systems can be highly effective for a wide range of robotic applications.

In this project, we use a differential drive robot known as the Turtle Bot 3 which will be discussed later on in the report. The robot has 2 drive wheels which are connected to 2 DC motors. On varying the individual velocities of the motors, the robot is able to move and turn in any direction.

2.2 Kinematics of a Differential Drive Robot

The kinematics of differential drive robots is concerned with the study of their motion, position, and orientation without considering the forces that cause that motion. Differential drive robots are a type of mobile robot that use two independently driven wheels to move and turn, and understanding their kinematics is essential for designing and controlling them. The forward and inverse kinematics of the robot determine its position and orientation in the world and the velocities of its wheels that will result in a desired motion. Odometry is used to estimate the robot's position and orientation based on the motion of its wheels, while motion planning is used to determine a path for the robot to follow that will allow it to reach a desired destination while avoiding obstacles.

In this chapter, we will be discussing about the Forward and Inverse Kinematics of the Differential Drive robots.

2.2.1 Forward Kinematics

The forward kinematics of differential drive robots is concerned with determining the robot's position and orientation in the world based on the motion of its wheels. This process involves using the velocities of the robot's wheels to calculate its linear and angular velocities and then integrating these velocities over time to determine its position and orientation. [1]

To derive the equations for the forward kinematics of a differential drive robot, we begin by considering the motion of its wheels. Let vr and vl be the velocities of the right and left wheels, respectively, and let L be the distance between the wheels. We assume that the wheels are rolling without slipping or skidding, and that the robot is moving on a flat surface.

The linear velocity of the robot, v, is given by:

$$v = (vr + vl)/2$$

This equation represents the average of the velocities of the right and left wheels and determines the linear speed of the robot.

Using these equations, we can determine the position and orientation of the robot at any point in time by integrating its velocity over time. Specifically, we can use the following equations:

$$x = x_0 + \int vcos(\theta)\, dt$$

$$y = y_0 + \int vsin(\theta)\, dt$$

$$\theta = \theta_0 + \int \omega\, dt$$

Where x and y are the coordinates of the robot's position, θ is the angle between the robot's orientation and a reference

axis, x_0 and y_0 are the initial coordinates of the robot's position, θ_0 is the initial orientation of the robot, and t is time.

These equations describe the robot's position and orientation in terms of its linear and angular velocities, as well as its initial position and orientation. By measuring the velocities of the robot's wheels and integrating these equations over time, we can track the robot's position and orientation as it moves through the world.

In summary, the forward kinematics of a differenOverall, differential drive robots are used in a wide variety of applications, from agriculture to entertainment, demonstrating their versatility and usefulness in many different fields.tial drive robot involves calculating the robot's position and orientation based on the motion of its wheels. By using the velocities of the robot's wheels to calculate its linear and angular velocities and integrating these velocities over time, we can determine the robot's position and orientation in the world. The equations for the forward kinematics of a differential drive robot are fundamental to the design and control of these robots and provide a powerful tool for navigating them through the world.

2.2.2 Inverse Kinematics

The inverse kinematics of a differential drive robot involves determining the required wheel velocities and orientations to move the robot to a desired position and orientation. [1] In contrast, forward kinematics involves calculating the robot's position and orientation based on the velocities and orientations of its wheels.

The relationship between forward kinematics and inverse kinematics for differential drive robots can be expressed using the following equations:

Forward Kinematics:

$$x = x_0 + (\frac{r}{2}) \times (\omega_r + \omega_l) \times cos(\theta)$$

$$y = y_0 + (\frac{r}{2}) \times (\omega_r + \omega_l) \times sin(\theta)$$

$$\theta = \theta_0 + (\frac{r}{L}) \times (\omega_r - \omega_l)$$

Inverse Kinematics:

$$\omega_r = \frac{(2v + \omega_L)}{2r}$$

where x and y are the robot's position coordinates, theta is its orientation, v is its linear velocity, ω is its angular velocity, ω_r and ω_l are the velocities of the right and left wheels, respectively, r is the radius of the wheels, and L is the distance between the wheels.

To understand the relationship between forward and inverse kinematics, consider the following example. Suppose we want to move the robot from its current position (x_0, y_0, θ_0) to a desired position (x_d, y_d, θ_d). To do this, we first need to calculate the required velocities of the robot's wheels using the inverse kinematics equations above.

Once we have the required wheel velocities, we can use the forward kinematics equations to calculate the robot's position and orientation as it moves towards its desired destination. By iteratively updating the wheel velocities based on the robot's current position and orientation, we can move the robot towards its desired destination.

In practice, there are several challenges associated with inverse kinematics for differential drive robots. One challenge

is the presence of wheel slippage, which can cause the robot to deviate from its desired trajectory. To account for this, some inverse kinematics algorithms incorporate feedback control to adjust the wheel velocities in real-time.

Another challenge is the non-linearity of the kinematic equations, which can make it difficult to compute exact solutions. To address this, some algorithms use numerical optimization methods, such as gradient descent or Newton's method, to iteratively refine the wheel velocities until the robot reaches its desired destination.

Overall, the relationship between forward and inverse kinematics is essential for controlling the motion of differential drive robots. By using a combination of forward and inverse kinematics, we can move the robot towards its desired destination while accounting for the non-linearity and uncertainties of the robot's motion.

2.3 Applications of Differential Drive Robots

Differential drive robots have a variety of applications in different fields. Here are some examples:

- **Agriculture**: Differential drive robots can be used in agriculture for tasks such as crop monitoring, irrigation, and pesticide spraying. For example, the Blue River Technology's "See and Spray" machine uses a combination of computer vision and differential drive to selectively apply herbicides to weeds in crop fields.
- **Search and Rescue**: Differential drive robots can be used in search and rescue operations in difficult terrains such as

rubble, snow, or water. The RoboCup Rescue robot competition features differential drive robots that are designed to navigate through disaster zones and locate victims.

- **Space Exploration**: Differential drive robots have been used for space exploration, such as NASA's Mars rovers Spirit, Opportunity, and Curiosity. These rovers use differential drive to navigate the rocky terrain and conduct scientific experiments.
- **Entertainment**: Differential drive robots are also used in the entertainment industry for performances and exhibits. The KUKA Robot Group developed a robot called "The Robot Animator" that uses differential drive to move around and create synchronized dance moves with human performers.
- **Education**: Differential drive robots are commonly used in educational settings to teach robotics and programming concepts. For example, the LEGO Mindstorms robot kit features a differential drive robot that can be programmed to perform various tasks using visual programming languages.

Overall, differential drive robots are used in a wide variety of applications, demonstrating their versatility and usefulness in many different fields.

Chapter 3

Navigation

3.1 What is Navigation

Navigation refers to the process of determining the position, direction, and movement of an object or a person relative to a reference point or destination. Navigation has been an essential aspect of human civilization since ancient times, when people used stars, landmarks, and other natural features to navigate over land and sea. In modern times, navigation has become even more critical with the advent of technologies such as GPS, radar, and sonar, which have made it possible to navigate with greater precision and accuracy. [2]

Navigation finds application in a wide range of domains, including aviation, maritime, land transportation, space exploration, and more. Effective navigation requires a combination of technical skills, tools, and knowledge of the environment, making it an interdisciplinary field that draws from physics, mathematics, geography, and other disciplines.

3.2 How do Robots Navigate

Robot navigation refers to the process of a robot moving from one point to another in its environment while avoiding obstacles and following a pre-determined path. Navigation is a crucial aspect of robotics, as it enables robots to perform tasks such as exploration, surveillance, inspection, and delivery in various environments. [2]

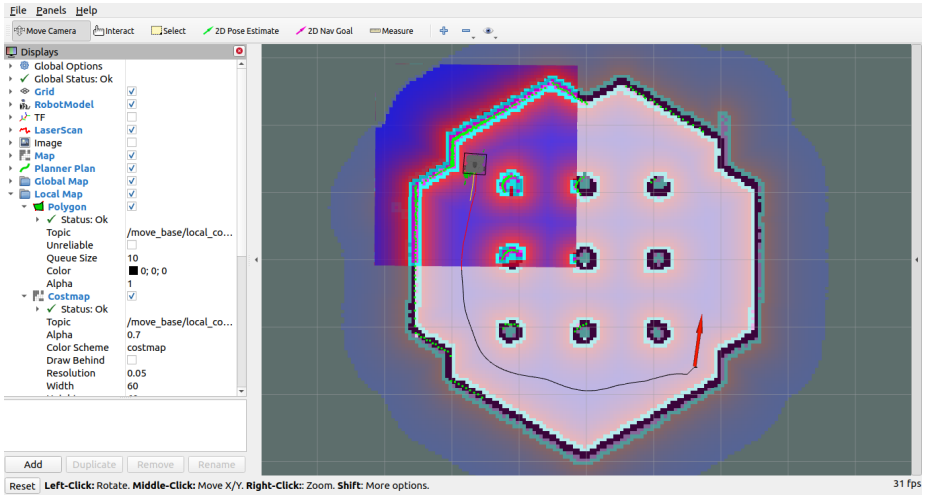


Figure 3.1: Navigation using a costmap

There are various navigation techniques that robots can use to navigate their environment. Some of these techniques include:

- **Reactive Navigation**:

  Reactive navigation is a type of navigation in which the robot reacts to the immediate environment without planning a path in advance. The robot uses sensors such as cameras and sonar to detect obstacles and adjust its path

to avoid them. Reactive navigation is useful in environments where the robot needs to move quickly and avoid obstacles that may change in real-time.

- **Map-Based Navigation**:

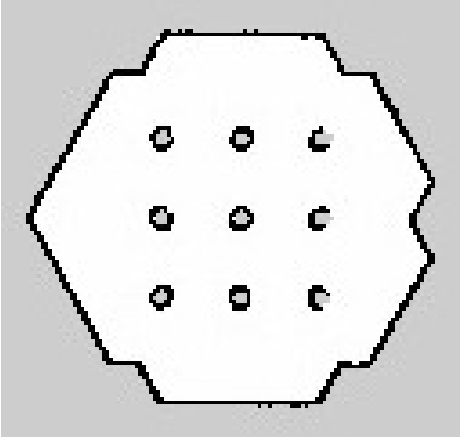


Figure 3.2: Map used for Map based navigation

Map-based navigation involves the robot creating a map of its environment and using that map to plan a path to a destination. The robot uses sensors such as lidar and cameras to create a map of the environment and then uses that map to plan the path. Map-based navigation is useful in environments where the robot needs to navigate through a complex environment while avoiding obstacles.

- **SLAM Navigation**:

Simultaneous Localization and Mapping (SLAM) is a type of navigation in which the robot creates a map of its environment while simultaneously determining its location within that environment. The robot uses sensors such as

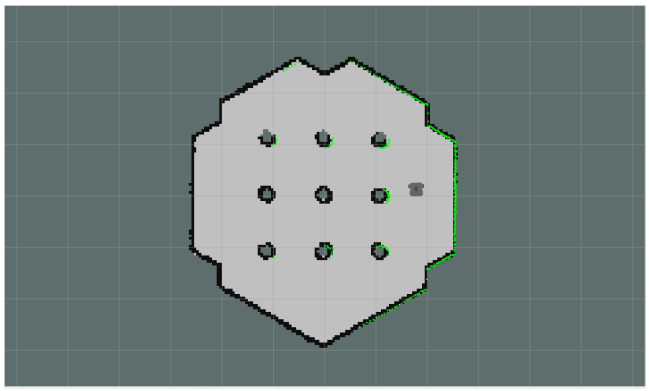


Figure 3.3: SLAM for navigation

LIDAR, cameras, and sonar to create a map of the environment and determine its location. SLAM navigation is useful in environments where the robot needs to operate in an environment that is not mapped in advance, such as in a disaster area or a new construction site.

