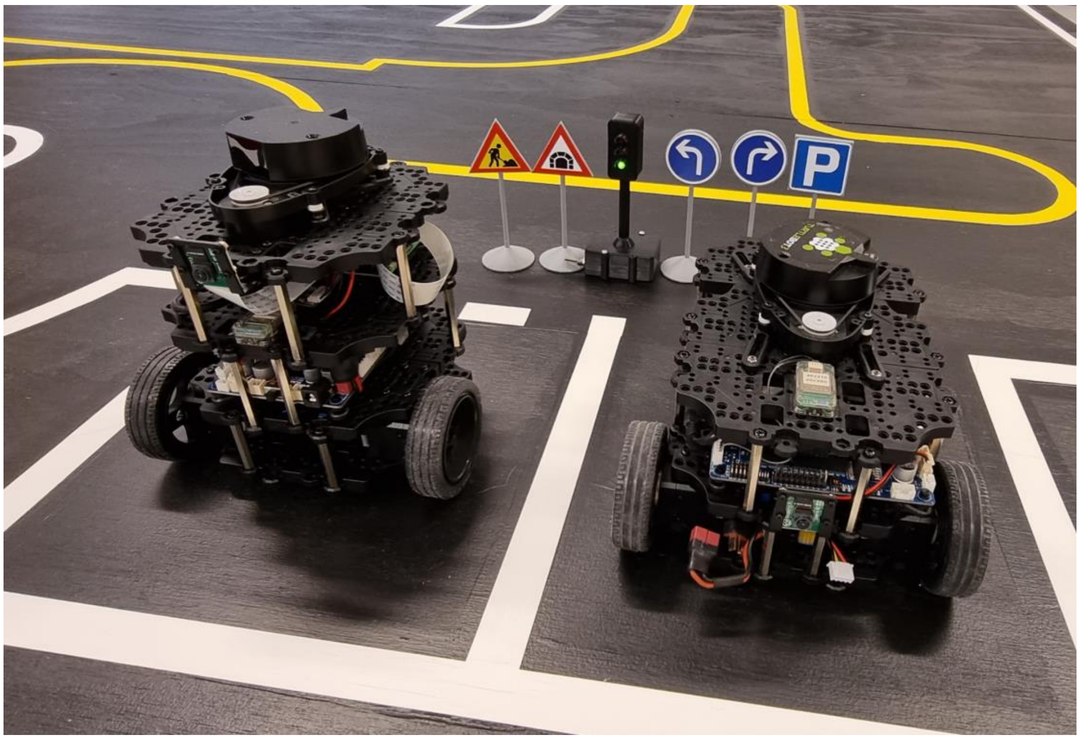
University of South-Eastern Norway

www.usn.no

FMH606 Master's Thesis 2023

Industrial IT and Automation (IIA)

Autonomous driving and machine learning with TurtleBot3 Waffle Pi mobile rover



Sindre Haugseter

Faculty of Technology, Natural sciences and Maritime Sciences
Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis 2023

Title: Autonomous driving and machine learning with TurtleBot3 Waffle Pi mobile rover

Number of pages: 63

Keywords: TurtleBot3, Autorace, SLAM, autonomous driving, ROS, obstacle avoidance, machine learning, reinforcement learning

Student:	Sindre Haugseter
Supervisor:	Associate Professor Roshan Sharma (USN) Senior Engineer Fredrik Hansen (USN)

Summary:

This project explores autonomous driving and machine learning with the TurtleBot3 Waffle Pi mobile rover in a custom-built racing rig. The project aims to develop an autonomous driving vehicle research and educational platform where the TurtleBot3 navigates the racing rig while clearing obstacles such as traffic lights, intersections, parking, construction work, and tunnels. Various methods are employed to achieve the objective, including camera calibration, image processing, feature detection, SLAM-based navigation, and reinforcement learning. The robot's performance is assessed in both simulated and real-world environments.

In the simulated environment, the robot successfully completes the course but encounters issues such as cutting corners and occasional difficulties generating a path through the tunnel. In the real environment, the robot can detect and follow the lane and clear obstacles, traffic lights, and tunnels but struggles with traffic sign detection, affecting its ability to start different missions.

The paper provides insights into the capabilities and limitations of the TurtleBot3 for autonomous navigation in complex environments and identifies areas for further development and optimization. This research lays the foundation for future work in autonomous driving and mobile robotics.

Preface

This Master's Thesis, titled "Autonomous driving and machine learning with TurtleBot3 Waffle Pi mobile rover," documents the work conducted at the University of South-Eastern Norway (USN) during the spring term of 2023. The project is part of the Industrial IT and Automation (IIA) program, and its findings and outcomes are presented in this report.

The target audience for this thesis is people with an interest or background in robotics, autonomous systems, and machine learning. A basic grasp of autonomous navigation, sensor technology, reinforcement learning, and computer vision is beneficial for a comprehensive understanding of this project.

I would like to express my gratitude to Associate Professor Roshan Sharma and Senior Engineer Fredrik Hansen for their guidance and support throughout the project.

Porsgrunn 15.05.2023

Sindre Haugseter

Contents

Nomenclature

- AMCL – Adaptive Monte Carlo Localization
- DQN – Deep Q-Network
- DWA – Dynamic Window Approach
- EKF – Extended Kalman Filters
- LiDAR – Laser imaging, Detection, and Ranging
- PID – Proportional-Integral-Derivative
- RBPF – Rao-Blackwellized Particle Filter
- RL – Reinforcement Learning
- ROS – Robot Operating System
- SBC – Single Board Computer
- SIFT – Scale-Invariant Feature Transform
- SLAM – Simultaneous Localization and Mapping

1 Introduction

Autonomous driving and mobile robots have experienced significant advancements in recent years. The TurtleBot3 Waffle Pi, a mobile rover using the Robot Operating System (ROS), has gained popularity in research and education settings. This project builds upon a previous research effort and aims to develop an autonomous driving research and educational platform using the TurtleBot3 Waffle Pi at the University of South Norway (USN).

1.1 Background

The TurtleBot3 Waffle Pi is a ROS-powered mobile rover widely used in robotics research and education. It features two DYNAMIXEL smart servo motors for mobility, a Raspberry Pi camera for image and video, and a 360-degree LIDAR sensor for 2D scanning and local map generation through algorithms like Simultaneous Localization and Mapping (SLAM). USN has recently acquired several TurtleBot3 Waffle Pi units for teaching and research activities. This project is a continuation of work initiated in a previous course called "Project" in collaboration with Christian Lauritzen, as documented in the report "Setup, configuration and simulation of Turtlebot3 Waffle Pi mobile rover for autonomous driving. (Haugseter & Lauritzen, 2022)" The first custom TurtleBot3 was built during that project, and the necessary codes and packages were installed to perform SLAM and Navigation. The racing rig's design and construction were also started, with the track being the primary focus.

The current project aims to complete the racing rig by adding traffic signs, lights, obstacles, and a tunnel. The main goal is to enable the TurtleBot3 to navigate the racing rig autonomously, following established autonomous driving rules and overcoming various obstacles.

1.2 Objectives

The primary objective of this project is to enable the TurtleBot3 Waffle Pi to autonomously navigate the custom racing rig, following established autonomous driving rules and overcoming various obstacles. To accomplish this, the project focuses on the following tasks:

- Install ROS on a Linux platform and set up the necessary ROS packages for the TurtleBot3 Waffle Pi rover.
- Test and improve the basic functionality of the TurtleBot3 Waffle Pi, including SLAM, navigation, and camera image acquisition. The tuning parameters for SLAM and navigation will be adjusted to optimize the rover's performance.
- Complete the construction of a racing rig suitable for the TurtleBot3 Waffle Pi, featuring a racing track, parking area, tunnel, traffic light, and traffic signs. This task includes 3D printing various elements for the racing rig.
- Test the TurtleBot3 Waffle Pi's autonomous driving capabilities on the racing rig, ensuring that the rover follows autonomous driving rules, such as driving along the road, stopping at red lights, reading traffic signs, parking correctly, and navigating through the tunnel.
- Use tools such as TensorFlow, Keras, and Python to implement reinforcement learning for obstacle avoidance during navigation, utilizing Gazebo and ROS as simulation platforms.

1.3 Methods

The methodology for this project encompasses the following stages:

- Finalizing the racing rig, which involves designing and adding traffic signs and lights.
- Setting up the project environment, which includes installing Linux, the Robot Operating System (ROS), and the necessary TurtleBot packages.
- Testing and fine-tuning the Simultaneous Localization and Mapping (SLAM) and navigation parameters to optimize TurtleBot3's basic functionality.
- Conducting autonomous driving tests on a simulated racing rig in Gazebo.
- Evaluating the TurtleBot3's autonomous driving performance on the physical racing rig.
- Implementing and testing reinforcement learning algorithms for obstacle avoidance in a simulated environment.

1.4 Scope

The scope of this research is limited to developing and testing an autonomous driving system for the TurtleBot3 Waffle Pi mobile rover using ROS, machine learning algorithms, and reinforcement learning. The research includes designing and constructing a custom racing rig for testing the autonomous driving system and evaluating the system's performance in actual and simulated environments.

1.5 Software

The software tools used in this research include:

- Word for documentation.
- Shapr3D for 3D design.
- Flashprint 5 and ideaMaker for 3D printer programs.
- Visio for diagramming.
- ROS for robot programming and control.
- Ubuntu (16.04 and 20.04) and Windows as operating systems.
- Visual Studio Code for code development.
- Gazebo and Rviz for simulation and visualization.
- Python for algorithm development.
- Arduino IDE for microcontroller programming.
- ChatGPT by OpenAI for troubleshooting and research.

1.6 Report layout

The report is structured as follows:

- Chapter 2: Background and Equipment, providing an overview of TurtleBot3 Waffle Pi, autonomous driving, machine learning, and the racing rig.
- Chapter 3: Setup and Configuration of TurtleBot3, describing the hardware assembly, installation of ROS and relevant packages, and tuning parameters for SLAM and navigation.
- Chapter 4: SLAM and Navigation with TurtleBot3, discussing the SLAM algorithm, navigation algorithm, and navigation parameter tuning.
- Chapter 5: Camera Calibration and Obstacle Detection, covering camera calibration, lane detection, and detection of various obstacles such as traffic lights, traffic signs, and level crossings.
- Chapter 6: Autonomous Driving in Simulated and Real Environments, presenting the results of image processing, obstacle detection, SLAM-based navigation, and performance evaluation in both simulated and real racing rig environments.
- Chapter 7: Reinforcement Learning for Obstacle Avoidance, introducing reinforcement learning, deep Q-networks, learning process, results analysis, and future work.
- Chapter 8: Discussion, interpreting the results, discussing implications, limitations, challenges, and future directions for research.
- Chapter 9: Conclusion, summarizing the main contributions and outlining the implications of the research findings.

2 Background and Equipment

This chapter provides an overview of the development of autonomous systems and introduces the equipment used in this project, including the TurtleBot3 platform and the Autorace racing rig.

2.1 Introduction to Autonomous Systems Development

The field of autonomous navigation, which aims to create vehicles and robotic systems capable of moving independently and safely through their environments, has grown rapidly in the last decade. Advancements in various technologies, such as computer vision, machine learning, sensor fusion, and control systems, have driven the development of autonomous navigation systems like self-driving cars, drones, and mobile robots. These autonomous systems can interpret their environment, make decisions, and execute appropriate actions in real-time. This is done by using a mix of multiple components and technologies, such as:

1. Sensing and Perception: Sensors like LiDAR, RADAR, ultrasonic sensors, and GPS, are used by autonomous systems to perceive and collect data about the environment, including information about obstacles, lanes, traffic signs, etc.
2. Localization and Mapping: Techniques such as Simultaneous Localization and Mapping (SLAM) are used to accurately determine the robot's position and orientation, which are important to navigate the environment safely.
3. Path Planning and Control: The autonomous system must find a safe and efficient path to its destination after perceiving its surroundings and determining its pose and position. This can be done using a path planning algorithm like, for example, Dijkstra's or A* algorithms to help find the optimal route. A control system, such as a proportional-integral-derivative (PID) or a Model predictive control (MPC) controller, is then used to ensure the robot follows the planned path while maintaining a safe speed and distance from obstacles.