- **Hybrid Navigation**:

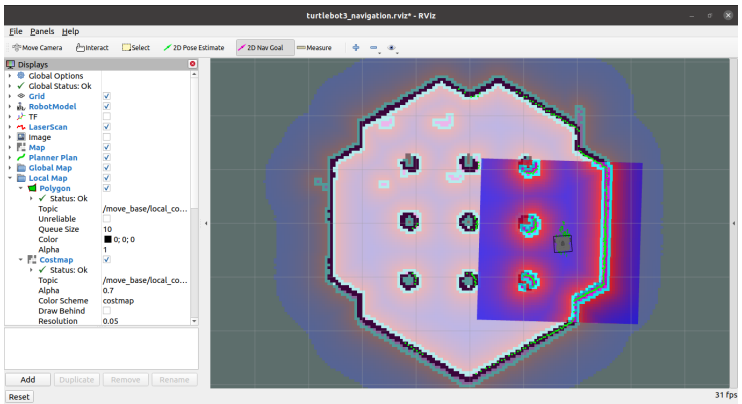


Figure 3.4: Active path planning while navigation

Hybrid navigation combines reactive and map-based navigation techniques to provide a more robust navigation

system. The robot uses a map to plan a path to a destination and adjusts its path in real-time based on its immediate environment. Hybrid navigation is useful in environments where the robot needs to navigate through a complex environment that may change in real-time.

In conclusion, robot navigation is a critical aspect of robotics that enables robots to perform tasks in various environments. The navigation technique used by a robot depends on the robot's design, purpose, and the environment it operates in. Robots can use reactive navigation, map-based navigation, SLAM navigation, or hybrid navigation to navigate their environment and perform tasks autonomously. As robots become more advanced and intelligent, their navigation capabilities will continue to improve, making them more versatile and useful in a wide range of applications.

3.3 Sensors used for Navigation

Robot navigation refers to the process by which a robot is able to move around in an environment, avoid obstacles, and reach a desired destination. In order to navigate accurately and efficiently, a variety of methods and sensors are used. Here are some of the most commonly used methods and sensors for robot navigation:

- **Global Positioning System (GPS)**: GPS is a satellite-based navigation system that is used to determine the robot's location on the Earth's surface. GPS is commonly used in outdoor environments where the robot has access to a clear view of the sky.

- **Inertial Measurement Unit (IMU)**: An IMU is a sensor that measures the robot's acceleration and angular velocity. By integrating these measurements over time, the robot's position and orientation can be determined. IMUs are commonly used in indoor environments where GPS signals are weak or unavailable.

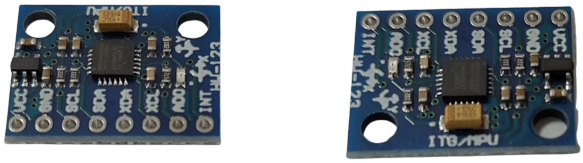


Figure 3.5: Inertial Measurement Unit

- **LIDAR**: LIDAR (Light Detection and Ranging) is a sensor that uses laser beams to create a three-dimensional map of the robot's environment. LIDAR is commonly used to detect obstacles and to create maps of indoor environments.

Figure 3.6: LIDAR sensor

- **Ultrasonic sensors**: Ultrasonic sensors use sound waves to detect the presence of obstacles. These sensors are

commonly used to detect objects that are close to the robot.

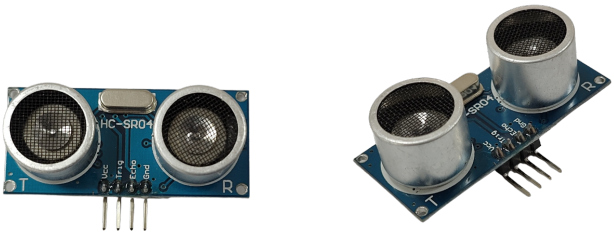


Figure 3.7: Ultrasonic distance Sensor

- **Infrared sensors**: Infrared sensors detect the presence of obstacles by measuring the amount of infrared radiation that is reflected back to the sensor. These sensors are commonly used to detect objects that are within a few feet of the robot.

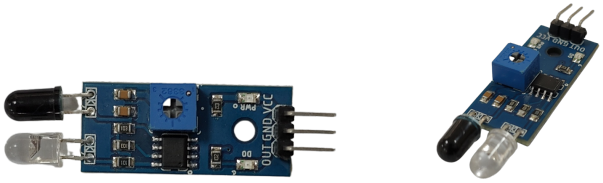


Figure 3.8: Infrared Sensor

- **Camera sensors**: Cameras are used to capture images of the robot's environment. These images can be used to create maps of the environment, detect obstacles, and locate landmarks.

- **Magnetic sensors**: Magnetic sensors are used to detect magnetic fields. These sensors are commonly used to detect metal objects or to navigate in environments where GPS signals are weak or unavailable.

In addition to these sensors, robots can also use a variety of navigation algorithms to plan their paths and avoid obstacles. These algorithms can range from simple obstacle avoidance to complex path planning and localization algorithms.

In this project, the robot used is equipped with a 2-D LIDAR which is used to crejate a 2-D map of the surrounding and then precise and calculated movement commands are decided to make the robot move from the initial to the target position.

In conclusion, robot navigation is a complex process that requires the use of a variety of sensors and algorithms. By combining these methods, robots are able to navigate their environments with accuracy. As technology continues to advance, we can expect to see even more sophisticated methods and sensors being developed for robot navigation.

Chapter 4

Odometry

4.1 What is Odometry

Odometry is the process of determining the position and orientation of a mobile robot by analyzing its motion. It is a fundamental technique used in robotics and autonomous navigation to estimate the robot's current location in a given environment. [3]

Odometry is typically implemented by using sensors to measure the rotation of the robot's wheels and the distance traveled by each wheel. By analyzing this data, the robot can calculate its change in position and orientation over time. Odometry is commonly used in a variety of applications, such as indoor navigation, outdoor mapping, and mobile robotics.

Although Odometry is a simple and effective method for robot localization, it has its limitations and is often combined with other techniques, such as simultaneous localization and mapping (SLAM), to improve accuracy and robustness.

4.2 Types of Odometry

Odometry is a process of measuring the movement of a robot based on the motion of its wheels or other sensors. There are different types of Odometry methods that vary in their accuracy, complexity, and applicability to different types of robots and environments. Some of the commonly used types of Odometry include:

- Laser Odometry
- RADAR Odometry
- Inertial Odometry
- Wheel Odometry
- Visual Odometry

In this project, we will discuss mainly about Wheel Odometry as our robot uses Wheel Odometry as the main source of localization for navigation.

4.3 Wheel Odometry

Wheel Odometry is a method of estimating the position and orientation of a mobile robot based on the rotation of its wheels. [3] The technique is based on the assumption that the robot's motion is mainly determined by the movement of its wheels. By monitoring the rotation of the wheels and knowing their geometry, it is possible to estimate the distance and direction traveled by the robot.

To completely understand Wheel Odometry, let us take an example.

Figure 4.1: Source of Wheel (no Visual) Odometry

Consider a blind person on a wheel chair. Let us assume that the person knows the radius of the wheels and also knows when the wheels make a complete rotation. From this information, the person can calculate the circumference of the wheels by the formula

$$c = 2\pi r$$

where c is the circumference, r is the radius of the wheel and π is the constant 3.14.

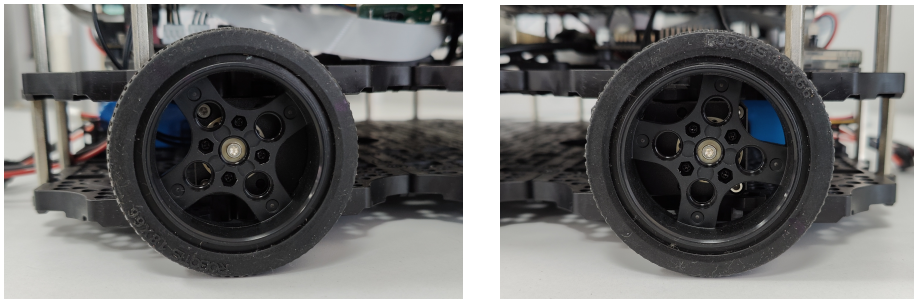


Figure 4.2: Robot wheels for Wheel Odometry

From the above information, the person can know exactly when the chair has moved forward by a unit distance (=c)

when the wheels make 1 complete rotation (assuming the distance between the wheels remains constant and both the wheels move at the same angular velocity).

For example, let us assume the wheel radius to be 50 cm or 0.5 m. Now, the circumference c of the wheel according to $c = 2\pi r$ will be

$$c = 2 \times \pi \times 0.5$$
$$\Rightarrow c = \pi$$
$$\Rightarrow 3.14 \ m$$

Since the person knows when the wheel makes 1 complete rotation, the person can move in the forward direction exactly by 3.14 m without even looking or measuring by any means.

Note: Here, the blindness of the person is symbolic of the fact that no means of Visual, Laser or Radar Odometry is used in the above scenario.

The system discussed above is effective only in 1 dimensional planes. However, if the individual angular velocities of the wheels are controlled in a calculated way (as mentioned in Ch 2 - Differential Drive Kinematics), a precise and accurate control system for navigation in 2 dimensional planes can be achieved which is demonstrated later on in this project.

Chapter 5

Robot Operating System (ROS)

5.1 What is ROS

Robot Operating System, commonly known as ROS, is a powerful platform for developing robotic applications. [4] It provides a collection of libraries and tools that facilitate the creation of complex robotic systems, making it easier for developers to build, test, and deploy robot applications.

ROS is widely used in both academia and industry, and has become the de facto standard for developing robotic software. With its modular architecture, ROS enables developers to build complex robotic systems by connecting individual software modules that perform specific functions, allowing for greater flexibility and reusability.

5.2 How does it work

ROS (Robot Operating System) is a framework that is implemented in multiple languages like python and C++. ROS has multiple components that aid in the proper and smooth functioning of Robotc components. Some of those components are:

- Nodes
- Topics
- Publishers
- Subscribers
- Services
- Actions
- Messages

Let us discuss these components in detail.

5.2.1 Nodes

In ROS, a node is a computational unit that performs a specific task or set of tasks. [4] Nodes can be thought of as individual programs that communicate with each other to accomplish a larger goal. For example, one node might read sensor data from a robot, while another node might control the robot's motors. Nodes can be written in a variety of programming languages, including C++, Python, and Java.

Nodes in ROS communicate with each other using a publish-subscribe messaging model. In this model, a node that produces data (the publisher) sends that data to a specific topic, and any nodes that are interested in that data (subscribers) can receive it from the topic. This decoupling of nodes allows for a more modular and flexible system, as nodes can be added or removed without affecting the entire system (more discussion on this later).

5.2.2 Topics

ROS Topics are a fundamental communication mechanism in the Robot Operating System (ROS) that enable different components of a robotic system to exchange data with one another. [4] Topics are essentially message-passing channels where a publisher node sends messages to a topic, and one or more subscriber nodes receive those messages.

ROS Topics use a publish-subscribe model, meaning that publishers and subscribers do not need to know about each other's existence or state. Instead, they communicate through the common message format that is defined for each topic. This allows for highly decoupled communication between different parts of a robotic system, which is important in complex robotic systems where different components may need to interact with each other without direct knowledge of each other.

ROS Topics can be used to transmit a wide variety of data, including sensor readings, control commands, and status updates. They can also be used to synchronize data between different parts of a system, making it easier to ensure that all components are working together effectively.

Overall, ROS Topics provide a powerful and flexible communication mechanism for robotic systems that can help simplify the development and integration of complex robotic systems.

5.2.3 Publishers

Publishers are responsible for sending data to a particular topic on a ROS network. [4] They typically generate and transmit messages at a fixed rate or whenever a specific event occurs.

The messages sent by publishers may contain sensor data, control commands, or other types of information that need to be shared with other nodes in the system.

ROS Publishers operate on a publish-subscribe model, where nodes that wish to receive messages from a particular topic subscribe to that topic. When a publisher sends a message to a topic, it is immediately delivered to all subscribers that are currently connected to that topic. This allows multiple nodes to receive the same data at the same time, which can be useful for tasks such as sensor fusion and data aggregation.

ROS Publishers are designed to be flexible and extensible. They can be programmed to send messages of various types and sizes, and can be configured to transmit data over different communication protocols. This flexibility makes ROS Publishers suitable for a wide range of robotic applications, from simple single-robot systems to complex multi-robot networks.

5.2.4 Subscribers

ROS Subscribers are another core component of the Robot Operating System (ROS) that enable a node to receive messages from a specific topic. [4] In ROS, a topic is a named communication channel that allows nodes to exchange data, and a node is a process that performs a specific function in a robotic system.

Subscribers are responsible for receiving and processing messages that are published to a particular topic on a ROS network. They typically listen for messages on a topic and respond to them when they arrive. The messages received by subscribers may contain sensor data, control commands, or other types of information that need to be processed by the

node.

ROS Subscribers operate on a publish-subscribe model, where nodes that wish to receive messages from a particular topic subscribe to that topic. When a publisher sends a message to a topic, it is immediately delivered to all subscribers that are currently connected to that topic. This allows multiple nodes to receive the same data at the same time, which can be useful for tasks such as sensor fusion and data aggregation.

ROS Subscribers are designed to be flexible and extensible. They can be programmed to receive messages of various types and sizes, and can be configured to handle data received over different communication protocols. This flexibility makes ROS Subscribers suitable for a wide range of robotic applications, from simple single-robot systems to complex multi-robot networks.

In summary, ROS Subscribers are a critical component in ROS that allow nodes to receive messages from a specific topic and process them. They play a key role in enabling nodes to communicate with one another in a flexible and decoupled manner, which is essential for building complex robotic systems.

5.2.5 Services

ROS Services are an essential part of the Robot Operating System (ROS) architecture, which is a popular open-source robotics middleware platform used for building and controlling robots. [4] Services are a way for nodes (components) in a ROS system to communicate with each other and request or provide specific functionalities.

In ROS, nodes can provide services, which are named functionalities that other nodes can request to use. A service call is a synchronous operation, which means that the calling node will wait for a response before proceeding with other tasks. When a node calls a service, it sends a request message to the node providing the service, which then processes the request and sends back a response message.

ROS Services are useful when a node needs to perform a task that requires specific inputs or parameters. For example, a node that controls a robot arm may provide a ”move to position” service that takes in a set of coordinates as input and moves the arm accordingly. Another node can then call this service to move the arm to a specific location.

5.2.6 Actions

ROS Actions are another important component of the Robot Operating System (ROS) architecture, which enables robust and flexible communication and coordination between different nodes in a robotic system. [4] Unlike ROS Services, which are synchronous and request-response based, ROS Actions are asynchronous and goal-oriented.

In ROS, an action is a named process that a node can request another node to perform. The process can take a long time to complete, or it may require feedback at intervals during execution. An action can have multiple steps, and each step may take a different amount of time to execute.

ROS Actions are useful when a task requires continuous feedback and monitoring, such as controlling a robot to follow a specific path or navigate through an environment. When a node requests an action, it sends a goal message to the node re-

sponsible for performing the action, which then starts executing the task. The node performing the action sends feedback messages to the requesting node at regular intervals, indicating the progress of the task. Once the action is complete, the node responsible for performing the action sends a result message back to the requesting node.

5.2.7 Messages

ROS Messages are a fundamental component of the Robot Operating System (ROS) architecture, which is an open-source middleware platform used for building and controlling robots. [4] Messages are the primary means of communication between nodes (components) in a ROS system, allowing them to exchange data and information.

In ROS, a message is a data structure that consists of typed fields. Each field in a message can be a primitive data type, such as integers or strings, or it can be a more complex data structure, such as a nested message or an array. Messages are defined using the ROS Message Description Language (MDL), which is a simple and flexible way to define the structure of messages.

ROS Messages are used in a variety of ways in a ROS system, such as passing sensor data, control commands, or status updates between nodes. When a node publishes a message, it sends the message to a topic, which is a named channel for exchanging messages. Other nodes that are interested in receiving messages from that topic can subscribe to it, and they will receive the messages as they are published.

5.3 Gazebo simulations

5.3.1 What is Gazebo

Gazebo is a widely-used open-source robot simulator that is often integrated with the Robot Operating System (ROS) to provide a powerful and flexible platform for simulating robotic systems. [4] It allows developers to test and debug their robotic applications in a virtual environment before deploying them on real hardware.

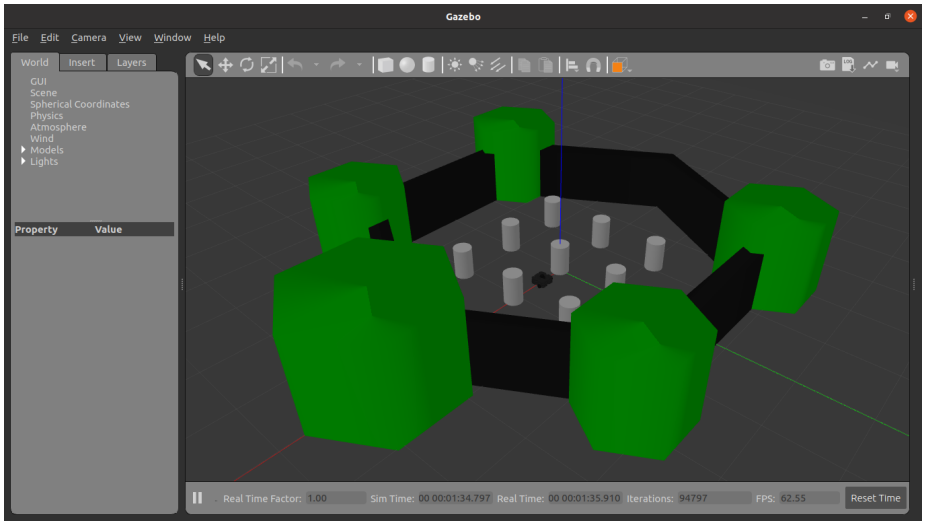


Figure 5.1: Gazebo simulation of Turtle Bot 3 - Waffle Pi

In ROS, Gazebo can be used to simulate various aspects of a robotic system, including sensors, actuators, and the environment. [4] It provides a high-fidelity simulation environment that allows developers to test their algorithms and control strategies in a realistic setting. Gazebo also supports the integration of various sensor models, such as cameras and LiDARs, enabling developers to generate sensor data that closely resembles real-world sensor data.

One of the key advantages of using Gazebo with ROS is the ability to simulate complex robotic systems that involve multiple sensors, actuators, and control loops. Gazebo can simulate various types of robots, such as humanoid robots, drones, and ground robots, making it a versatile tool for testing different types of robotic systems.

Overall, Gazebo is a valuable tool for developers who are working with ROS and building complex robotic systems. Its ability to simulate complex robotic systems in a virtual environment enables developers to iterate and test their applications quickly and efficiently, ultimately leading to more reliable and robust robotic systems.

5.3.2 URDF files

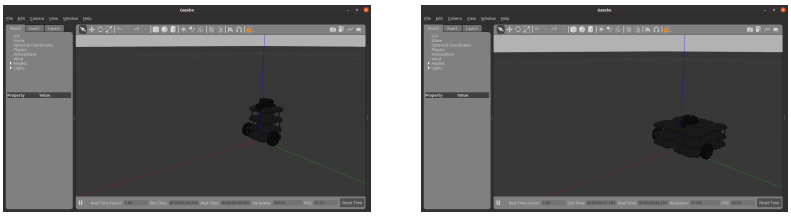


Figure 5.2: URDF for TB3 Burger and Waffle-Pi in Gazebo

URDF stands for ”Unified Robot Description Format,” and it is a standardized format for describing robotic models in a structured way. [4] In Gazebo, URDF files are used to define the physical properties and kinematics of robots, including their geometry, joint types, and joint limits. This allows Gazebo to simulate the behavior of the robot accurately, including its movements, interactions with the environment, and sensor readings.

URDF files are essential for robot simulation and are used

in a variety of applications, such as robotics research, robot control algorithm development, and educational purposes. They provide a convenient and portable way to describe robots, which can be easily shared and modified by the community. Overall, URDF files enable efficient and accurate robot simulation in Gazebo, making it an indispensable tool for robot development and testing.

5.4 RViz

5.4.1 What is RViz

RViz is a 3D visualization tool in ROS (Robot Operating System) that provides a graphical interface for visualizing data from sensors and robot models in real-time. [4] It allows users to see what the robot is perceiving and doing, providing valuable insight for development, testing, and debugging purposes.

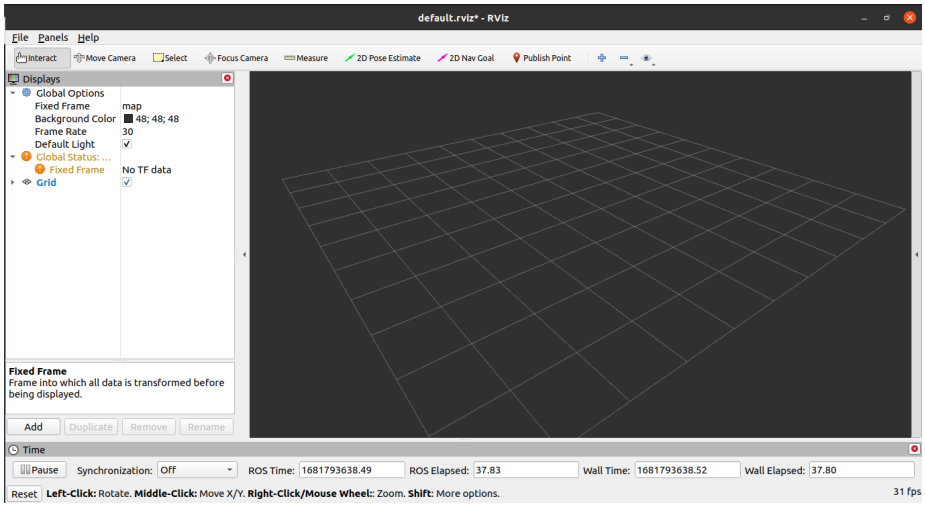


Figure 5.3: RViz window

RViz can display various types of data, including point clouds, laser scans, images, and robot models. Users can also interact with the visualization by moving the robot or objects, changing the camera view, and modifying the properties of the displayed data. RViz can also be used to visualize the robot's sensor data and the output of its algorithms, such as path planning or object detection.

RViz is a powerful tool for robotics development and testing and is widely used in research, education, and industrial applications. Its intuitive interface and versatility make it accessible to users with various levels of experience, from beginners to advanced users. Overall, RViz is an essential component of ROS that provides a critical window into the robot's operation and behavior.

5.4.2 RViz for Data Visualization

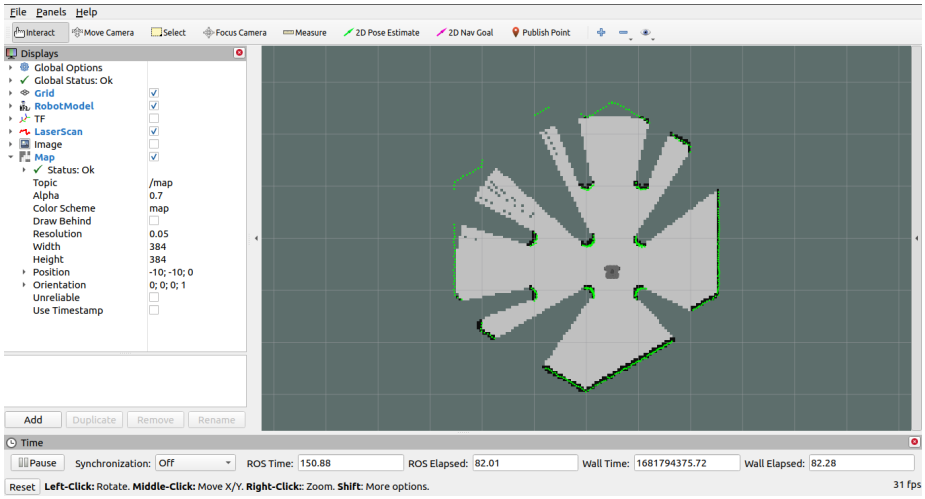


Figure 5.4: Visualising SLAM data in RViz from Gazebo simulation

RViz is a powerful tool for data visualization in ROS. It

can display various types of data in real-time, including point clouds, laser scans, images, and robot models. With RViz, users can visualize the robot's sensor data, the output of its algorithms, and the overall state of the robot.