2.2 Robotics Operating System (ROS)

The Robotics Operating System (ROS) is an open-source framework for developing robotics software, offering tools, libraries, and conventions to streamline the creation and management of complex robotic systems. ROS has become popular among researchers and developers due to its flexibility, modularity, and support for various hardware platforms. Its ecosystem includes a vast collection of software packages for mapping, localization, navigation, and perception tasks, making it an ideal choice for autonomous systems development. In ROS, software modules called nodes communicate with each other using a publisher-subscriber model, exchanging messages through topics. ROS organizes software components into packages, which can be easily shared and reused within the community, promoting collaboration and accelerating development. Additionally, ROS provides various tools for visualizing data, debugging, and monitoring the performance of robotic systems. (Open Robotics, 2022)

2.3 TurtleBot3

TurtleBot3 is a low-cost, modular, customizable mobile robot platform based on the ROS framework. It is designed for research, education, and prototyping purposes and is used worldwide. The platform offers many different configurations, but the two focused on in this project are the TurtleBot3 Burger and the TurtleBot3 Waffle Pi, which include core technologies such as SLAM and Navigation. The Burger and Waffle Pi configurations differ in size, payload capacity, and hardware specifications. The Burger is smaller and more affordable, featuring a Raspberry Pi 3 as its main controller, a 360-degree LiDAR sensor, and an inertial measurement unit (IMU). The Waffle Pi is a slightly larger platform with a Raspberry Pi 4 and multiple sensors, including a 360-degree LiDAR, a Raspberry Pi camera, and an IMU. The TurtleBot3 platform uses the ROS framework for its software stack, making it easy to integrate with other ROS-based robotic systems.

2.4 Autorace

Autorace is a competition designed for autonomous robots, such as the TurtleBot3, based on the ROS framework. The objective of the competition is to develop and implement autonomous driving algorithms to navigate various obstacles on a small-scale racetrack. Participants are ranked based on points awarded for successfully clearing obstacles and the total time to complete the race. More information about the racing rig, rules, and obstacles can be found on the Robotis website (ROBOTIS, 2022).

2.5 Racing rig

As mentioned in Chapter 1, this is a continuation of an earlier project where the racing rig was designed based on the official Autorace competition track, and the road was constructed and painted. The rig includes a series of obstacles the robot must recognize and navigate. A Visio drawing of the designed racing rig can be seen in Figure 2.1, while the finished road can be seen in Figure 2.2. A detailed description of the design and construction process can be found in (Haugseter & Lauritzen, 2022). While the road was complete, several components still needed to be added, including traffic signs, traffic lights, obstacles, and the tunnel. These were used for testing TurtleBot3's ability to recognize and navigate obstacles autonomously.

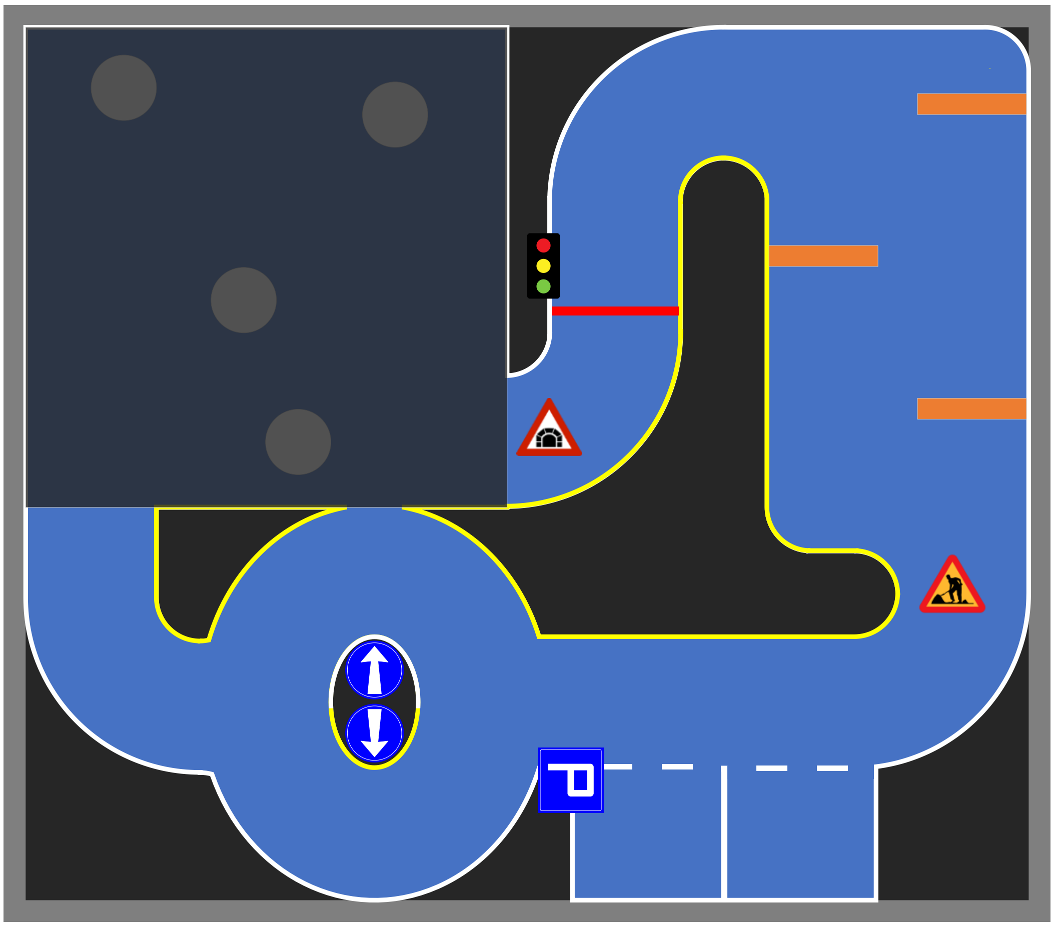


Figure 2.1: Racing track design and obstacle layout

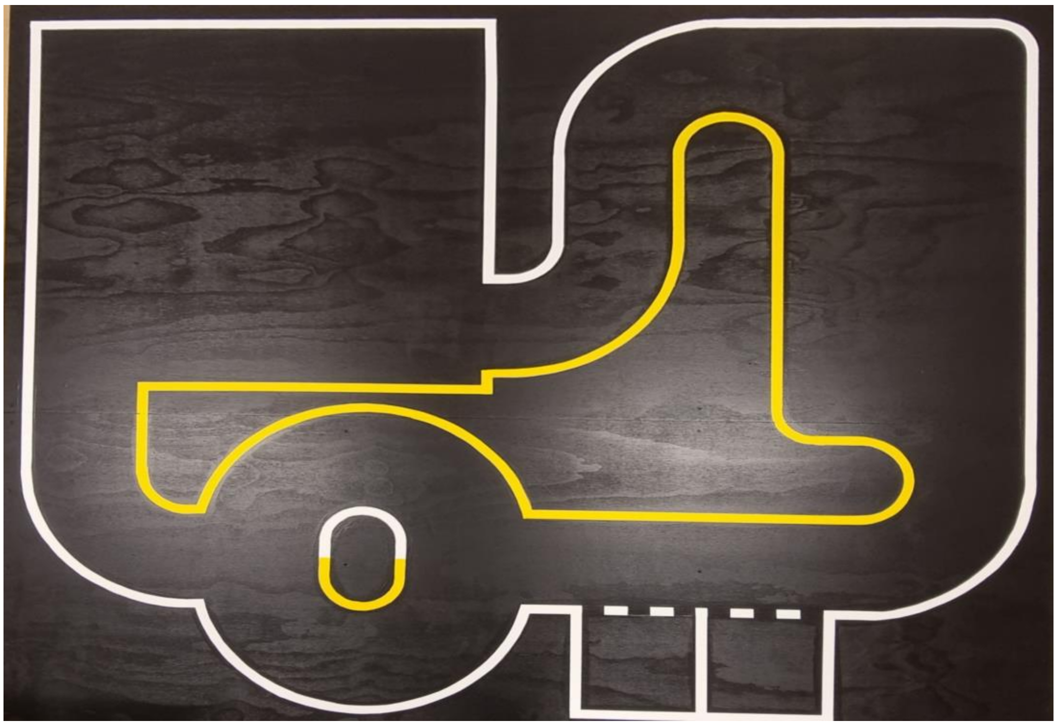


Figure 2.2: Existing racing track

2.5.1 Traffic light

One of the obstacles the TurtleBot3 should recognize and navigate correctly is the traffic light. To accomplish this, a traffic light had to be created from scratch. The first step in creating the traffic light was to design a circuit controlling the LEDs. The circuit, as seen in Figure 2.3, included an Arduino Nano Every, three LEDs in the colors red, yellow, and green, three 220-ohm resistors, a switch, and a battery. The Arduino Nano Every was chosen for its small size, making it possible to integrate it inside the traffic light. The resistors limit the current through the LEDs, preventing them from burning out. The switch enables the traffic light to be turned off, the battery was used to power the circuit.

Once the circuit was designed, the next step was to write a program for the Arduino Nano Every using Arduino IDE. The program is a simple sequence of delays and switching between the different colors of the LEDs. The code for the program can be found in Appendix B.

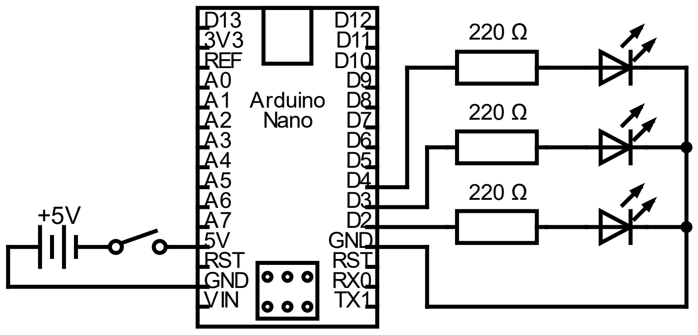


Figure 2.3: Traffic light circuit

The next step was to design a casing for the components. This was done using the 3D modeling program Shapr3D, as seen in Figure 2.4. The model consisted of four parts: a case for the Arduino and lights with the pole, a casing for the battery, and two lids. The size of the cases was made to be a small as possible while still fitting the components.

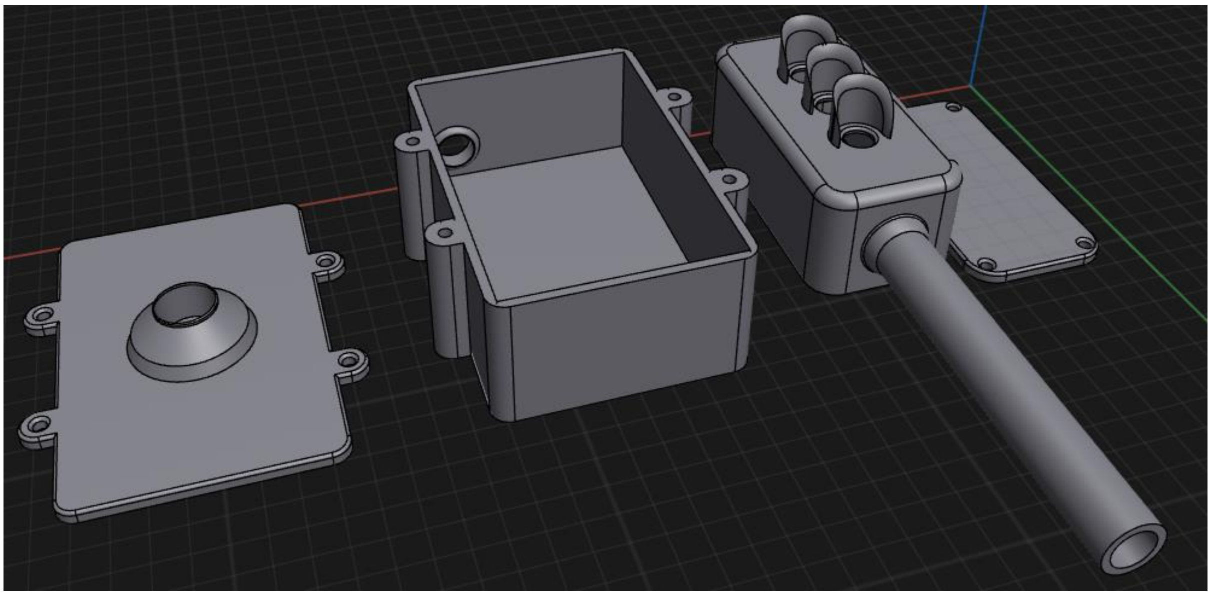


Figure 2.4: 3D model in Shapr3D

Once the design was complete, Flashprint 5 was used to add supports, slice the model, as seen in Figure 2.5, and convert it into machine code for the 3D printer. The 3D printer FlashForge

Adventurer 3 was used to print the parts of the casing. When the printing was done, it gave the parts shown in Figure 2.6. The parts were then assembled, and the electronics were added. The finished result with the light sequence can be seen in Figure 2.7.

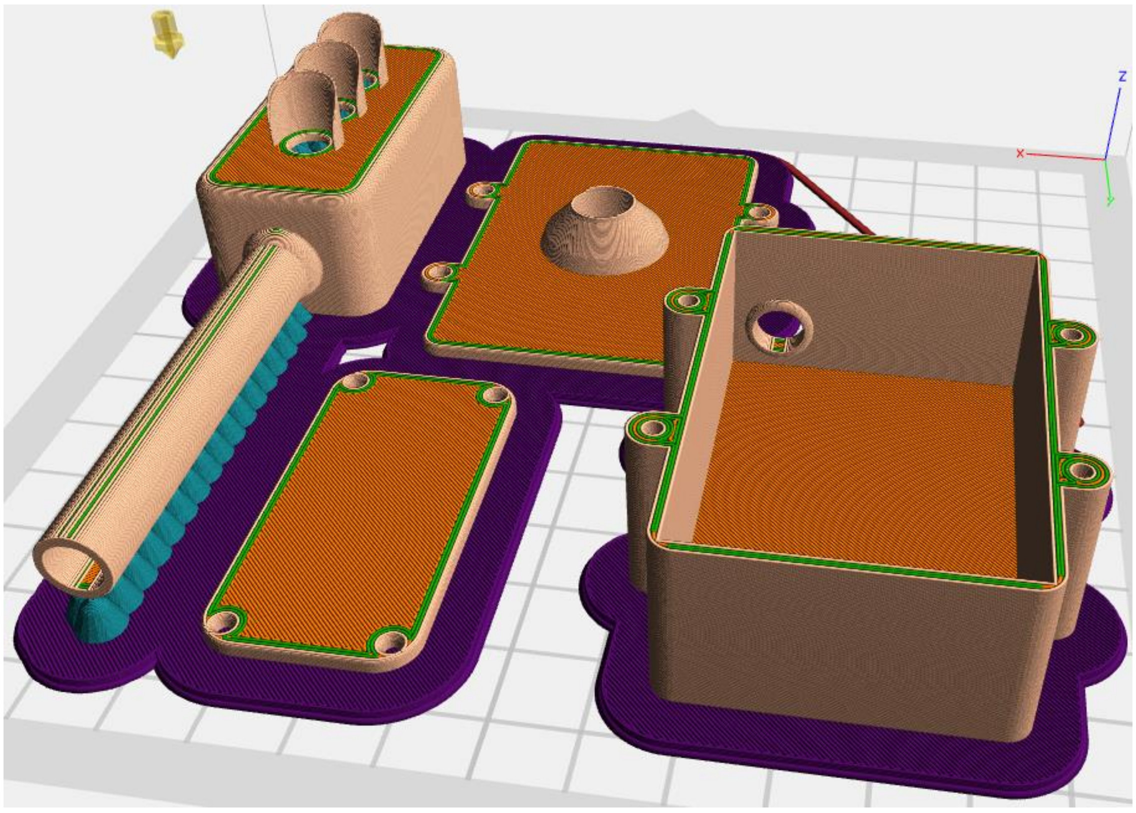


Figure 2.5: Sliced model in Flashprint 5 ready for printing

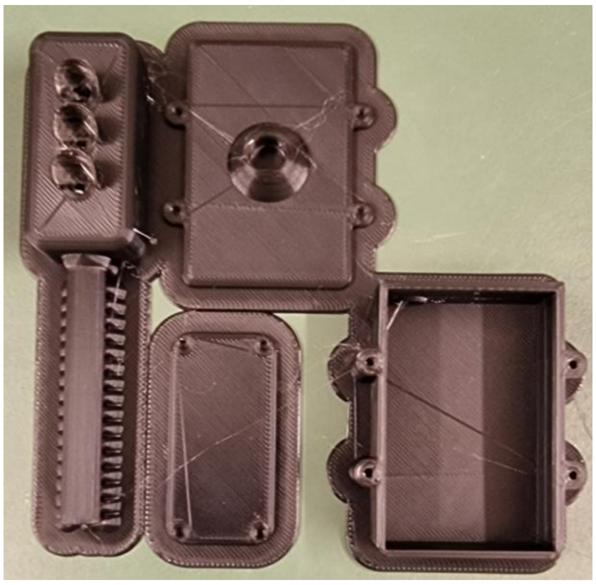


Figure 2.6: 3D printed parts before assembly

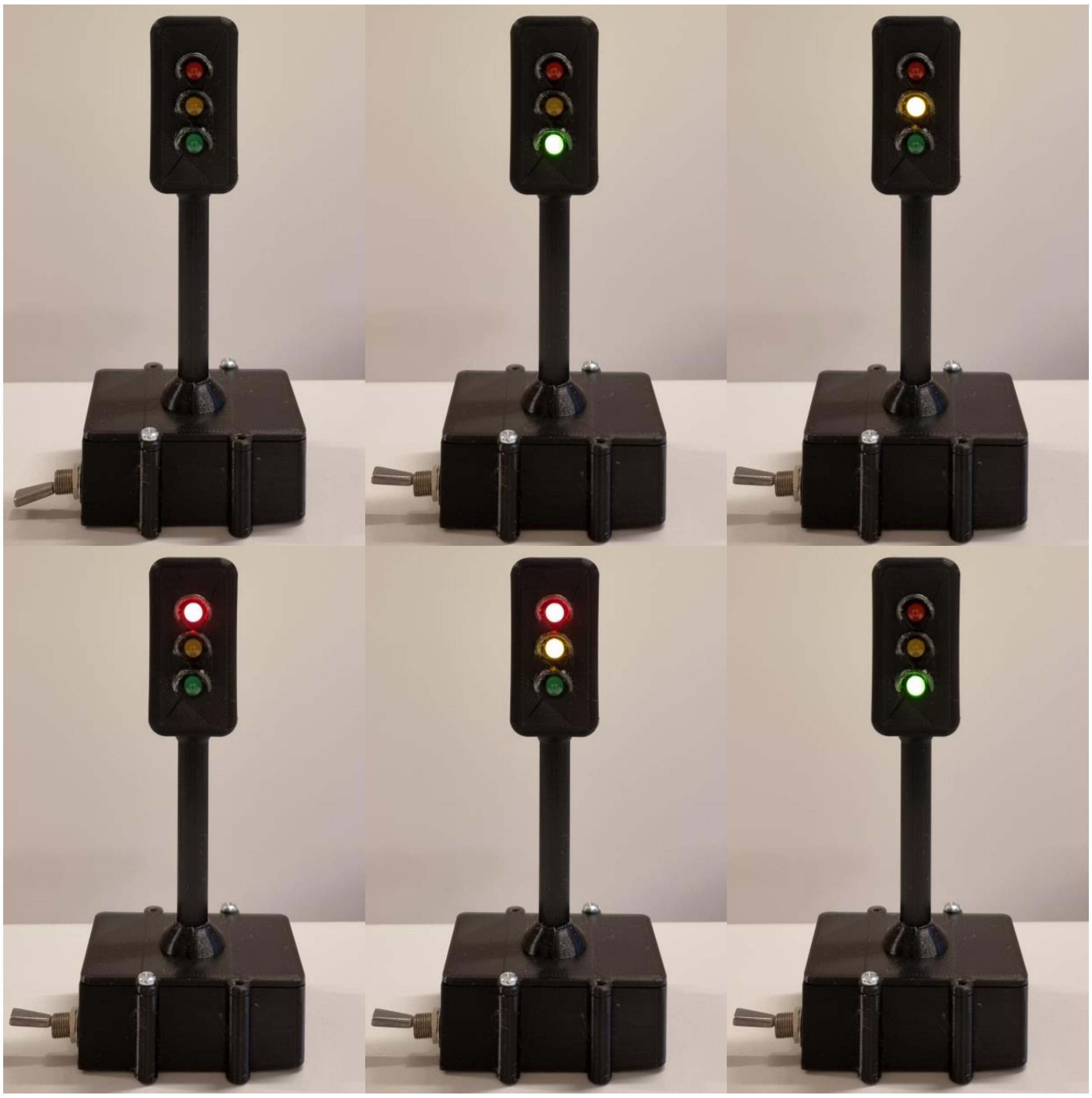


Figure 2.7: Finished traffic light, complete color sequence.

2.5.2 Traffic signs

One of the main things missing from the racing rig was traffic signs that the robot should recognize before each obstacle. Traffic signs play an essential role in the racing rig, activating the different missions that allow the TurtleBot3 to identify and navigate the various obstacles.

Three different variations, round, square, and triangular, of traffic signs were created and imported into the slicing software Flashprint, which divided the object into a stack of flat layers and described these layers as linear movements for the 3D printer. These instructions were then exported as g-code files to the FlashForge Adventurer 3 3D printer to produce the final physical models.

The sign and baseplate were printed as separate parts, then assembled. The final step was to print out a sticker that was attached to the 3D-printed sign, which contained the specific symbol and text that the TurtleBot3 was programmed to recognize. The result can be seen in Figure 2.8.

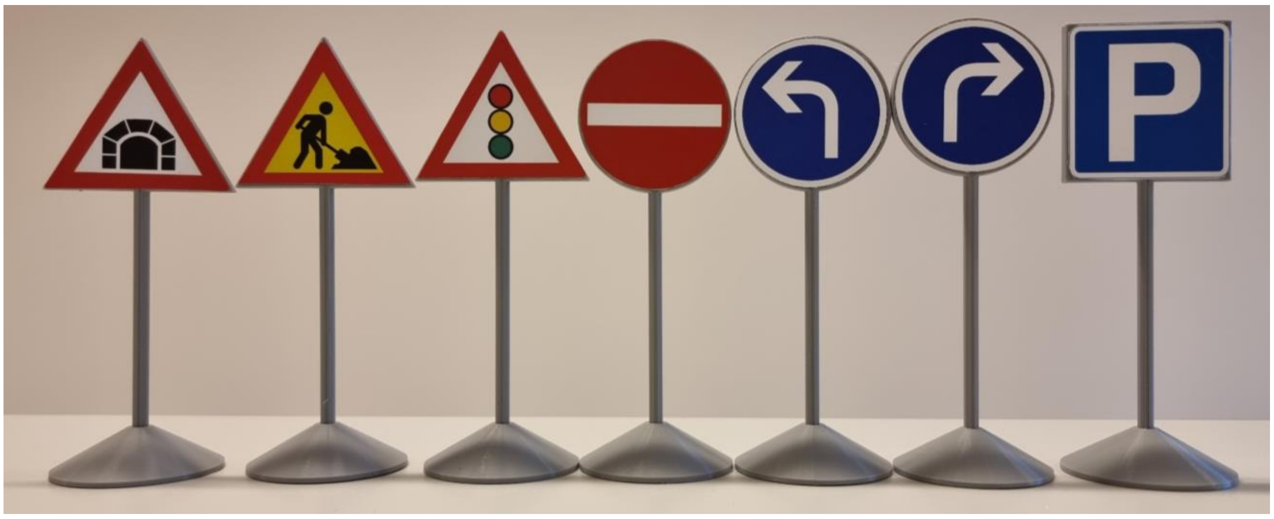


Figure 2.8: Traffic signs

2.5.3 Tunnel

The last obstacle missing on the racing rig is the tunnel. The tunnel is a dark, enclosed square box with several obstacles and one entrance and exit. It has a dimension of 110 x 110 cm and a height of 24 cm. The USN staff did the construction of this tunnel. The tunnel is designed to test the robot's navigation capabilities using SLAM.

3 Setup and configuration of TurtleBot3

This section provides an overview of two different TurtleBot3 configurations used in this project, explaining the assembly and software installation.

3.1 Hardware Assembly

Two different configurations of the TurtleBot3 were tested in this project, which can be seen in Figure 3.1. The model on the left is built following the official Turtlebot3 configurations, while the one on the right is a custom configuration that mixes the Turtlebot3 Burger and Turtlbot3 Waffle Pi. Both versions are built using a TurtleBot3 Waffle Pi kit, except for the Raspberry Pi 3B used in the Turtlebot3 Burger. For a more detailed assembly guide, refer to the TurtleBot3 documentation by ROBOTIS (ROBOTIS, 2022).