One of the primary uses of RViz is to visualize the environment around the robot. This can be done using sensors such as LIDAR or cameras, and the resulting data can be displayed as point clouds or images in RViz. Users can then manipulate the view of the data to better understand the environment and how the robot is navigating through it.

Another use of RViz is for debugging and testing. Users can use RViz to monitor the robot's sensors and output, ensuring that they are functioning correctly. RViz can also be used to visualize the robot's planned paths and trajectories, making it easier to identify errors and improve the robot's performance.

Overall, RViz is an essential tool for data visualization in ROS, allowing users to better understand the environment and the robot's behavior. With RViz, users can quickly and easily visualize data, making it an invaluable tool for robotics development and testing.
‘

Chapter 6

Turtle Bot 3

6.1 Turtle Bot 3 Waffle-Pi

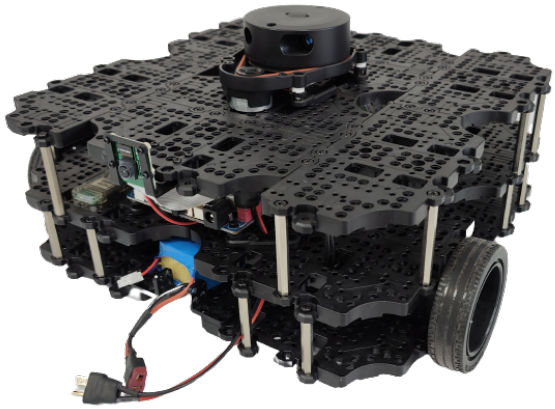


Figure 6.1: Turtle Bot 3 Waffle Pi

Turtle Bot 3 Waffle Pi is a popular and versatile mobile robot platform that is designed for education, research, and hobbyist robotics. [5] It is an upgrade from the Turtle Bot 2 and features a more compact and lightweight design, making it more portable and easier to use.

The Turtle Bot 3 Waffle Pi is powered by a Raspberry Pi, which provides a familiar and accessible platform for pro-

gramming and controlling the robot. It also comes with a range of sensors, including a 360-degree LIDAR sensor, a camera, and an IMU, which enable it to navigate its environment and interact with objects.

The Turtle Bot 3 Waffle Pi is compatible with ROS (Robot Operating System), which provides a powerful and flexible framework for developing robot applications. ROS allows users to program the robot's behaviors, interface with its sensors and actuators, and visualize its data using tools such as RViz.

Overall, the Turtle Bot 3 Waffle Pi is an excellent platform for learning and experimenting with robotics, as well as for conducting research in areas such as perception, navigation, and control. Its versatility and ease of use make it an attractive choice for both beginners and advanced users.

6.2 Devices and Sensors

The Turtle Bot 3 Waffle-Pi consists of a number of components like sensors, peripheral devices, motors, chassis plates etc. Let us briefly discuss these devices and their functions. [5]

6.2.1 Open CR 1.0

OpenCR 1.0 is an open-source micro-controller board that is used in the Turtle Bot 3 robot platform. It provides a powerful and flexible platform for controlling the robot's hardware, including its motors, sensors, and other peripherals. [5]

One of the primary uses of Open CR 1.0 in Turtle Bot 3 is

for motion control. The board can interface with the robot's motors and encoders, allowing users to program the robot's movements and behaviors. This can be done using a variety of programming languages and frameworks, including ROS, Arduino, and C/C++.

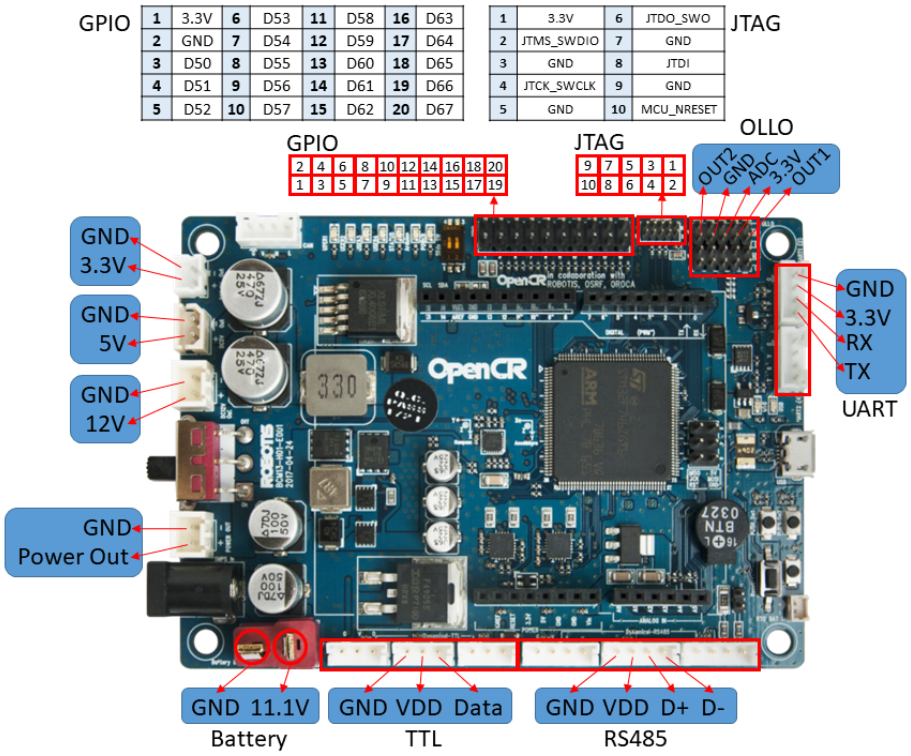


Figure 6.2: Open CR 1.0 Pin diagram

OpenCR 1.0 also includes a range of communication interfaces, such as USB, UART, and CAN, which enable the robot to interface with external devices and sensors. This makes it easy to add additional sensors and peripherals to the robot, expanding its capabilities and functionality.

Another use of OpenCR 1.0 is for sensor data processing. The board can interface with the robot's sensors, such as the

LiDAR, camera, and IMU, and process the data in real-time. This can be used for applications such as object detection, localization, and mapping.

Overall, OpenCR 1.0 is an essential component of the TurtleBot 3 platform, providing a powerful and flexible platform for controlling the robot's hardware and interfacing with external devices. Its versatility and compatibility with various programming languages and frameworks make it an attractive choice for robotics research, education, and hobbyist projects.

6.2.2 Raspberry Pi 4 - B

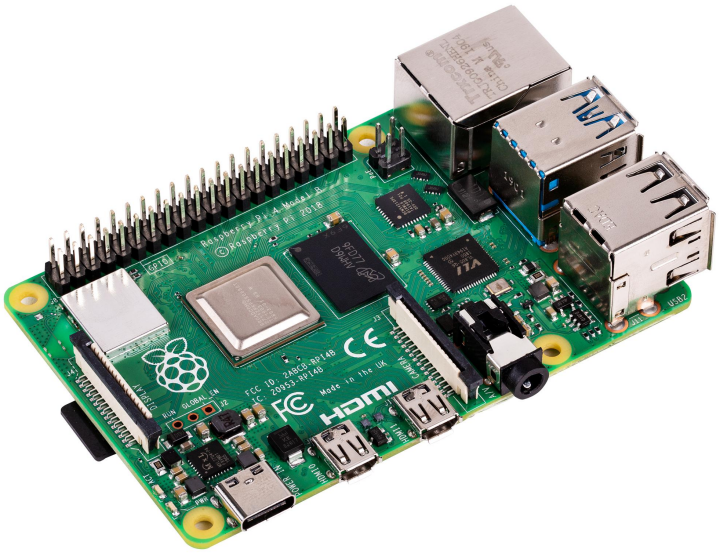


Figure 6.3: Raspberry Pi 4-B

Raspberry Pi 4 - B is a versatile and powerful single-board computer that can be used in a wide range of applications. One such application is in Turtle Bot 3, which is a popular open-source robotics platform for education and research. [5]

The Raspberry Pi 4 - B can be used as the brain of the Turtle Bot 3, controlling its motors, sensors, and other components. It can run a variety of software packages, including ROS (Robot Operating System), which provides a rich set of tools for programming and controlling robots.

One of the advantages of using Raspberry Pi 4 - B in Turtle Bot 3 is its small size and low power consumption, which make it ideal for mobile robotics applications. It also has built-in Wi-Fi and Bluetooth connectivity, which can be used to communicate with other devices and sensors.

Additionally, the Raspberry Pi 4 - B has a powerful quad-core processor, up to 8GB of RAM, and multiple USB ports, which make it capable of handling complex computations and data processing tasks. This makes it an ideal choice for running AI and machine learning algorithms on the Turtle Bot 3.

Overall, the Raspberry Pi 4 - B is a powerful and flexible platform for building and controlling robots, and its compatibility with ROS and other software packages makes it an ideal choice for the Turtle Bot 3.

6.2.3 Dynamixel

DYNAMIXEL XM430-W250-T is a high-performance servo motor commonly used in robotic applications. It is often used in the Turtle Bot 3 platform as it provides precise control and feedback for the robot's joints. [5]

The DYNAMIXEL XM430-W250-T servo motor works by using a PID control algorithm to accurately and quickly position the servo's output shaft to the desired angle. It can rotate continuously or be set to specific angles, and it also provides

Figure 6.4: Dynamixel XL-430-W250-T

feedback on its position, speed, and temperature to the controlling device.

Turtle Bot 3 uses the ROS (Robot Operating System) framework to control the DYNAMIXEL motors. The ROS software sends commands to the DYNAMIXEL motor through the Open CR 1.0 board, which then converts those commands into signals that the motor can understand.

With the help of the DYNAMIXEL XM430-W250-T servo motor, the Turtle Bot 3 can perform various tasks such as navigating, mapping, and manipulating objects. Its high-performance capabilities and precise control make it an essential component in the Turtle Bot 3 platform.

6.2.4 LIDAR

The LDS-02 LIDAR is a laser sensor commonly used in the Turtle bot 3 robot to map the surrounding environment and aid in autonomous navigation. This sensor is compact and

lightweight, making it a perfect fit for mobile robots like the Turtle Bot 3. [5]

Figure 6.5: RP LIDAR

The LDS-02 LIDAR works by emitting a laser beam that scans the environment, creating a 360-degree map of the area. The sensor measures the time it takes for the laser beam to bounce back from objects in the environment, which provides precise distance measurements. The data from the sensor is then processed and used by the robot's software to create a map of the environment, which the robot can use to navigate autonomously.

The LDS-02 LIDAR has a range of up to 12 meters and a scanning rate of 300 rotations per second, which makes it ideal for use in a variety of environments. The sensor can detect objects as small as 1 cm in size, which allows the robot to navigate through tight spaces without colliding with obstacles.

In addition to mapping and navigation, the LDS-02 LIDAR can be used for obstacle detection and avoidance. The sensor can detect obstacles in the environment, and the robot's soft-

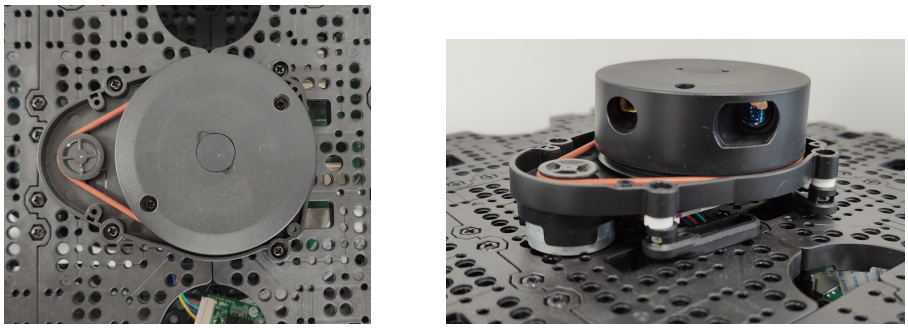


Figure 6.6: LDS-02 LIDAR

ware can adjust its trajectory to avoid collisions.

The LDS-02 LIDAR is easy to use and can be easily integrated into the Turtle bot 3 robot. The sensor is connected to the robot's computer through a USB cable, and the robot's software is configured to use the sensor data for mapping and navigation.

6.2.5 RPi Camera

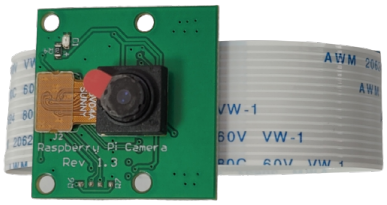


Figure 6.7: Raspberry Pi Camera

The Raspberry Pi (RPi) camera is a popular camera module that can be used with the Turtle bot 3 robot platform. The RPi camera is a small, lightweight camera that can capture high-

resolution images and videos. It is designed specifically to work with the Raspberry Pi microcomputer and can be easily connected to the Raspberry Pi board. [5]

The Turtle Bot 3 is a popular low-cost robot platform that can be used for education, research, and hobbyist projects. It is designed to be easy to use and is equipped with a variety of sensors, including a camera. The RPi camera can be used with the Turtle bot 3 to capture images and videos, which can then be processed and analyzed using software.

Figure 6.8: RPi Camera module

The RPi camera is an excellent choice for use with the Turtle Bot 3 because it is inexpensive, easy to use, and provides high-quality images and videos. It can be used for a variety of applications, including object detection, navigation, and surveillance. Additionally, the RPi camera is compatible with a variety of software libraries, including OpenCV and Python, which makes it easy to use with the Turtle bot 3 platform.

6.3 Tele-operation

Turtle Bot 3 tele-operation is the process of controlling the movement of the Turtle Bot3 robot remotely using a computer or a mobile device. The tele-operation process is facilitated by the ROS (Robot Operating System) framework, which allows

for the transmission of commands and data between the robot and the remote device. [5]

Some of the pre-requisites of Tele-operation in Turtle Bot 3 is that ROS should be installed on the host PC and the Turtle Bot 3 should be setup according to the instructions given on the official website of Robotis.

Once the above requirements have been fulfilled, the following steps can be followed in order to tele-operate the Turtle Bot 3 (as mentioned on the official website of Robotis):

- **Run roscore on host PC**

  Start the roscore service on the host PC

  ```
  $ roscore
  ```

- **Connect with Raspberry Pi**

  Open a new terminal and connect to the remote PC

  ```
  $ ssh ubuntu@{IP_ADDRESS_OF_RASPBERRY_PI}
  ```

- **Run Bringup**

  Run the Bringup on the Raspberry Pi by the following commands

  ```
  $ export TURTLEBOT3_MODEL=${TB3_MODEL}
  $ roslaunch turtlebot3_bringup turtlebot3_robot.launch
  ```

  TB3_MODEL is the Turtle Bot 3 model like *burger* or *waffle_pi*

- **Run Tele-operation**

  Run the tele-operation node on the host PC

```
$ export TURTLEBOT3_MODEL=${TB3_MODEL}
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

  Once the tele-operation node is running, the Turtle Bot 3 can be remotely controlled by the keyboard at the host PC.

```
Control Your Turtlebot3
Moving around
   w
a___s___d
    x
w/x : increase/decrease linear velocity
a/d : increase/decrease angular velocity
space key, s : force stop
CTRL-C to quit
```

Chapter 7

Simultaneous Localisation and Mapping (SLAM)

7.1 What is SLAM

SLAM stands for Simultaneous Localization and Mapping. It is a technique used in robotics to create a map of an unknown environment while simultaneously keeping track of the robot's position within that environment. [6]

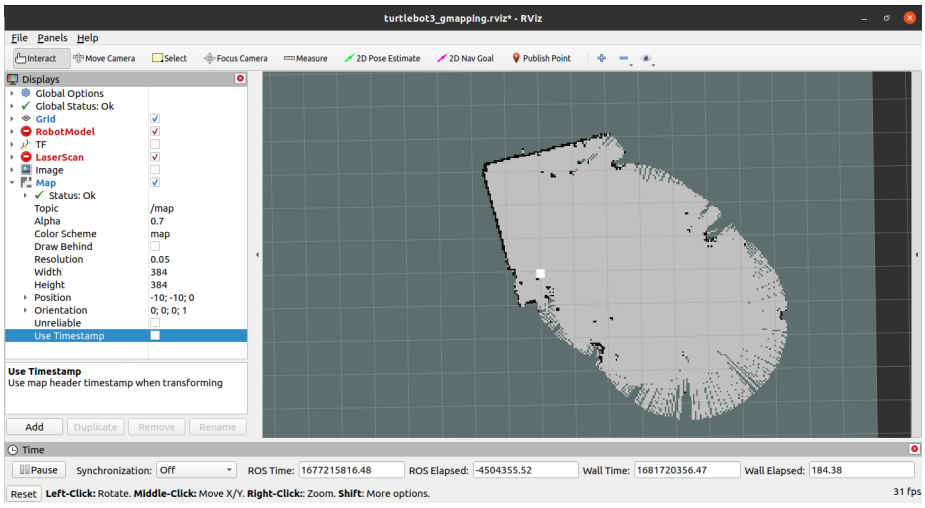


Figure 7.1: SLAM visualised in RViz

SLAM algorithms use data from sensors such as CAMERAs, LIDARs, and SONARs to construct a map of the environment. The algorithm also estimates the robot's position and orientation relative to the map, allowing it to navigate through the environment.

SLAM is an important problem in robotics because it enables robots to operate autonomously in unknown or changing environments. This is particularly useful in applications such as search and rescue, exploration, and automated manufacturing.

There are many different approaches to SLAM, ranging from probabilistic methods to geometric methods. These algorithms have different strengths and weaknesses, and the choice of algorithm depends on the specific application and the characteristics of the sensors used.

7.2 SLAM with Turtle Bot 3

Turtle Bot 3 comes with a range of sensors such as a 360-degree LIDAR, RGB-D camera, and an IMU. These sensors provide valuable data to the SLAM algorithm to build an accurate map of the environment and estimate the robot's position. [6]

There are several SLAM algorithms that can be used with Turtle Bot 3, including G-Mapping, Hector SLAM, and Cartographer. These algorithms differ in their computational requirements, accuracy, and complexity.

With the help of SLAM, Turtle Bot 3 can perform tasks such as autonomous navigation, exploration, and object de-

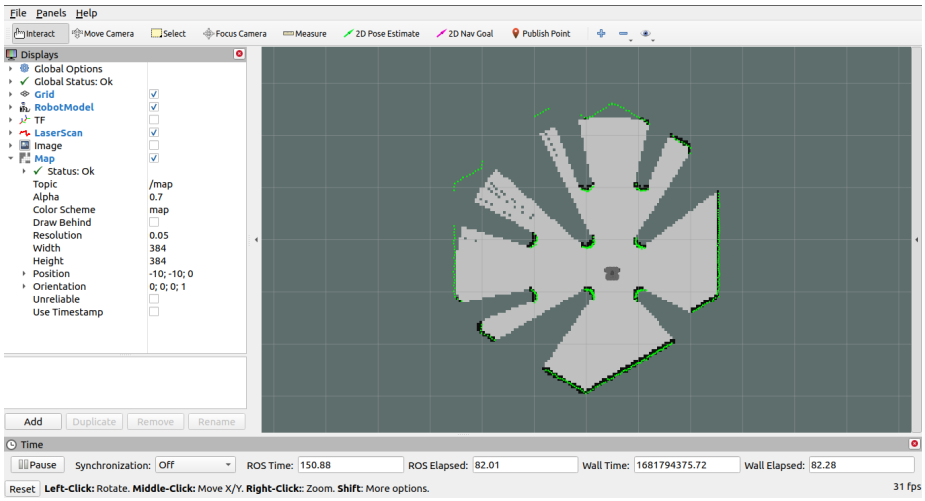


Figure 7.2: SLAM with Turtle Bot 3

tection. We use SLAM in our project for making a 2-D map of the surroundings and navigating using that map (more on this later).

7.2.1 Tele-operation

SLAM is a technique of self localization that we use in this project. It works by creating a 2-D map of the surroundings by applying the SLAM algorithm. [6]

For making the map of the surroundings, the LIDAR on the Turtle Bot 3 is used. Once the SLAM node is ready to be launched, we launch the tele-operation node first.

- Run Bringup on the Raspberry pi.

```
$ ssh pi@{IP_ADDRESS_OF_RASPBERRY_PI}
$ export TURTLEBOT3_MODEL=${TB3_MODEL}
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

- Open a new terminal and run tele-operation

```
$ export TURTLEBOT3_MODEL=${TB3_MODEL}
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

The tele-operation node is required because as the SLAM node is launched, the LIDAR starts creating a map of the surroundings which can be visualised in RViz. In order to make the map of the complete area, the Turtle Bot 3 needs to be guided around using tele-operation to complete the map. [6]

7.2.2 SLAM node

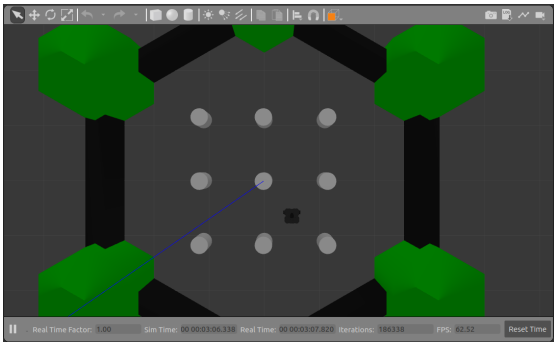


Figure 7.3: Gazebo SLAM environment

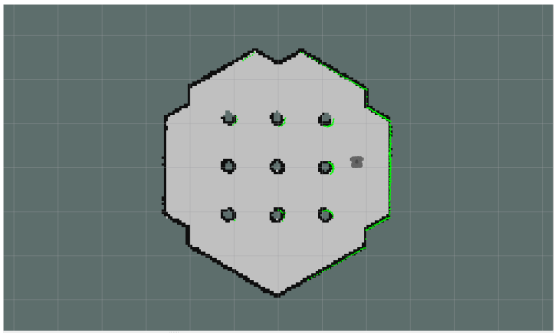


Figure 7.4: SLAM in RViz

The SLAM is node runs the SLAM algorithm in Turtle Bot 3. This is the main node that uses the LIDAR data to create a cost map of the surroundings.

The SLAM algorithm can be launched by running the given command on the host PC.

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_slam turtlebot3_slam.launch
```

The SLAM node uses an algorithm that combines information from different sensor modalities to create a more accurate and reliable map of the environment. This algorithm can be either a probabilistic method such as Extended Kalman Filter (EKF) or a graph-based method such as GraphSLAM. [6]

The SLAM node in Turtle Bot 3 performs the following steps:

- **Data Acquisition**: The node collects data from the robot's sensors such as LIDAR, RGB-D camera, and IMU.
- **Feature Extraction**: The SLAM algorithm extracts features from the sensor data such as edges and corners, which are used to identify different objects in the environment.
- **Data Association**: The SLAM algorithm associates the features extracted from the sensor data with their corresponding objects in the environment.
- **Map Building**: Based on the associated features, the algorithm constructs a map of the environment. The map can be represented in different forms such as occupancy grids, point clouds, or 3D models.

- **Localization**: The SLAM algorithm estimates the robot's position within the environment by comparing the sensor data with the map.