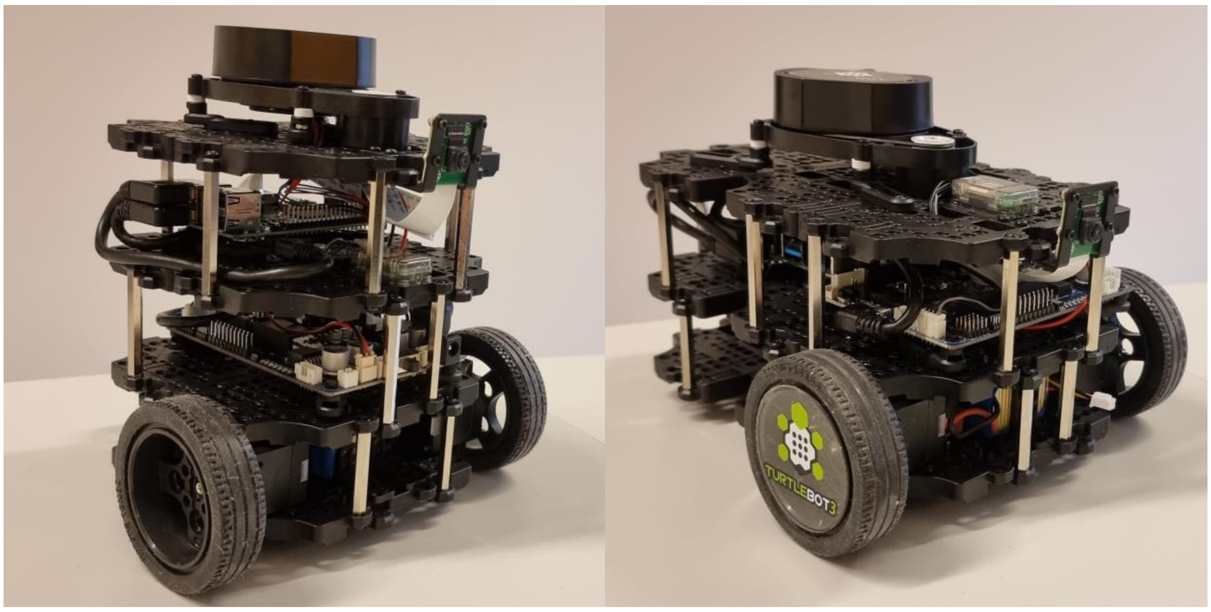


Figure 3.1: Assembled turtlebot3 rovers

3.1.1 TurtleBot3 Burger with Raspberry Pi 3B

The first configuration is the standard TurtleBot3 Burger equipped with a Raspberry Pi 3B. The hardware components are assembled in four layers, as depicted in Figure 3.2, in the following order:

1. First layer: The Dynamixel servos and the battery pack are mounted on the base plate.
2. Second layer: The OpenCR board is installed above the first layer.
3. Third layer: The Raspberry Pi 3B is placed above the OpenCR board.
4. Fourth layer: The LDS-01 LiDAR sensor and the Raspberry Pi camera are mounted on the top layer.

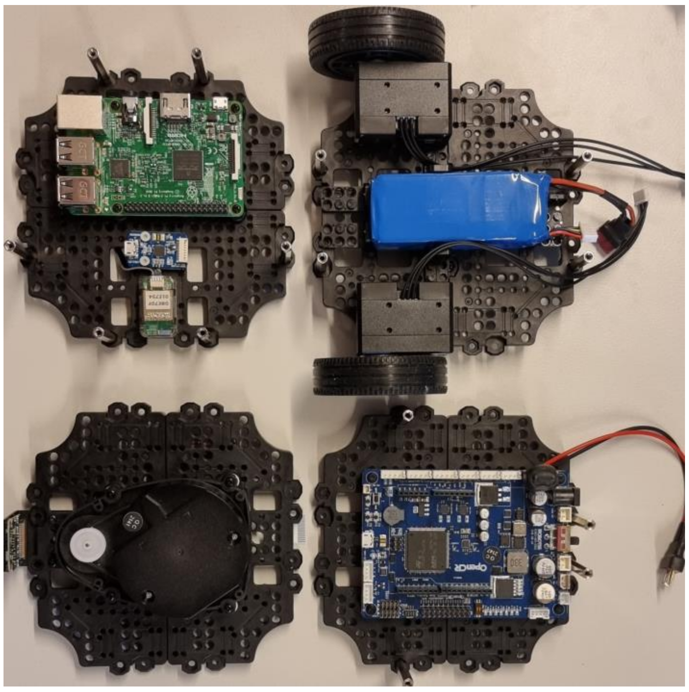


Figure 3.2: The four layers of the turtlebot3 Burger

All connections between components were carefully checked and secured to ensure proper functioning and prevent potential issues during operation.

3.1.2 Custom TurtleBot3 with Raspberry Pi 4

The second configuration is a custom-built TurtleBot3 with a Raspberry Pi 4. This design has the height of the TurtleBot3 Waffle Pi and the width of the Turtlebot3 Burger. A size comparison between the custom TurtleBot3 and the TurtleBot3 Waffle Pi can be seen in Figure 3.3. The hardware components, which are from a turtlebot3 waffle pi, are assembled in the following order:

1. First layer: The Dynamixel servos and the battery pack are mounted on the base plate.
2. Second layer: The OpenCR board and the Raspberry Pi 4 are installed above the first layer.
3. Third layer: The Raspberry Pi camera and the LDS-01 LiDAR sensor are mounted on the top layer.

All connections between components are carefully checked and secured to ensure proper functioning and prevent potential issues during operation. With the hardware assembly of both TurtleBot3 configurations complete, the next steps involve the software setup and configuration for each robot.

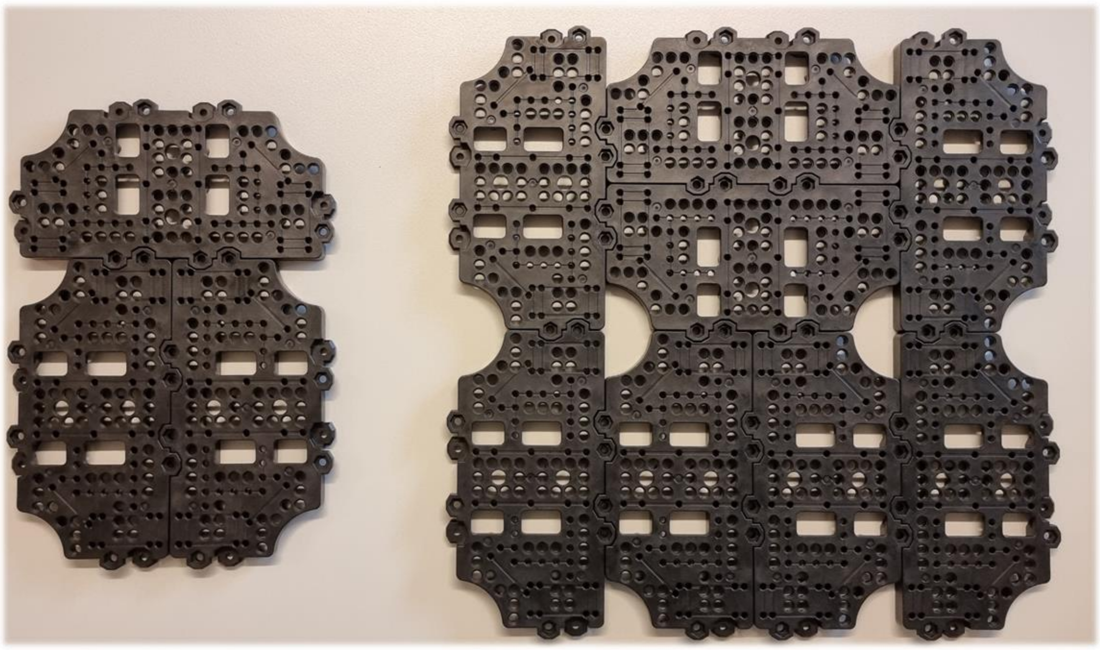


Figure 3.3: Custom base plate vs. TurtleBot3 Waffle Pi base plate.

3.2 Installation of ROS and relevant packages

The software setup for the TurtleBot3 Burger and the custom TurtleBot3 is mostly the same but with different packages. The software setup consists of three main steps for both versions of the TurtleBot3: the PC setup, SBC setup, and OpenCR setup. The commands for downloading all relevant packages can be found in Appendix C for Kinetic and Appendix D for Noetic.

The first part of the software setup was setting up a computer with Ubuntu. For ROS Kinetic, Ubuntu 16.04 is required, while ROS Noetic requires Ubuntu 20.04. Ubuntu is an open-source Linux-based operating system popular with robotics and software development. In this project, both versions were installed by partitioning the disk and dual booting the pc. With the correct version installed, the following steps were to set up the development environment by installing ROS (Kinetic or Noetic) and Turtlebot3 packages, including, among others, turtlebot3_msgs, and turtlebot3_simulations packages. The last step for the PC setup was to set up the correct IP in the ~/.bashrc file.

The second part is the Single Board Computer (SBC) setup or the setup for the raspberry pi. The Raspberry Pi operating system is installed using a microSD card on the PC using Raspberry Pi Imager. Then configure the Wi-Fi network settings with the microSD card still connected to the PC by editing the "50-cloud-init.yaml" file with the correct SSID and password. Then inserted into the Raspberry Pi, and the TurtleBot3 was powered on. For this project, Angry IP finder was used to find the IP, and the remote computer was used to connect to the raspberry pi using SSH with the default username "ubuntu" and password "turtlebot". Alternatively, a keyboard and screen can be connected to obtain the IP. Then, the network is configured in the ~/.bashrc file with the PC and Raspberry Pi IP addresses. Finally, ROS (Kinetic or Noetic) and the necessary TurtleBot3 packages are installed on the Raspberry Pi. Some important installations include ROS navigation, SLAM, and teleoperation packages. The detailed instructions for the SBC setup can be found in the TurtleBot3 SBC Setup Guide.

The third and final step was the setup of the OpenCR, which is responsible for controlling the motors, sensors, and other peripherals. The setup includes installing the required drivers, uploading the TurtleBot3 firmware, and testing the board's functionality. Some crucial components installed in this step are the Dynamixel SDK, used for controlling and managing Dynamixel motors, and the OpenCR firmware, the operating software for the OpenCR board.

4 SLAM and Navigation with TurtleBot3

This chapter goes through SLAM and Navigation on the TurtleBot3, explains the algorithms used, and the tuning process for both SLAM and Navigation.

4.1 SLAM algorithm for dynamic map generation

Simultaneous Localization and Mapping is a technique to build a map of a robot's environment while estimating its positions within that map (Durrant-Whyte, 2006). This can be done with several sensors like LiDAR, cameras, and sonar. For this project, the LiDAR "LDS-01" is used for mapping the environment. LDS-01 is a 360-degree laser scanner 2D LiDAR that provides accurate distance measurements to surrounding obstacles. This is used for both constructing the map and estimating the robot's pose within that map.

TurtleBot3 uses the Gmapping algorithm for SLAM, which is an implementation of the Rao-Blackwellized Particle Filter (RBPF) SLAM approach. Gmapping improves upon the traditional RBPF approach by using an enhanced proposal distribution and adaptive resampling to address the particle-depletion problem, which occurs when meaningful particles are eliminated during the resampling step (Wang, 2020) This results in a more efficient and robust SLAM algorithm suitable for real-time applications in mobile robots like the TurtleBot3. The algorithm is implemented as a ROS package, facilitating seamless integration into the robot's software stack.

In Gmapping, the map is represented as an occupancy grid. Each cell indicates the probability of that cell being occupied by an obstacle. The algorithm uses the raw LiDAR data to update the occupancy grid and TurtleBot3's position simultaneously. RBPF is employed for more efficient localization by dividing the problem into robot localization and feature position estimation. A particle filter is used to estimate the robot's position, while Extended Kalman Filters (EKFs) are used to estimate the position of environmental features, significantly reducing the number of particles needed to localize the robot's position.

The gmapping algorithm consists of five main steps: initialization, sampling, updating the importance weighting, adaptive resampling, and map estimation (Wang, 2020). Motion update adjusts the robot's pose based on odometry information, while the sensor update uses LiDAR measurements to update the occupancy grid map and the particle weights. Resampling selects particles with higher weights, discarding those with lower weights, to converge on the most likely map and pose hypothesis. Loop closure detects and corrects the map and robot's pose when the robot revisits a previously mapped area.

4.2 Navigation algorithm for autonomous driving

The TurtleBot3 employs a two-tier navigation algorithm with a global and local planner. The global planner devises a comprehensive path from the robot's current position to the goal position, while the local planner generates the robot's local trajectories, ensuring its ability to navigate the environment safely, avoiding real-time obstacles.