7.2.3 Tuning the SLAM parameters

There are several parameters in G Mapping SLAM in Turtle Bot 3 which can be tuned to enhance performance in different environments. These parameters are present in the *turtlebot3_slam/config/gmapping_params.yaml* file. The following parameters can be altered to best fit the needs of the surroundings to run SLAM the most efficient way. [6]

- **maxUrange**: The maxUrange parameter in SLAM Turtle Bot 3 sets the maximum range (in meters) that the LIDAR sensor can detect. Any obstacles beyond this range will not be considered by the SLAM algorithm during map creation and localization.

- **map_update_interval**: The map_update_interval parameter in SLAM Turtle Bot 3 is used to set the time interval (in seconds) for updating the map. The algorithm only updates the map if the robot has moved a certain distance or angle since the last update, as specified by the **update_min_a** and **update_min_d** parameters.

- **minimumScore**: minimumScore is a parameter in Turtlebot 3 SLAM that defines the minimum likelihood score required for a scan match to be considered valid. This helps improve map accuracy by filtering out unreliable data.

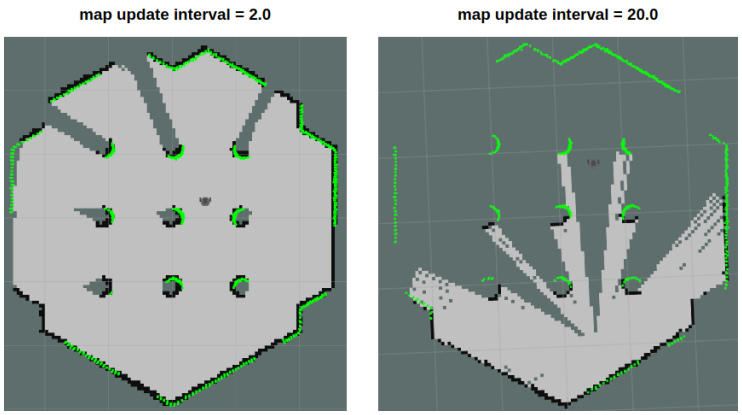


Figure 7.5: Altering the **map_update_interval** parameter

- **linearUpdate**: linearUpdate is a parameter in Turtlebot 3 SLAM that controls the frequency of updates to the map based on robot movement. Higher values increase the update rate, improving map accuracy but also increasing computation time. Lower values decrease update rate but decrease accuracy.

- **angularUpdate**: angularUpdate in Turtlebot 3 SLAM refers to the process of updating the robot's orientation information based on sensor data. This information is used to create a map of the robot's environment.

7.2.4 Saving the Map

After running the SLAM node, the costmap generated by results in an Occupancy Grid Map. In order to run Navigation using that map, the map needs to be saved.

The map can be saved by running the following command:

```
$ rosrun map_server map_saver -f ~/map
```

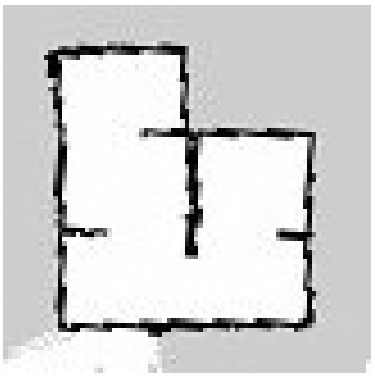


Figure 7.6: Map to be saved

The above command saves the *map.pgm* and *map.yaml* files in the home folder */home/*${*username*}.

Chapter 8

Path Planning

8.1 Single Source Shortest Path Algorithms

Single source shortest path algorithms are widely used in path planning for robotics. Some of the commonly used algorithms are Dijkstra's algorithm, Bellman-Ford algorithm, and A* algorithm. Dijkstra's algorithm is an efficient method for finding the shortest path in a graph with non-negative edge weights, while the Bellman-Ford algorithm can handle graphs with negative edge weights.

The A* algorithm is a heuristic search algorithm that uses an estimated cost-to-go function to guide the search towards the goal. These algorithms have been successfully applied in various applications such as autonomous navigation, obstacle avoidance, and task planning for mobile robots.

In this project, we use the A* algorithm to determine the shortest path from a given start position to a target position. The A* algorithm is applied on the 2-D matrix of the previously saved map image hence achieving autonomous navigation.

Let us discuss the A* algorithm and it's working in detail.

8.1.1 A* Algorithm

The A* algorithm is a popular path planning algorithm used in robotics and other applications. It is a heuristic search algorithm that uses a cost-to-go estimate to guide the search towards the goal.

The A* algorithm uses heuristics to find the shortest path between two points in a graph or a grid. The algorithm works by maintaining a priority queue of nodes to be expanded, with the node with the lowest expected total cost being expanded next.

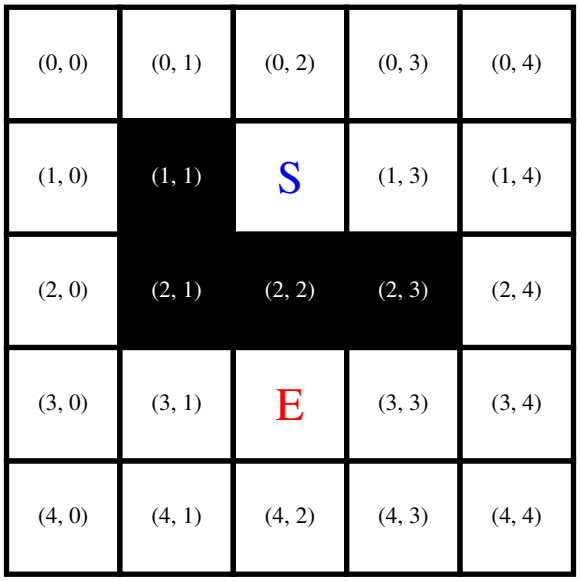
The expected cost of a node is the sum of the actual cost to reach that node from the starting node and the estimated cost to reach the goal node from that node. The estimated cost is calculated using a heuristic function, which provides an estimate of the remaining distance between the current node and the goal node.

At each step, the algorithm expands the node with the lowest expected total cost, generating its neighboring nodes and updating their expected costs if necessary. The algorithm terminates when the goal node is reached, or when there are no more nodes to expand.

The Matrix generator algorithm used in this project takes the map image and converts it into the 2-D matrix of 0s and 1s. Let us understand the working of the A-Star algorithm on the Map matrix.

Let us consider a 5 x 5 matrix similar to the matrix generated by the Matrix generator. Here, the solid checks represent

the blocked area (similar to 0s in the Map matrix) and the hollow checks represent the movable area (similar to 1s in the Map matrix). The symbol S represents the Start position and the symbol E represents the End position.



The A-Star Algorithm functions by selecting the next node from the current node on the basis of a parameter known as the f-cost. The f-cost is the sum of 2 other parameters known as the G-cost and the H-cost.

The g-cost of a node is the movement cost to move from the starting point to that particular node whereas the h-cost is the movement cost from the current node to the end node. These values are calculated mostly based on the Eucledian distance between the two respective nodes.

Let us implement the above logic in the given matrix to find out the shortest path from the Start to the End position.

The Start is S(1, 2) and the End is E(3, 2).

The A-Star algorithm has the following steps to find the shortest path:

- **Initialise the Open list:** The Open list (also known as the priority queue or fringe) is a data structure that stores the nodes that have been discovered but have not yet been fully evaluated. We add the Start node to the Open list and make it's F-cost 0.

- **Initialise the Closed list:** The Closed list is a data structure that stores the nodes that have been evaluated during the search.

  while (Open list is not empty){

- Determine the node with the least F-value from the Open list and name it 'n'

- Pop n off the Open list

- Generate the 8 neighbours of n and set their parents to n

  for each neighbour{

- **if** neighbour is the goal node, **then** stop search

- **else** calculate the G-cost and the H-cost of the neighbour and add them to get the F-cost.

- **if** a node with the same position but lower F-cost than the neighbour is in the Open list, **then** skip this neighbour.

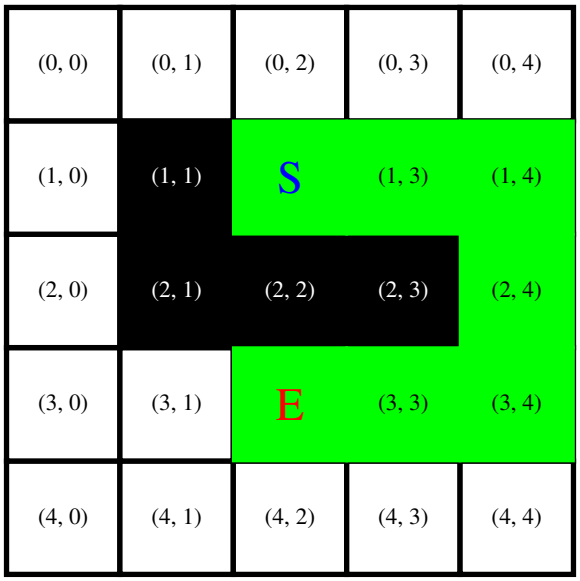
- **if** a node with the same position as the neighbour is in the Closed list which has a lower F-cost than the neighbour, **then** skip this neighbour, **else** add this node to the Open list.

} //end For loop

- push n on the Closed list

} //end While loop

On applying the above steps of the A-Star algorithm, the Single Source Shortest path we obtain is:



Path: S(1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (3, 3), E(3, 2).

8.2 Map matrix

The Map matrix refers to the 2-dimensional matrix of 0s and 1s. The map generated by the SLAM algorithm is a *.png* image where the white area is the movable region, the black area are the obstacles and the grey area is the unknown region. To make this map understandable to the system and to implement path planning on this map, we need to convert this map into a 2-dimensional matrix.

The map matrix that we generate by the Matrix Generator code (refer Appendix - 9.1) generates a 2-D matrix of 0s and 1s. The 0s in the matrix represent the movable white region of the map image and the 1s represent the obstacles.

8.3 Python OpenCV

Python OpenCV is a powerful computer vision library that provides a wide range of tools and algorithms for image and video processing. It is an open-source library that allows developers to perform various tasks such as image and video capture, filtering, feature detection, object recognition, and more.

With its easy-to-use Python API and extensive documentation, OpenCV has become one of the most popular libraries for computer vision applications. Its versatility and robustness make it suitable for a wide range of industries, including robotics, automotive, healthcare, security, and more.

We use Python OpenCV PIL library in this project for Map image processing to read the data from the Map and turn it into digital numpy matrix. OpenCV helps us to set the grayscale

threshold for classifying the different pixels of the map image to cast it into a 2-D matrix (128 in this case). All the pixels with intensity value above 128 are pushed to 255 and the remaining pixels are pulled to 0 therefore generating a 2-D matrix of 0s and 1s.

The *binary.resize()* function of the PIL library helps us to alter the dimensions of the 2-D matrix according to the size of the real world map. This function is crucial because we often need to resize the map size according to the actual world so as to get a feasible unit cell size.

By nested looping through the numpy array, we can print the 2-D matrix for reference purposes as well. The *np.savetxt* function allows us to store the numpy array in a text file with *.txt* extension for reference.

8.4 A-Star

The A-Star algorithm is the program that takes the map created by the Map generator code and applies the A-Star algorithm on it to determine the Single Source Shortest Path between the given Start and the End nodes.

The first line of the code

```
#!/usr/bin/env python3
```

determines that the code is to be run as a python script when executed in the terminal. This is crucial as the script needs to interact with the *roscore* for which it needs to be run as a python script when invoked from the terminal.

The program uses a form of queue data structure known as the heap queue available as a python library by the name of *heapq*. The code takes the *X* and the *Y* coordinates of the Start and the End nodes and the radius of the robot. The variable *ROBOT_RADIUS* takes the radius of the robot as an input which is nothing but a number of cells that when combined represent one side width of the robot. When the path planning algorithm is applied on the matrix, it considers the robot as a point mass instead of a whole body due to which if not provided with a threshold, it generates the path adjacent to the obstacles. Therefore when actually executed, the sides of the robot collide with the obstacles. The *ROBOT_RADIUS* variable is the number of cells that represents half the horizontal size of the robot. The A-Star algorithm checks all the eligible nodes if they are clear from the obstacles by this magnitude so as to keep the robot clear from the obstacles.

The A-Star algorithm considers the neighbouring nodes of the current node in the Open list. The neighbouring nodes are the 8 adjacent nodes of the current node. The 8 adjacent nodes of the current node can be determined by adding and subtracting 1 from both the *X* and the *Y* coordinates of the current node systematically.

For example,

Let the current node be *(5, 5)*
The 8 neighbouring nodes will be:

- $(5-1,5) = (4,5)$
- $(5,5-1) = (5,4)$
- $(5+1,5) = (6,5)$
- $(5,5+1) = (5,6)$

- $(5+1, 5+1) = (6, 6)$
- $(5-1, 5-1) = (4, 4)$
- $(5-1, 5+1) = (4, 6)$
- $(5+1, 5-1) = (6, 4)$

The algorithm determines the node with the least F-cost and pushes it into the heap queue. When the list is ready, it is traversed in the reversed order, which then reveals the Shortest path between the given Start and the End nodes.

The nested loop at the end of the program is used to print the map matrix of 0s and 1s with the path shown as '0's. The variable can be changed as per the user's choice as it is only a method of representing the generated path.

This path is then fed to the Autonomous Navigation algorithm which determines the directions and the individual wheel velocities to navigate the robot to the desired coordinates.

Chapter 9

Autonomous Navigation

9.1 Reading the path

The path generated by the path planning algorithm (refer Appendix - 9.2) is in the form of a list of tuples. The tuples have the *X* and the *Y* coordinates of the nodes in the list. The Autonomous Navigation algorithm determines the direction of motion and orientation of the robot by establishing a relationship between 2 consecutive nodes in the path list.

Let us understand the working of the Autonomous Navigation algorithm in detail.

9.1.1 Setting offsets

The map matrix starts from $(0, 0)$ at the top left corner. However, that area is usually outside the movable region of the map as it corresponds to the gray area of the map image which means it is unknown. Therefore, the robot needs to be placed at the Start coordinates manually or we can say that the start coordinates of the robot need to be manually defined.

In order for the Autonomous Navigation algorithm to suc-

cessfully navigate the robot along the path, we need to change the current coordinates of the robot in such a way that they coordinate with the nodes present in the path. One way of achieving that is the Offset reduction method. In this method, the starting coordinates of the robot are considered as the Offset variables and they are constantly subtracted from the path nodes coordinates.

Let us say that the Start coordinates of the robot are $(30,40)$. The *x_offset* will be 30 and the *y_offset* will be 40. The path will start from the current position of the robot *i.e.* $(30,40)$ and will end at the end coordinates (say $(70,70)$). Therefore, the effective path that will be traversed by the robot will be

$$\Rightarrow (30-30,40-40),(31-30,41-40),(32-30,42-40).....$$

$$\Rightarrow (0,0),(1,1),(2,2).....$$

Note: The offsets once determined need to be kept constant for the complete session of Navigation within that environment unless the robot loses connectivity or in case of wheel drag or when restarting the session *i.e.* when re-calibrating the robot at the Start position.

9.1.2 Solving Turning

Due to lack of Active Intelligence in the robots, we face the problem of Intelligent Turning. This refers to the issue that when the robot is facing in a certain direction and is required to turn towards a certain heading, it always turns in one direction (clockwise or counter-clockwise) unless provided with a certain intelligent angular distance measurement system. We solved this problem using a little Trigonometry.

The problem is that when provided with a certain heading, the robot turns towards that heading only in either clockwise or counter-clockwise direction. However, we need the robot to determine the minimum angular distance between its current and desired heading and thereby determine the appropriate direction of turning *i.e.* clockwise or counter-clockwise.

The Odometry transform that we use in this program creates a 2-D Cartesian plane of reference around the robot. Suppose the robot is placed horizontally on the ground in the *X*-*Y* plane, the point where the robot is calibrated defines the Origin of the Cartesian plane. At the origin, the *X* and the *Y* coordinates read $(0,0)$ whereas the *Z* axis projects vertically upwards from the ground, therefore making the origin coordinates as $(0,0,0)$. The direction in which the robot is facing is the positive *X* direction and turning 90° to the left of positive *X* direction, we get the positive *Y* direction.

The robot Odometry reads the orientation in the form of Quaternions which is a way to define the orientation of an object in 3-D space. The *tf.transformations* package provides us with the *euler_from_quaternion* library which converts the given Quaternions to Euler angles. We use this package to convert the Quaternion data from the Odometry topic into Euler angles to determine the orientation of the robot in Radians in 3-D space.

The *angle_to_goal* variable is the distance in radians between the current and the desired heading in radians. It uses the basic formula of coordinate geometry for slope of a line. An imaginary line is considered between the current position and the goal position and the slope of that line determines the *angle_to_goal*.

With the use of a little Trigonometry, it determines the minimum angular distance between its current orientation and the desired orientation.

```
delta_heading = math.atan2(
                    math.sin(angle_to_goal - yaw),
    math.cos(angle_to_goal - yaw))
```

The *delta_heading* variable uses the *math.atan2* function to determine the minimum angular distance between the current and the desired orientation of the robot, therefore allowing the robot to implement Intelligent Turning as discussed earlier.

9.2 Proportional Controller

9.2.1 What is a Proportional Controller

A proportional controller is a type of feedback control system that generates an output signal proportional to the difference between the desired setpoint and the measured process variable. The output signal is typically used to adjust the process variable in order to bring it closer to the desired setpoint.

We use a Proportional Controller in this project to control the linear and the angular velocities of the robot. The use of a Proportional Controller helps us to achieve a smooth movement of the robot along the planned path.

9.2.2 K_P Linear and K_P Angular

The K_P Linear or the *p_controller_linear* is the variable that is used to regulate the Linear velocity and the K_p Angular or the *p_controller_angular* is the variable that is used to regulate the Angular velocity of the robot. Let us understand the working

of these parameters.

The controller uses the size of a unit cell as a standard method of distance measurement. For example; let us consider the unit cell size to be 5 centimeters (one side of the cell). The Linear controller considers the current position of the robot and implements the proportional controller until the robot moves by the unit cell distance in the desired direction. Therefore, if the current cell is at $(0,0)$, then the proportional controller regulates the linear velocity until its current position becomes $(0.05,0)$ in meters (assuming the desired direction of motion is in the positive *X* direction.)

Similarly, the Angular Proportional controller uses the *angle_to_goal* variable to regulate the angular velocity of the robot. For example; if the current heading of the robot is 0 radians and the desired heading is $\frac{\pi}{2}$ radians, then the angular proportional controller will regulate the angular velocity of the robot until its Euler angle value reads $\frac{\pi}{2}$ radians.

The proportional controller regulates the velocities in such a way that the velocity is provided with an upper limit and a lower limit. When the robot is the farthest from the desired position, the velocity value (linear or angular) hits the maximum value and keeps dropping until the robot reaches the desired position where the velocity hits the minimum value. However, the minimum limit of the velocity is never set to 0 as it would then trigger an infinite feedback controlled loop as the robot will never reach the target position.

The velocity magnitude is regulated with the help of a constant value which is known as the P value in the Proportional control system. This value is multiplied by the velocity values to get the magnified velocities in order to achieve a feed-

back controlled loop. In case of a PID controller, where there are 3 control variables, the Proportional, the Integral, and the Derivative, all the 3 constant values are provided which then regulate the feedback controlled loop.

9.2.3 Deducing directions

The process of deducing directions from the list of tuples is quite simple. The path that is generated by the Path Planning algorithm is provided as a list of tuples where the tuples are the X and the Y coordinates of the nodes of the matrix.

The Autonomous Navigation algorithm takes the first tuple of the list to be the Start node or the current node. From the first node, further directions are deduced by the Autonomous Navigation algorithm. Directions are not deduced through a completely separate process, instead the Proportional controller itself helps us to traverse through the path.

Taking the first node as the current node, the Angular Proportional controller calculates the desired heading of the robot and while the angular velocity is getting regulated, the Linear Proportional controller calculates the Eucledian distance between the current and the next node. As the Angular Proportional controller approaches the desired heading, the Linear controller also tends to approach the desired node position. Therefore, the Autonomous Navigation is a continuous and smooth process where with the help of Intelligent Turning and the use of Proportional controllers, we are able to achieve a smooth and jerk free movement of the robot along the planned path.

9.2.4 Publishing velocities

The velocities in ROS (Robot Operating System) are measured in terms of meters per second ($ms^{-}1$). These velocities are published as a ROS Message on the */cmd_vel* topic.The ROS Message used to publish the velocities on the */cmd_vel* topic is the *Twist* message from the *geometry_msgs.msg* package.

The *Twist* message has 2 parts - the Linear and the Angular velocities that are described as velocities in the 3 axes - *X*, *Y* and *Z*. The structure of the *Twist* message is as follows:

Twist,
linear,
x:,
y:,
z:,
angular,
x:,
y:,
z:,

When publishing the velocities on the */cmd_vel* topic, the *Twist* message is referred to through an object of the *Twist* class. The individual components of the *Twist* object *i.e.* the *X*, *Y* and the *Z* components of the Linear and the Angular velocities are referred to in the following way.

Let us consider *obj* to be the object of the *Twist* class. Then the individual components of the object will be addressed as:

Linear velocity:
obj.linear.x = 0;

Angular velocity:
obj.angular.z = 0;

A conditional while loop is used to continuously check the current position of the robot is the Cartesian plane. Until the robot reaches its desired position and orientation in the plane, the calculated Linear and the Angular velocities are continuously published on the */cmd_vel* topic.

The SBC (Single Board Computer) *i.e.* the Raspberry Pi implements the ROS framework and receives these velocities over the */cmd_vel* topic. The SBC in turn calculates the appropriate individual wheel velocities and feeds the required commands to the Open CR 1.0 controller that controls the Dynamixel motors' speed by altering the voltage supply.

The Open CR 1.0 also reads the encoder ticks from the rotary encoders in the Dynamixel to the SBC which in turn publishes the current position and orientation of the robot on the */odom* topic.

Chapter 10

Conclusion

10.1 Result

The problem statement of this project revolved around the objective *i.e.* Autonomous Navigation of Differential Drive Robots using only Wheel Odometry as the source of localisation without the use of any other sensors like Cameras or Lasers.

With the correct implementation of the algorithms in this project, we were able to successfully navigate a Differential Drive Robot (Turtle Bot 3) from a Starting to an End point in a Static environment using data from the wheel encoders. We were able to successfully create a Map of the surroundings using the LIDAR mounted on the Turtle Bot 3 and implementing the SLAM algorithm (as mentioned in Chapter 7 - Simultaneous Localization And Mapping.)

Further, the *map_maker.py* script enabled us to convert the Map image into a 2-D matrix representing the movable and the blocked area (as mentioned in Chapter 8 - Path Planning). We apply the path planning algorithm **A-Star** for determining the path between a Start and End position avoiding the various obstacles and computing the shortest path.

Lastly the *auto_nav.py* algorithm is executed (as mentioned in Chapter 9 - Autonomous Navigation) to convert the nodes from the path generated into velocity commands and publish them on the */cmd_vel* topic in ROS which in turn converts these values into individual wheel velocities for the two drive wheels.

In this project, we were successfully able to deploy a fully functional proportional controller for controlling the wheel velocities to optimise the Autonomous Navigation process resulting in a fairly smooth movement of the robot (as mentioned in Chapter 9 - Autonomous Navigation). The successful completion of this project demonstrates the potential for using Wheel Odometry in Autonomous Navigation systems, and opens up new possibilities for the development of more efficient and cost-effective robotic systems.