The global planner deploys the graph search algorithm A* to find the shortest path between the initial and goal positions on the occupancy grid map produced by the SLAM process. The A* algorithm can find the optimal path efficiently using a heuristic function that guides the search towards the goal. This heuristic function estimates the remaining cost from the current node to the goal node, significantly reducing the number of nodes explored compared to algorithms like Dijkstra's that lack heuristic information (Educative, 2023).

Upon generating the global path, the Adaptive Monte Carlo Localization (AMCL) algorithm estimates the robot's pose within the map, merging data from the robot's odometry and the laser range finder. AMCL is a probabilistic localization technique that employs a particle filter to represent the distribution of the robot's pose, allowing for accurate and robust localization even in noisy and uncertain environments (Lauttia, 2022).

The local planner, specifically the Dynamic Window Approach (DWA), uses the global path and the robot's current pose to generate local trajectories. These trajectories enable the robot to navigate the environment while avoiding obstacles in real-time. The DWA local planner considers the robot's kinematic constraints, such as maximum and minimum velocities and accelerations, to generate smooth and safe trajectories (Fox, Burgard, & Thrun, 1997)

The DWA local planner evaluates multiple trajectories based on criteria like distance to obstacles, adherence to the global path, and the robot's velocity. It then selects the optimal trajectory, which is sent to the robot's controller for execution. This process is performed continuously, enabling the robot to navigate dynamic environments and respond to changes in the environment or the global path.

4.3 Navigation Parameters and Tuning

This chapter discusses the most crucial navigation parameters for the TurtleBot3 autonomous navigation system and the tuning of these parameters to optimize performance in different environments. The TurtleBot3 navigation system uses ROS navigation stack, seen in Figure 4.1, showing the various components that work together to enable autonomous navigation. The robot's movement is planned in the move_base box, including a global and local costmap, planner, and recovery behaviors. They use the information from the stored map, sensor readings, odometry, and AMCL. The navigation stack can be configured and fine-tuned using a series of parameters set in the YAML configuration files.

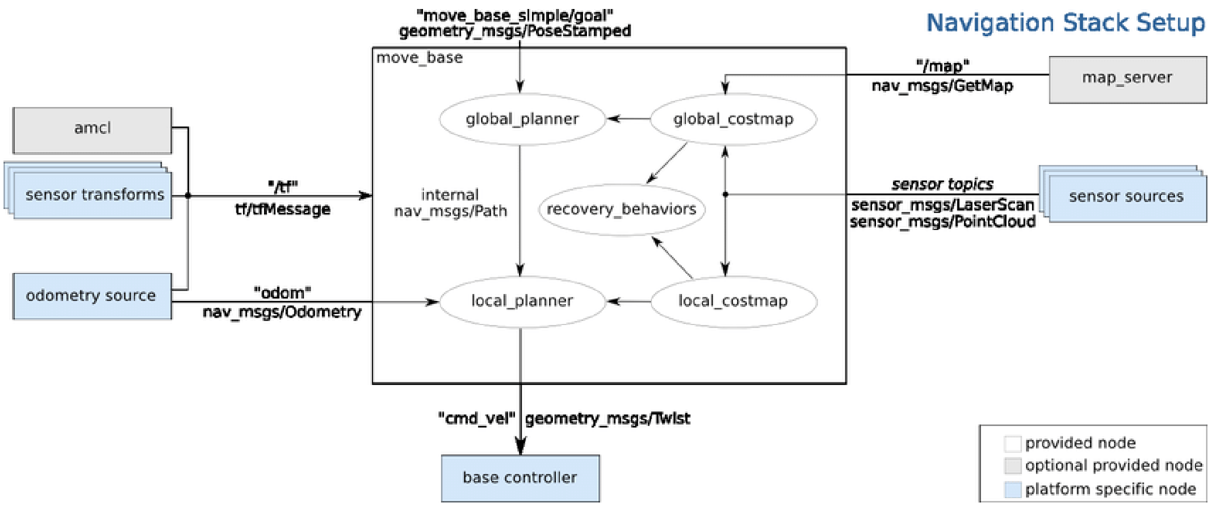


Figure 4.1: ROS navigation stack (Granosik, 2016)

Before moving to the different parameters important to know some of the primary components of the ROS navigation stack:

1. Global Planner: Generates a high-level path for the robot from its current position to the goal position, considering the available map and static obstacles.
2. Local Planner: Generates a local trajectory for the robot to follow, considering the global planner, dynamic obstacles, and real-time sensor data.
3. Costmap: Represents the environment as a grid of cells, each with an associated cost indicating the distance from the robot to the obstacles (SCHAETZEN, 2019). The global costmap is generated based on the known map and is used by the global planner, while the local costmap incorporates real-time sensor data and is used by the local planner.

4.3.1 Parameters

Two of the most important parameters are for the costmap affecting how close to the obstacle the global and local planner can plan a route for the robot. The costmap is a vital navigation system component, representing the robot's understanding of the environment.

- *inflation_radius*: This parameter defines the inflation area around obstacles to prevent the robot from getting too close. Increasing this value will create a larger buffer around obstacles, making path planning more conservative. Decreasing this value will allow the robot to plan paths closer to the obstacles.
- *cost_scaling_factor*: This factor is multiplied by the cost value and is inversely proportional to it. Increasing this value will decrease the cost, leading the robot to plan paths closer to obstacles. Lowering this value will make the robot stay further away from obstacles.

The local planner generates a local trajectory for the robot to follow while considering dynamic obstacles and incorporating real-time sensor data. The TurtleBot3 navigation system uses the DWA (Dynamic Window Approach) local planner, which has several parameters that can be tuned to optimize performance:

- *max_vel_x*, *min_vel_x*, *max_trans_vel*, and *min_trans_vel*: The parameters determine the robot's maximum and minimum translational velocity. A negative minimum value allows the robot to move backward.
- *max_rot_vel* and *min_rot_vel*: Sets the maximum and minimum values of rotational velocity.
- *acc_lim_x* and *acc_lim_theta*: Sets the limits for translational and rotational acceleration. Changing these values will affect how quickly the robot can accelerate or decelerate.
- *xy_goal_tolerance* and *yaw_goal_tolerance*: Gives the allowed x,y distance, and yaw angle from the target the robot can be when reaching its goal pose. Increasing these values will enable the robot to stop further from the goal. A too-low value can cause the robot to drive in a circle and never stop as it can't reach its exact target.
- *sim_time*: Sets the forward simulation times in seconds for the local planner, in other words, how far along the path it should plan. Giving it a too low will result in being unable to pass narrow areas, but two large values can also cause trouble.

The parameters for tuning the costmap and local planner can be found inside the following files:

- TurtleBot3_navigation/param/costmap_common_param_Burger.yaml
- TurtleBot3_navigation/param/dwa_local_planner_params_Burger.yaml

4.3.2 Tuning Parameters for Efficient Navigation

This section describes the tuning process for the different parameters to enable more efficient navigation. The primary parameters adjusted are xy_goal_tolerance, yaw_goal_tolerance, inflation_radius, and cost_scaling_factor, which were tested with various values to identify the most optimal settings.

For the local planner, the following values were found after some testing:

- xy_goal_tolerance: Set to 0,1, the robot can stop near the target while avoiding circular driving behavior due to unreachable precision.
- yaw_goal_tolerance: Set to 0,3, it provides the robot with an acceptable angular tolerance when reaching its goal pose.

These values were tested at their extremes to understand better how the inflation_radius and cost_scaling_factor parameters affect the costmap and navigation. Figure 4.2 shows the results obtained from four different combinations of these values:

1. Low inflation_radius and low cost_scaling_factor: In this image, the costmap has a minimal buffer around obstacles due to the low inflation_radius setting. The low cost_scaling_factor causes the robot to plan paths closer to obstacles. This makes it so that the robot can navigate efficiently through the environment but with a higher risk of colliding with obstacles or getting stuck in tight spaces.
2. Low inflation_radius and high cost_scaling_factor: This image shows a costmap with a minimal buffer around obstacles, similar to Image 1. However, the high cost_scaling_factor leads the robot to plan paths further away from obstacles. The robot maintains a safer distance from obstacles but has a less efficient path, as it may take longer paths to avoid getting too close to obstacles.
3. High inflation_radius and low cost_scaling_factor: In this scenario, the costmap has a larger buffer around obstacles due to the high inflation_radius setting. The low cost_scaling_factor still encourages the robot to plan paths closer to obstacles, but the larger buffer ensures that the robot maintains a safe distance. This could be a good balance between efficiency and safety, depending on the complexity of the environment.
4. High inflation_radius and high cost_scaling_factor: In this image, the costmap features a large buffer around obstacles, similar to Image 3. The high cost_scaling_factor causes the robot to plan paths further away from obstacles, prioritizing safety over path efficiency. In this case, the robot will be more cautious when navigating the environment, but it may take longer to reach its destination due to the longer and more conservative paths it takes to avoid obstacles.

Based on the experiments, the following values were chosen for the costmap:

- inflation_radius: Set to 1, it ensures that the robot maintains a sufficient buffer around obstacles to prevent collisions.
- cost_scaling_factor: Set to 3, the robot can maintain a safe distance from obstacles without taking unnecessarily long paths.

By adjusting the parameters mentioned above, the robot can effectively navigate through the environment while maintaining an optimal balance between safety and efficiency. This fine-tuning allows the robot to navigate through tight spaces, such as a tunnel, while minimizing the risk of collisions and maintaining satisfactory path efficiency.

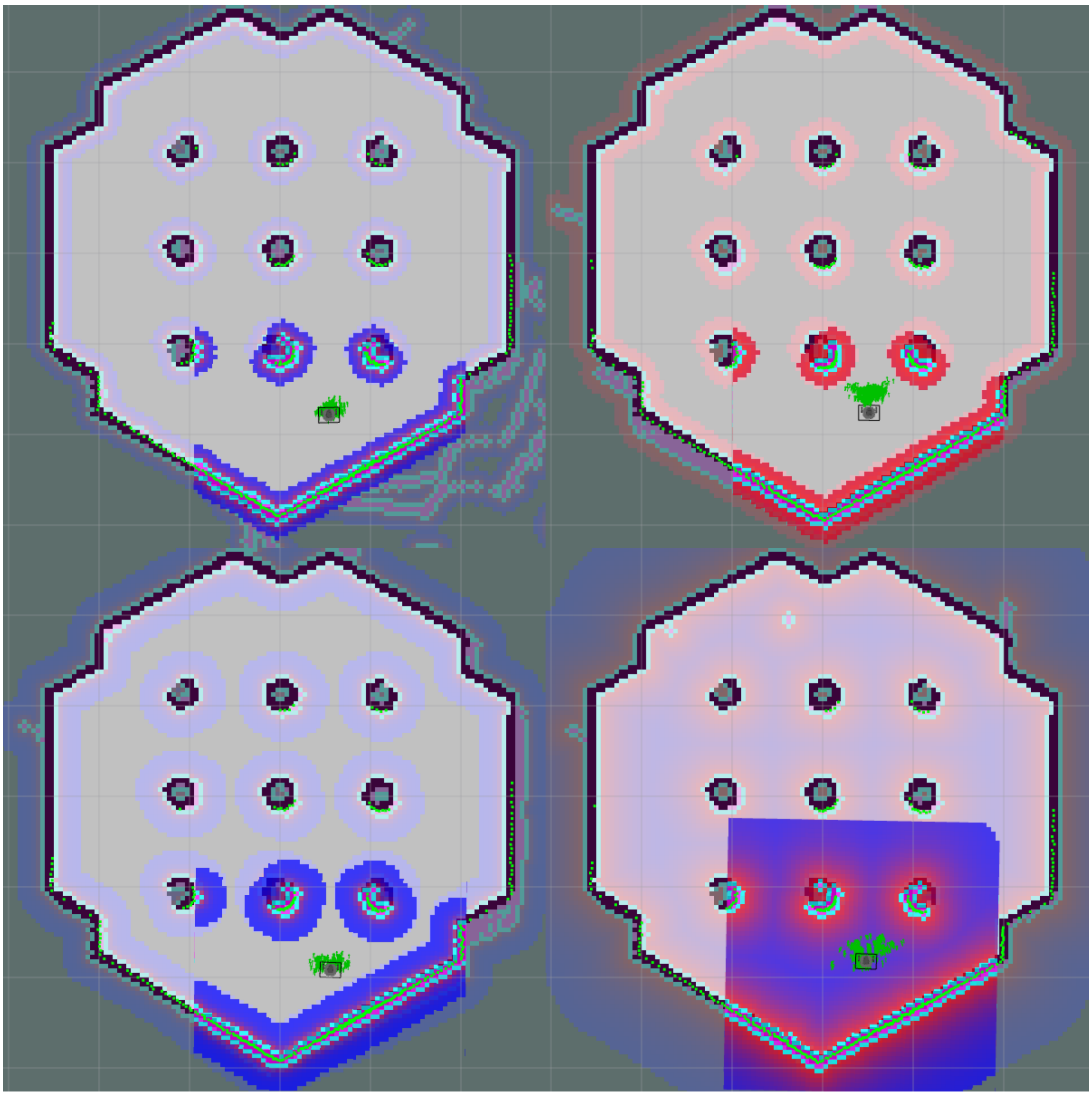


Figure 4.2: Visual Comparison of Inflation Radius and Cost Scaling Factor Effects

5 Image Processing, Feature Detection, and Parameter Configuration

This chapter discusses implementing image processing and detection techniques for autonomous navigation using the TurtleBot3 robot platform. The key steps include camera calibration, lane detection, traffic light detection, and traffic sign detection. The integration of these techniques with the TurtleBot3 navigation stack, achieved using the Robot Operating System (ROS), is also explained. For a more detailed guide on calibrating the camera on TurtleBot3, please refer to Appendix F.

5.1 Camera Calibration

An essential step before the image processing task is camera calibration. Camera calibration is a process for correcting distortions in the images and extracting accurate spatial and geometric information from them. Intrinsic and extrinsic camera parameters within computer vision describe the mathematical relationship between the 3D coordinates to the 2D projection onto an image plane (Zhang, 2016).

5.1.1 Intrinsic Calibration

Intrinsic calibration is the calibration of the camera's internal parameters, such as focal length, optical center, and distortion coefficients (Zhang, 2016). One way to do intrinsic camera calibration is to use a checkerboard pattern. When running the intrinsic calibration node, a window will open, as shown in Figure 5.1, where different parameters like X, Y, Size, and Skew are used to calibrate for any distortions in the camera. The checkerboard pattern is placed in front of the camera and moved around until it has a diverse set of calibration data. The captured images are then processed to detect the pattern corners and compute the intrinsic parameters, which are saved to a calibration file.

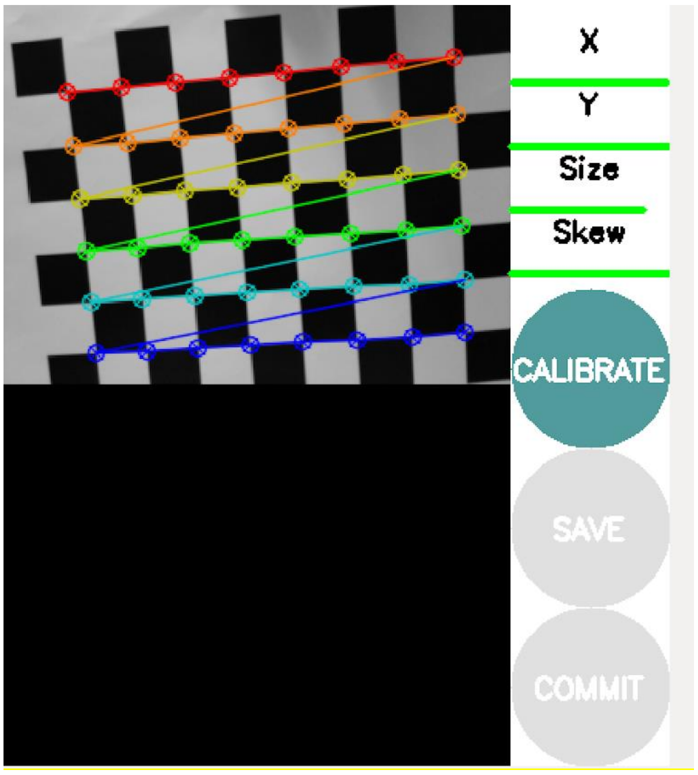
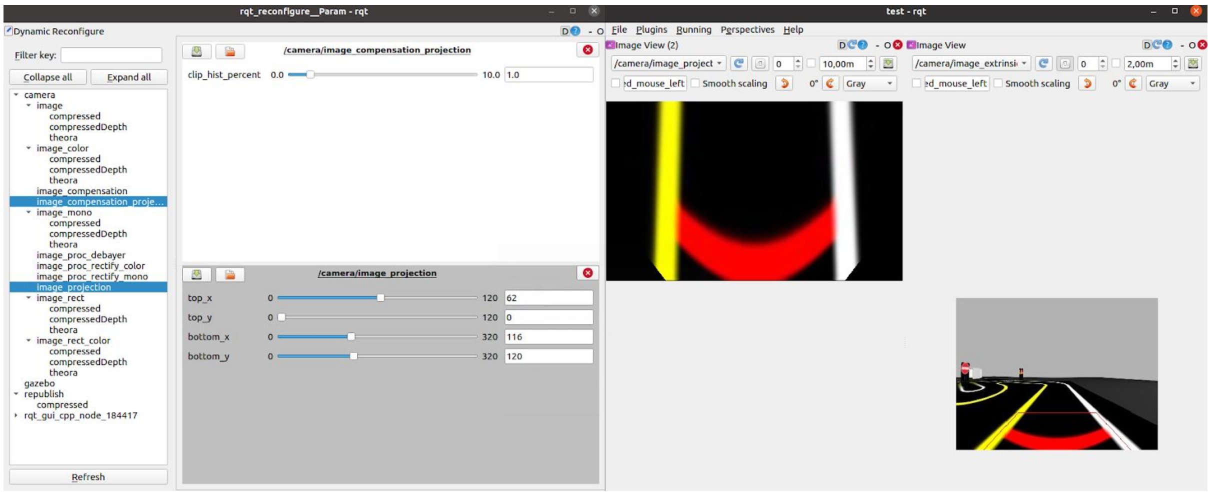


Figure 5.1: Checkerboard pattern for intrinsic calibration

5.1.2 Extrinsic Calibration

Extrinsic calibration is the calibration of external parameters of the camera, such as position and orientation, with respect to the world coordinate system using a transformation matrix (Zhang, 2016). This is important for understanding the relationship between the camera and other sensors on the robot to get a comprehensive understanding of the environment. In the context of TurtleBot3, extrinsic calibration involves adjusting the "image_projection" parameters to ensure the camera's position and orientation are accurately represented in the system. As shown in Figure 5.2, the calibration process involves adjusting the parameters "top_x," "top_y," "bottom_x," and "bottom_y." These adjustments ensure the camera's projection aligns with the robot's coordinate system, allowing for accurate mapping and navigation.



/camera/image_compensation_projection

clip_hist_percent	0.0		10.0	1.0

/camera/image_projection

top_x	0		120	62
top_y	0		120	0
bottom_x	0		320	116
bottom_y	0		320	120

Figure 5.2: Extrinsic Camera Calibration

5.2 Lane and obstacle detection

This section discusses the techniques and methods used by TurtleBot3 for detecting various features on the road, such as lanes, traffic lights, intersections, parking areas, construction zones, and tunnels. These detection tasks play a crucial role in the autonomous navigation capabilities of the TurtleBot3.

5.2.1 Lane Detection and tracking

Lane detection is one of the most important tasks for the TurtleBot3, as it allows the vehicle to identify and stay within the lanes and the robot to navigate between the obstacles. The TurtleBot3 uses a color-based approach for lane detection, focusing on isolating white and yellow lane markings from the input images. The calibration process for this can be found in Appendix G and is seen in Figure 5.3.

1. Color thresholding: The input image is converted to the HSV color space, and color thresholds are applied to isolate white and yellow lane markings, effectively separating the lanes from the rest of the image.

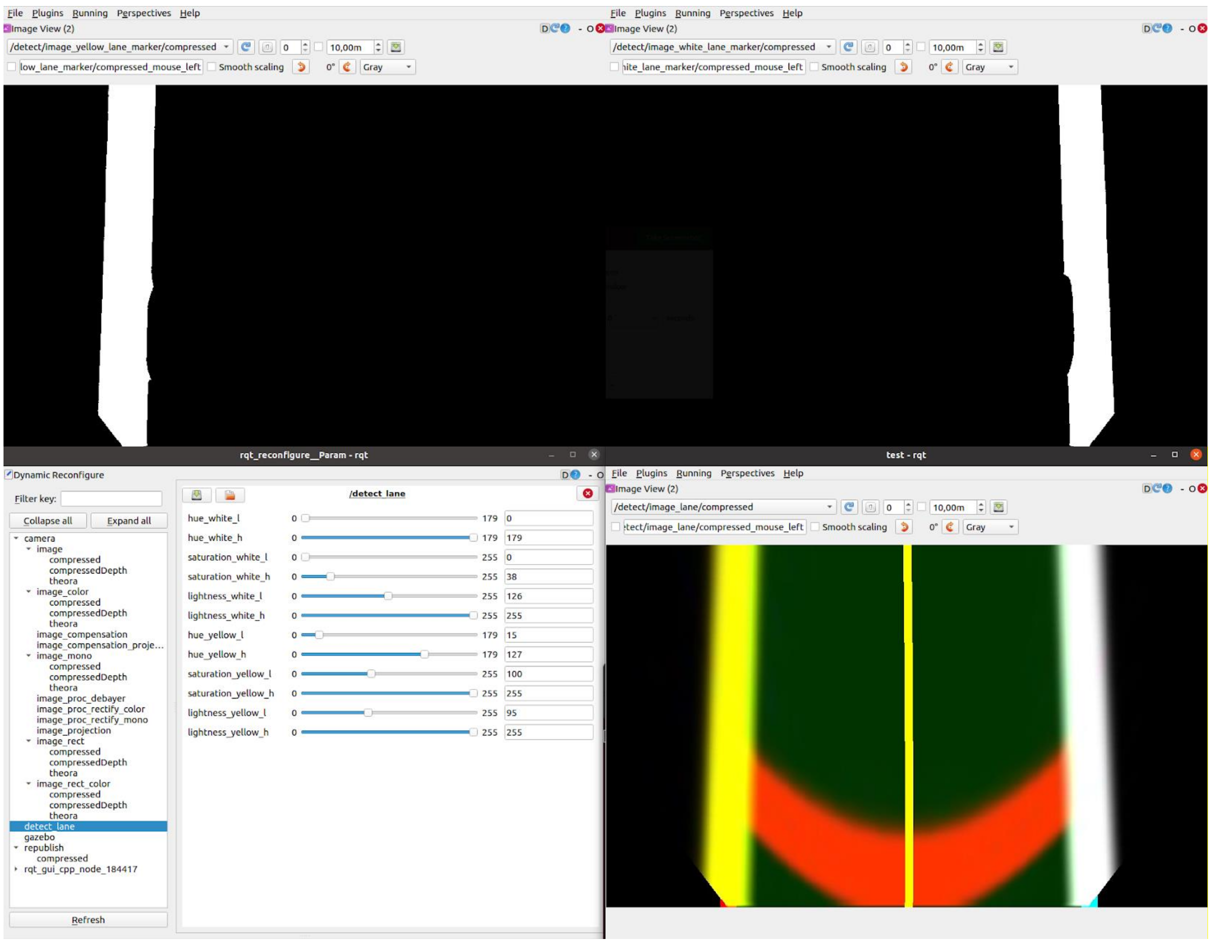
2. Lane detection: The percentage of white and yellow pixels in the thresholded images is calculated. If a sufficient number of pixels are detected, a second-order polynomial curve is fitted to the lane pixels using the sliding window technique, which helps identify the position and orientation of the lane lines.

3. Lane tracking: A moving average of the detected lane curves is maintained to improve stability and reduce noise. The moving average is calculated separately for both the left and right lanes.

4. Lane visualization: An overlay image with visual representations of the detected lanes is generated using the detected lane curves. Polylines representing the lane boundaries are drawn, and the lane area is filled with a color.

5. Lane reliability: The reliability of the detected lanes is evaluated based on the number of pixels found in the threshold images, and the reliability values are adjusted based on the consistency of lane detection.

6. Lane center estimation: If both white and yellow lanes are detected, the center of the road is calculated by averaging the x-coordinates of the left and right lane curves, and the estimated lane center is published as a Float64 message.