The most important result of this project is that by developing this project, we have simplified the process of teaching, demonstrating, developing and understanding this concept to great limits. The use of Python programming language with proper documentation on the GitHub repository makes understanding, implementing and further developing the project a lot easier than it would appear to a novice. The complete process of setting up, demonstrating and clearing any queries is mentioned in the GitHub repository - Autonomous Navigation.

10.2 Issues

Though the project - Autonomous Navigation using Wheel Odometry is completed successfully, there are a few issues that we need to address so that they can be solved should

someone develop this project further. Following are a few major issues that were faced during the development of this project:

- **Map scaling:** The *map_maker.py* script takes the Map image and converts it into a 2-D matrix of 0s and 1s (as mentioned in Chapter 8 - Path Planning). However, the script does not do anything to generalise the dimensions of the matrix for all cases, instead the developer has to manually adjust the dimensions of the matrix according to the size of the Map to get a clear resolution. If the dimensions are generalised for all cases, it would make the process more user friendly and easier to automate for more complex tasks.

- **Path Planning:** The path planning algorithm that we use *i.e.* **A-Star** implements path planning on a 2-D matrix considering the Robot to be a group of cells. The Robot's radius is taken as an input (as mentioned in Chapter 9 - Autonomous Navigation), and the algorithm adds a margin value of 1 or 2 cells to the Robot's radius to keep it clear off the walls to prevent collision. However, in this approach in some test cases it was observed that the algorithm failed to generate a complete path between the Start and the End positions. To overcome this problem, a solution can be to add a margin thickness to the walls in the Map and therefore in the Matrix and consider the robot as a point mass while planning the path.

- **Real time obstacle avoidance:** Though the Robot is able to navigate autonomously from the Start to the End position, it is not able to avoid obstacles in real time *i.e.* Autonomous Navigation in a Dynamic environment. Future

developers of this project can work on this issue to make the robot a real time obstacle avoiding robot by using either the mounted LIDAR or other distance or visual sensors like the Ultra Sonic distance sensors or RGB-Depth cameras.

- **Use of PID:** This project deploys a proportional controller to control the speeds of the wheels of the Robot in order to achieve a smooth movement. However, the Integral and the Derivative controllers can be integrated in the Autonomous Navigation algorithms to make the movement even smoother and it's integration with other devices like a Manipulator etc.

The future developers of this project can work on the above mentioned issues to make the algorithms even better and user-friendly therefore making it easier to deploy this robot in coordination with other robots to perform multiple tasks simultaneously.

Bibliography

The following Bibliography provides a comprehensive list of resources that were referred to or used directly or indirectly in any way. Any kind of knowledge or resource that is used for the development of this project that is not the property of the Author is duly cited in the following Bibliography and can be accessed by the readers of this report for future reference.

Concepts

- **Differential Wheel Drive**: United States Naval Academy, [Mobile Robot Kinematics]. [1]
- **Robot Navigation**: Research Gate, [Navigation of Autonomous Mobile Robots]. [2]
- **Odometry**: medium.com - [Wheel Odometry Model for Differential Drive Robotics]. [3]
- **ROS (Robot Operating System)**: ros.org - [Documentation, Tutorials, Papers]. [4]
- **Turtle Bot 3**: robotis.us - [Turtle Bot 3 e-Manual]. [5]
- **SLAM**: theconstructsim.com - [ROS Development Studio, SLAM with Turtle Bot 3]. [6]
- **Path Planning**: ieeexplore.ieee.org - [Path Planning Using an Improved A-star Algorithm]. [7]

- **A-Star algorithm**: cs.cmu.edu - [Carnegie Mellon University - Robotic Motion Planning: A* and D* Search]. [8]

Figures

- **Fig 4.1**: healthline.com
- **Fig 6.2**: robotis.us - [Turtle Bot 3]
- **Fig 6.3**: in.element14.com - [Raspberry Pi 4-B]
- **Fig 7.5**: robotis.us - [Turtle Bot 3 e-Manual]

All the figures in this report other than the ones mentioned in the above list are taken from the **Integral Robotics Lab, Integral University, Lucknow**.

Appendix

All the code used in the project is given in the following subsections.

Matrix generator

The following code takes in the Map image generated by the SLAM algorithm using LIDAR and generates a 2 dimensional matrix of 0s and 1s. The 0s symbolise the obstacles in the map and 1s symbolise the free movable area of the map.

```
from PIL import Image
import numpy as np

class map():

    # Open the maze image and make greyscale, and get its           dimensions

    im = Image.open('<address of the map image>').convert('L')

    # Ensure all black pixels are 0 and all white pixels are 1
    binary = im.point(lambda p: p > 128 and 1)

    # Resize to half its height and width so we can fit on Stack Overflow, get new dimensions

    binary = binary.resize((70, 70)) #dimensions can be altered as the user sees fit

    w, h = binary.size

    # Convert to Numpy array - because that's how images are best stored and processed in Python
    nim = np.array(binary)
```

```
21
22      for r in range(h):
23          for c in range(w):
24              nim[r, c] = int(nim[r, c])
25
26      # np.savetxt("map.txt", nim)
27
28      # for r in range(h):
29          # for c in range(w):
30              # print(nim[r, c],end='')
31          # print()
```

A-Star algorithm

The following code takes the 2 dimensional matrix generated by the Map generator script and applies the A-Star algorithm on it to determine the path from a given starting position to a given end position. The path generated by the script is given as a list of tuples. The tuples have 2 arguments, the x and the y co-ordinates.

```
1
2  #!/usr/bin/env python3
3
4  from map_maker import map
5  import heapq
6
7  # Function to determine the validity of the cell
8
9  def walkable(grid, x, y, robot_radius):
10     for i in range(-robot_radius, robot_radius+1):
11         for j in range(-robot_radius, robot_radius+1):
12             if not (0 <= x + i < len(grid) and 0 <= y
       + j < len(grid[0])):
13                 return False
14             if grid[x + i][y + j] == 0:
15                 return False
16     return True
17
18 # Function to find the path
19
20 def find_path(grid, start, goal, robot_radius):
21     # Create a priority queue to store potential path
       squares
22     heap = [(0, start)]
23     # Create a dictionary to store the cost of each
       square
24     cost = {start: 0}
25     # Create a dictionary to store the parent of each
       square
```

```
    parent = {start: None}
    # Create a set to store visited squares
    visited = set()

    while heap:
        # Pop the square with the lowest cost from the heap
        current = heapq.heappop(heap)[1]

        # If the current square is the goal square, return the path
        if current == goal:
            path = []
            while current != start:
                path.append(current)
                current = parent[current]
            path.append(start)
            return path[::-1]

        # Mark the current square as visited
        visited.add(current)

        # Check the squares adjacent to the current square
        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1), (1, -1), (-1, 1), (-1, -1), (1, 1)]:
            x, y = current
            next_square = (x + dx, y + dy)
            # If the next square is outside the grid or is an obstacle, skip it
            if not walkable(grid, x + dx, y + dy, robot_radius):
                continue
            # If the next square has been visited, skip it
            if next_square in visited:
                continue
            # Calculate the cost of the next square
            next_cost = cost[current] + 1
            # If the next square is not in the heap or has a higher cost, update it
            if next_square not in cost or next_cost < cost[next_square]:
                cost[next_square] = next_cost
                priority = next_cost + (abs(goal[0] - x - dx) + abs(goal[1] - y - dy))
                heapq.heappush(heap, (priority, next_square))
                parent[next_square] = current

    # If no path was found, return None
    return None


grid = map.nim

```

```
# Start and goal coordinates and Robot radius for
    obstacle clearance

print("Enter Start Co-ordinates: ")
st_x = int(input("Enter X: "))
st_y = int(input("Enter Y: "))
start = (st_x, st_y)

print("Enter End Co-ordinates: ")
en_x = int(input("Enter X: "))
en_y = int(input("Enter Y: "))
goal = (en_x, en_y)

ROBOT_RADIUS = int(input("Robot Radius: "))

if __name__ == "__main__":
    path = find_path(grid, start, goal, ROBOT_RADIUS)
    print(path)
    planned_grid = grid

    for point in path:
        planned_grid[point[0], point[1]] = "0"

    for i in range (0, 70):
        for j in range (0, 70):
            print(planned_grid[i, j], end = "")
```

Autonomous Navigation

The following code is the main driver code for the robot. It takes the path generated by the path generator script and deduces the direction in which it needs to move. It also takes the cell size of the matrix as compared to the real world map so as to calculate the distance between 2 cells.

```

#!/usr/bin/env python3

import math
import rospy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion

current_x = current_y = current_yaw = None
c = 0


def callback_method(msg):
    global current_x, current_y, current_yaw
    current_x = msg.pose.pose.position.x
```

```
17
18      current_y = msg.pose.pose.position.y
19      quaternion = (msg.pose.pose.orientation.x, msg.
        pose.pose.orientation.y, msg.pose.pose.orientation
        .z,
20                    msg.pose.pose.orientation.w)
21      value = euler_from_quaternion(quaternion)
22      current_yaw = value[2]
23
24
25  def mover(goal_x, goal_y):
26      global current_x, current_y, c
27
28      p_controller_linear = 0.8
29      # p_controller_angular = 0.8
30      rate = rospy.Rate(40)
31      speed = Twist()
32
33      while not rospy.is_shutdown():
34          if current_x and current_y and current_yaw is
        not None:
35              while not rospy.is_shutdown():
36
37                  dist = abs(math.sqrt(((goal_x -
        current_x) ** 2) +
38                                       ((goal_y - current_y) ** 2)
        ))
39
40                  if (dist < 0.02):
41                      c = c + 1
42                      break
43
44                  linear_speed = dist *
        p_controller_linear
45
46                  # set upper limit of linear velocity
47                  linear_speed = min(linear_speed, 0.1)
48                  # set lower limit of linear velocity
49                  linear_speed = max(linear_speed, 0.06)
50
51                  angle_to_goal = math.atan2(
52                      goal_y - current_y, goal_x -
        current_x)
53
54                  if (current_yaw < 0):
55                      yaw = 6.28 - abs(current_yaw)
56
57                  else:
58                      yaw = current_yaw
59
60                  if (angle_to_goal < 0):
61                      angle_to_goal = 6.28 - abs(
        angle_to_goal)
62
63                  delta_heading = math.atan2(
64                      math.sin(angle_to_goal - yaw),
        math.cos(angle_to_goal - yaw))
```

```
65
66
67                  if abs(angle_to_goal - yaw) > 0.02:
68
69                      # set upper limit of angular
    velocity
70                      angular_speed = min(delta_heading,
     0.15)
71                      # set lower limit of angular
    velocity
72                      angular_speed = max(delta_heading,
     -0.15)
73
74                  else:
75                      angular_speed = 0.0
76
77                  if abs(angular_speed) > 0.05:
78                      linear_speed = 0.0
79
80                  speed.linear.x = linear_speed
81                  speed.angular.z = angular_speed
82
83                  pub.publish(speed)
84                  rate.sleep()
85
86              break
87
88      speed.linear.x = 0.0
89      speed.angular.z = 0.0
90
91      if c > 0:
92          pub.publish(speed)
93          print("Done")
94          c = 0
95
96      else:
97          rospy.signal_shutdown("kill")
98          print("Keyboard interrupt!")
99
100
101 if __name__ == "__main__":
102
103     rospy.init_node("mover")
104     pub = rospy.Publisher('/cmd_vel', Twist,
    queue_size=10)
105     rospy.Subscriber('/odom', Odometry,
    callback_method)
106
107     cell_size = float(input("Enter the cell size: "))
108
109
110     # path = <path generated by the path planner
    script goes here>
111
112     x_offset = int(input("Enter X offset: "))
113     y_offset = int(input("Enter Y offset: "))
114
```

```
    print("X-offset =", x_offset)
    print("Y-offset =", y_offset)

    for node in path:
        x_goal = (node[0] - x_offset) * cell_size
        y_goal = (node[1] - y_offset) * cell_size
        print("Moving to", x_goal, y_goal)
        mover(x_goal, y_goal)

    print("Reached.")
```