/detect/image_yellow_lane_marker/compressed
/detect/image_white_lane_marker/compressed
rqt_reconfigure__Param - rqt
Dynamic Reconfigure
test - rqt
/detect/image_lane/compressed
/detect_lane

```
/detect_lane
hue_white_l           0  179  0
hue_white_h           0  179  179
saturation_white_l    0  255  0
saturation_white_h    0  255  38
lightness_white_l     0  255  126
lightness_white_h     0  255  255
hue_yellow_l          0  179  15
hue_yellow_h          0  179  127
saturation_yellow_l   0  255  100
saturation_yellow_h   0  255  255
lightness_yellow_l    0  255  95
lightness_yellow_h    0  255  255
```

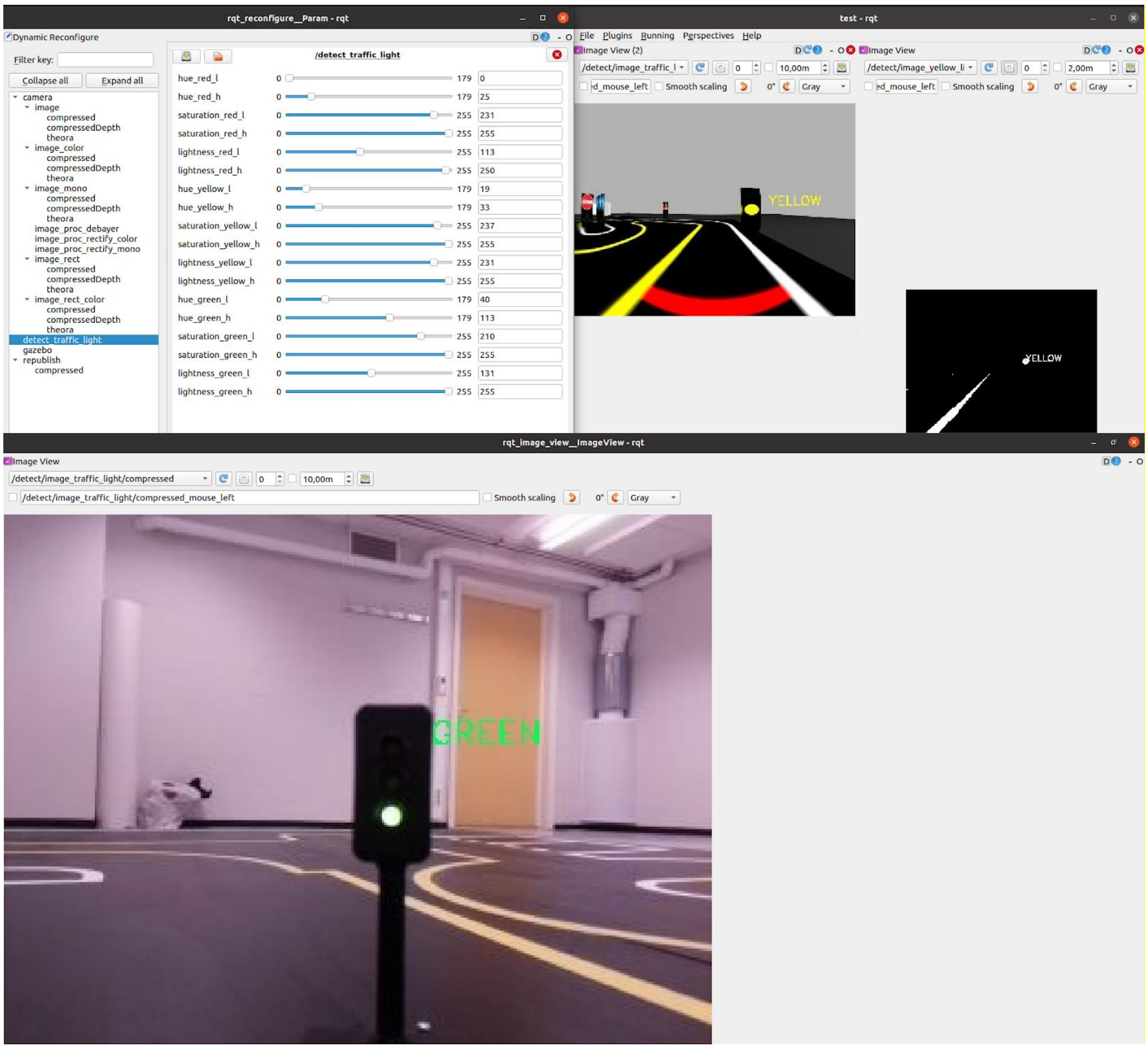
Figure 5.3: Lane detection

5.2.2 Traffic Light Detection

Detecting traffic lights is essential for the safe and legal navigation of autonomous vehicles. The Traffic light detection process involves identifying the current state of the traffic light (red, yellow, or green) and reacting accordingly. The detection process is performed by segmenting the image based on color and then applying a shape-based detection algorithm to identify the traffic light's position and state. The tuned parameters for color segmentation and shape detection are used to ensure accurate detection and avoid false positives. The following steps describe the process.

1. Color Thresholding: This step involves isolating specific colors associated with different traffic light signals using color thresholding techniques. It utilizes the HSV color space and defines color ranges for red, yellow, and green traffic lights. The configuration process for this can be seen in Figure 5.4.
2. Image Processing: To reduce noise and improve the accuracy of subsequent operations, the input image undergoes Gaussian blur processing.
3. Circle Detection: A blob detection algorithm identifies circular regions in the image that correspond to traffic lights. The algorithm's parameters are configured to filter out noise and select suitable blob candidates based on area, circularity, and convexity.

4. Traffic Light Classification: The detected circular regions are analyzed to determine the color of the traffic light. This is achieved by comparing the position of the detected circles relative to predefined regions of interest for each color (green, yellow, red).
5. Traffic Light Status: The system keeps track of the number of consecutive frames in which a particular traffic light color is detected. Separate counters are maintained for green, yellow, and red lights. If a specific color is detected for a sufficient number of frames, the system publishes the corresponding traffic light status (GREEN, YELLOW, RED) and sends control signals accordingly.



```
/detect_traffic_light
hue_red_l           0  179  0
hue_red_h           0  179  25
saturation_red_l    0  255  231
saturation_red_h    0  255  255
lightness_red_l     0  255  113
lightness_red_h     0  255  250
hue_yellow_l        0  179  19
hue_yellow_h        0  179  33
saturation_yellow_l 0  255  237
saturation_yellow_h 0  255  255
lightness_yellow_l  0  255  231
lightness_yellow_h  0  255  255
hue_green_l         0  179  40
hue_green_h         0  179  113
saturation_green_l  0  255  210
saturation_green_h  0  255  255
lightness_green_l   0  255  131
lightness_green_h   0  255  255
```

```
/detect/image_traffic_light/compressed
/detect/image_traffic_light/compressed_mouse_left
```

Figure 5.4: Traffic light detection

5.2.3 Construction Zone Detection and Navigation

Another obstacle the TurtleBot3 must navigate through is the Construction zone. The construction zone is a wider area of the road with three walls placed in the road that the Turtlebot3 must drive around. The TurtleBot3 system is designed to detect obstacles in construction zones using LiDAR and execute specific maneuvers to navigate around them. The process begins with obstacle detection, where the robot subscribes the LiDAR data to gather information about its surroundings. It scans for obstacles within a specific range in front of the vehicle and acts when an obstacle is detected. When detecting an obstacle, the TurtleBot3 executes a sequence of maneuvers to drive around it. During the navigation process, the system adjusts the vehicle's speed, steering angle, and linear/angular velocities according to the required action.

5.2.4 Intersection Detection and Navigation

The TurtleBot3 detects intersections by subscribing to traffic sign data and determining the direction to take at the intersection based on the received traffic sign message. Upon receiving an intersection order, the robot executes a series of movements to navigate through the intersection, adjusting its speed, steering angle, and linear/angular velocities as needed. The system provides control and feedback signals throughout the process.

5.2.5 Parking Detection and Navigation

The TurtleBot3 is designed to detect and navigate parking spaces by identifying empty slots, aligning itself, and executing a parking maneuver. The process is initiated upon detecting a parking sign, which triggers the parking mission. The robot employs its LiDAR sensor to detect obstacles within a specific range and angle cantered around the front of the vehicle, enabling it to locate an empty parking space. Utilizing a series of timed maneuvers, the TurtleBot3 parks in the designated spot and subsequently drive back onto the road, resuming lane tracking.

5.2.6 SLAM-based Navigation in Tunnel Sections

Turtlebot3 will switch to using only SLAM for the tunnel section, as explained in Chapter 4. After detecting the sign, it drives a set amount straight forwards into the tunnel before it stops and opens a pre-generated map located at **"/turtlebot3_autorace/turtlebot3_autorace_driving/maps/"** before navigating to the set goal at the exit of the tunnel.

5.3 Traffic Sign Detection

One of the most essential detection tasks is traffic sign detection, as this algorithm is responsible for initializing each of the missions described earlier. TurtleBot3 does this by using the Scale-Invariant Feature Transform (SIFT) algorithm, which is a detection algorithm that robust against scale, rotation, and illumination changes. SIFT works by identifying unique keypoints, also referred to as features or interest points, and their associated feature descriptors in an image. The keypoints are compared to a database of pre-existing traffic sign images, shown in Figure 5.5, for the simulated traffic signs, allowing the robot to recognize the traffic signs, as demonstrated in Figure 5.6.

The SIFT algorithm operation can be broken down into five main steps.

1. **Scale-space Extrema Detection:** The first step of the SIFT algorithm involves the detection of potential keypoints across different scales of the image. The goal is to identify points that are invariant to scale changes. To achieve this, the algorithm employs a Difference of Gaussians (DoG) approach, which approximates the Laplacian of Gaussian (LoG) method used in blob detection. The DoG is calculated at various scales (σ values) of the image, and local maxima across scale and space are identified as potential keypoints.

2. **Keypoint Localization:** After detecting potential keypoints, they are refined for more accurate results. This step involves eliminating points with low contrast and edge points, which could negatively impact the algorithm's performance. For this, the algorithm uses a Taylor series expansion of scale space to refine the location of extrema, and a 2x2 Hessian matrix to compute the principal curvature, similar to the Harris corner detector.
3. **Orientation Assignment:** An orientation is assigned to each keypoint to achieve rotation invariance. The algorithm calculates the gradient magnitude and direction within a neighborhood around the keypoint, and creates an orientation histogram. The peaks in this histogram correspond to the dominant orientations.
4. **Keypoint Descriptor:** The next step is the creation of the keypoint descriptor, which provides a unique fingerprint for each keypoint. This involves taking a 16x16 neighborhood around the keypoint and dividing it into 16 sub-blocks of 4x4 size. For each sub-block, an 8-bin orientation histogram is created. This generates a 128-bin vector, which forms the keypoint descriptor.
5. **Keypoint Matching:** The final step in the SIFT algorithm is keypoint matching. This is done by identifying the nearest neighbors of the keypoints in two images. The algorithm uses a ratio test to eliminate false matches, thus increasing the robustness of the matching process.

In the context of the TurtleBot3, the SIFT algorithm allows the robot to reliably detect and recognize signs in its environment, despite variations in scale, rotation, and lighting conditions. It's important to note that the performance of the SIFT algorithm heavily depends on the quality and diversity of the pre-trained sign images in the database.

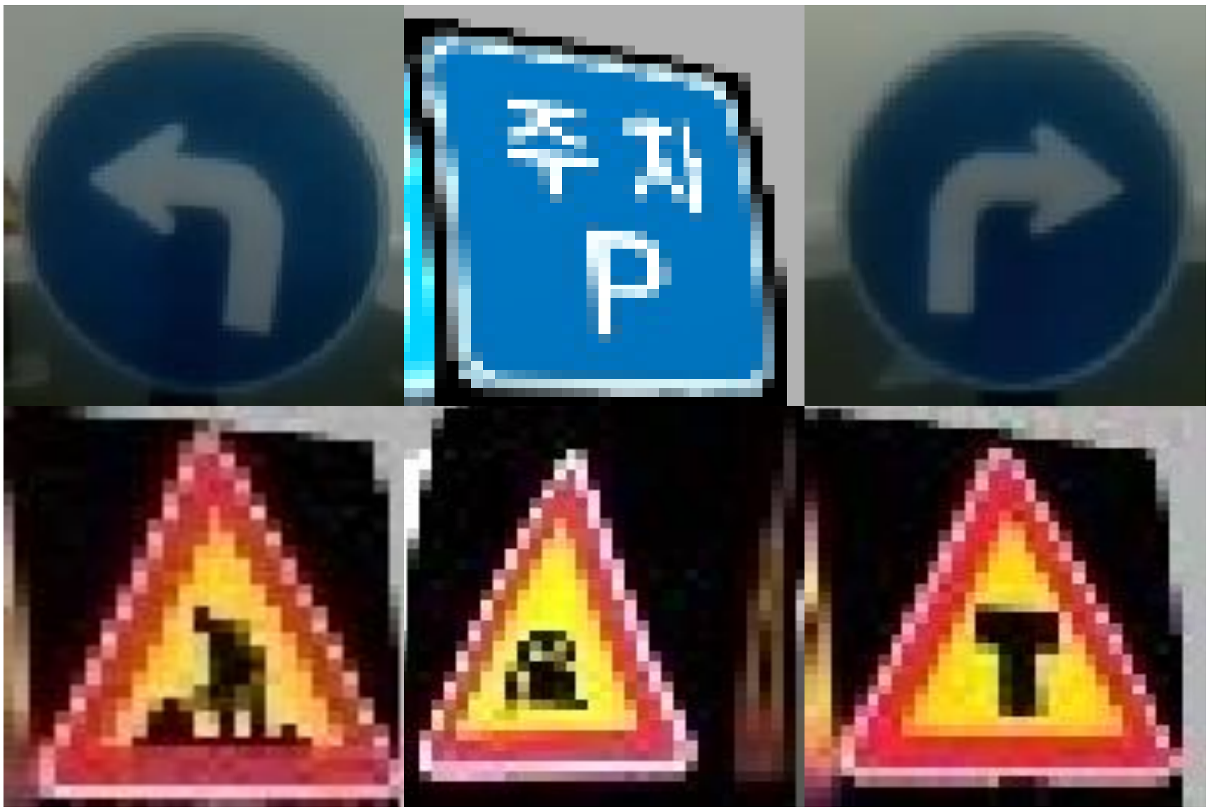
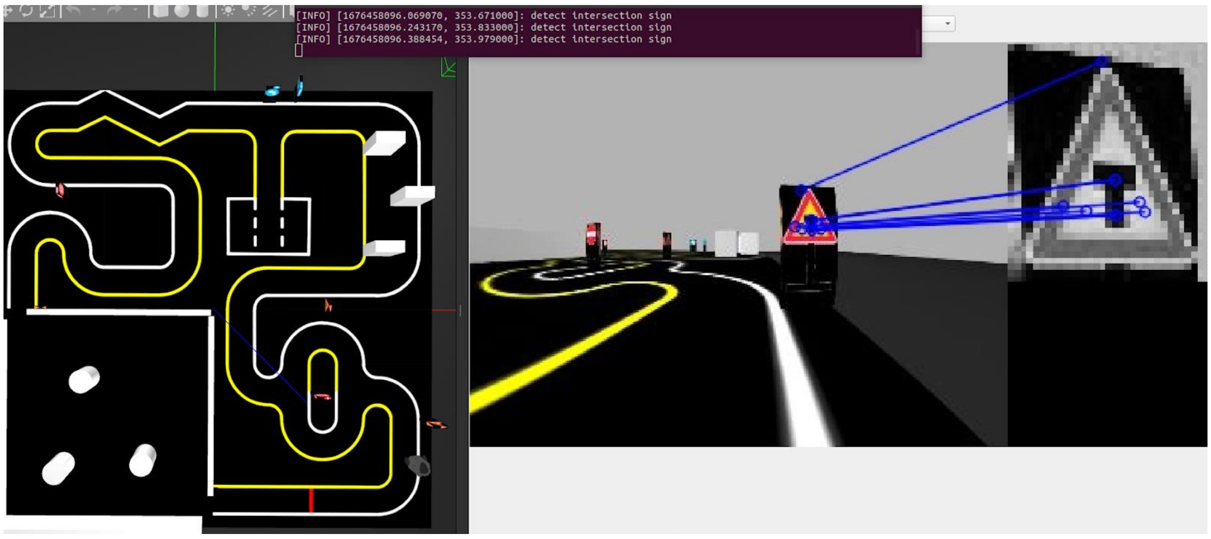


Figure 5.5: Images used for traffic sign detection in simulated environment.



```
[INFO] [1676458096.069070, 353.671000]: detect intersection sign
[INFO] [1676458096.243170, 353.833000]: detect intersection sign
[INFO] [1676458096.388454, 353.979000]: detect intersection sign
```

Figure 5.6: Sign detection in Gazebo

5.4 Integration with TurtleBot3 Navigation Stack

The integration of detection algorithms within the TurtleBot3 navigation stack is achieved using the Robot Operating System (ROS) framework, which ensures seamless communication and coordination between different components of the autonomous navigation system. Detection algorithms are incorporated as dedicated ROS nodes that subscribe to the camera's image topic and generate custom messages containing detected features, such as position, orientation, and status (e.g., traffic light color). These custom messages are then published to specific topics for further processing by other nodes in the navigation stack. A decision-making node subscribes to the topics containing detected features, processes the information, and determines the appropriate action for the robot, such as commanding the robot to stop or slow down based on detected traffic light color or adjusting its trajectory based on identified lanes. Control commands are generated based on the processed information and published to the appropriate topics for execution by TurtleBot3.

6 Autonomous Driving in Simulated and Real Environments

This chapter discusses the experimental setup and results of testing autonomous navigation algorithms in both simulated and real environments, highlighting the system's performance and potential areas for improvement.

6.1 Experimental Setup for Navigation

The experimental setup for testing autonomous navigation algorithms involves both simulated and real environments, aiming to evaluate the TurtleBot3's performance and robustness under different conditions.

Two courses are used in the simulated environment: AutoRace 2019 with Kinetic and AutoRace 2020 with Noetic. Both courses are similar, but they have some differences. AutoRace 2019 was chosen due to the parking obstacle's resemblance to the parking spaces on the actual racing rig. However, it lacks the intersection and construction obstacles present in AutoRace 2020. Both simulated versions also include an additional level crossing obstacle, which functions correctly but is not the main focus.

As explained in Chapter 2.5, the real racing rig is designed to simulate the actual conditions and challenges an autonomous vehicle faces. The rig includes a traffic light, an intersection with signs indicating left or right turns, a parking area with two slots where one is occupied by another TurtleBot3, a construction work zone with three obstacles to avoid, and a dark tunnel with obstacles inside. The TurtleBot3 primarily uses its camera for lane detection and other detection algorithms to navigate the racing rig. However, it also utilizes LiDAR for some obstacles (parking and construction). In the tunnel section, the TurtleBot3 relies solely on LiDAR with SLAM and a pre-generated map for navigation.

6.2 Simulated Environment Results

In the simulated environment, the autonomous driving system demonstrated promising results overall, with the robot successfully navigating the track and detecting traffic signs. This section presents the findings from the simulated environment tests and provides a concise summary of the outcomes.

6.2.1 Corner Cutting

The first problem that occurs is when following preforming lane detection and tracking. The robot drives as it should on the straight parts but cuts corners in the turns. The TurtleBot3 started the turns too early, causing it to move outside the lane boundaries in the corners, which could compromise its overall performance in navigating complex environments. This issue was observed consistently throughout the tests.

6.2.2 Parking Obstacle Performance

The robot's parking performance in the simulated environment was inconsistent, as it sometimes missed the first parking space, parking on the line or outside the line. This issue occasionally occurred throughout the tests.

6.2.3 Tunnel Navigation and SLAM

In the tunnel navigation and SLAM tasks, the robot faced difficulties generating a path to the goal and stopped. Out of 10 tests conducted in the Kinetic version of the simulated environment, the robot successfully completed the tunnel mission in 5 instances. The remaining five tests failed due to the robot's inability to generate a path through the tunnel obstacle.

6.3 Real Environment Results

The real-world testing of the autonomous driving system revealed some challenges, particularly in the lane and sign detection tasks. This section presents the results from the real-world tests and summarizes the findings.

6.3.1 Lane Detection

In the real environment, the robot used a standard Raspberry Pi camera without a fisheye lens, which limited its field of view. Consequently, the robot could only see the road in front of it 35 cm ahead. The visibility slightly improved when the camera was moved further down on the robot, reducing the blind zone to 15 cm in front of it. However, the robot still struggled to follow the lanes when the roads turned, mainly when approaching corners and losing sight of the road ahead.

6.3.2 Sign Detection

In the real-world tests, the sign detection system demonstrated mixed results. The robot recognized the turn left and turn right signs with the European traffic signs, as seen in Figure 6.1, but struggled to detect other signs. Detection improved slightly when using the Korean traffic signs the robot was trained on, but only under specific lighting conditions, angles, and a black background. Additionally, the system frequently generated false detections of the wrong traffic sign, as seen in Figure 6.2, and even when no traffic signs were present.

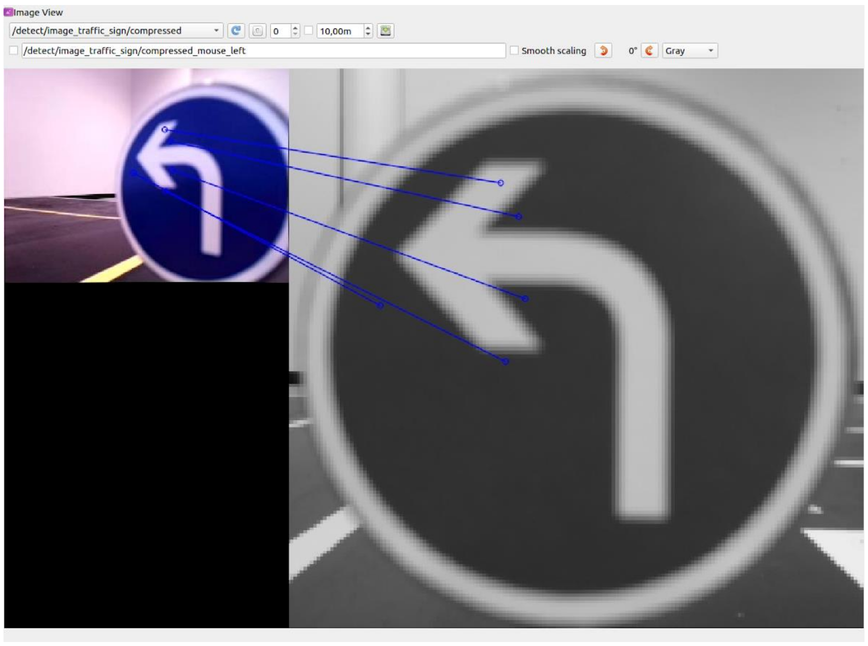


Image View
/detect/image_traffic_sign/compressed
0
10,00m
/detect/image_traffic_sign/compressed_mouse_left
Smooth scaling
0°
Gray

Figure 6.1: Successful sign detection.

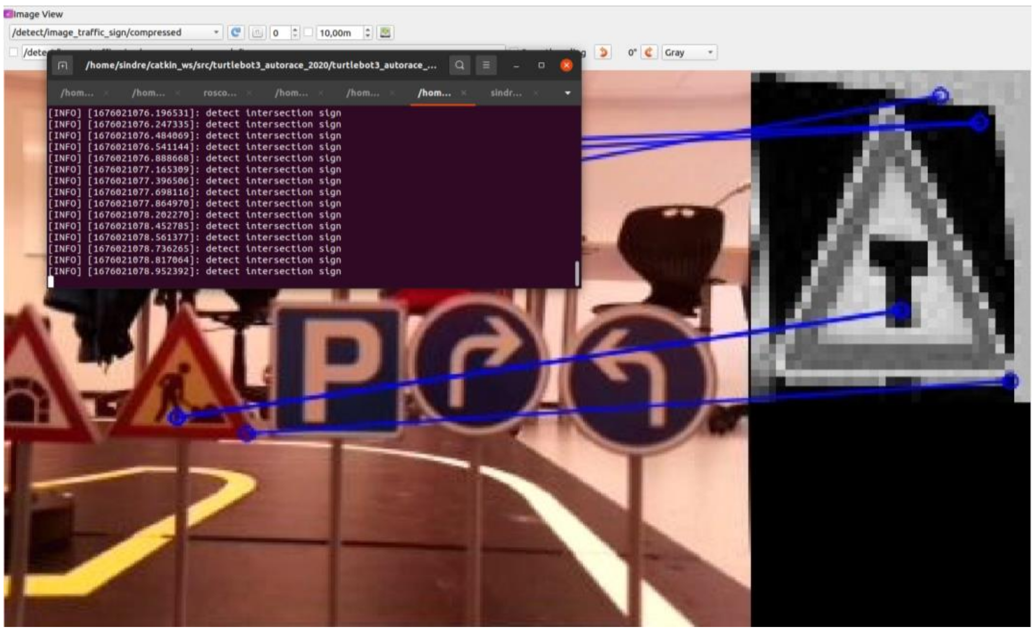


Image View
/detect/image_traffic_sign/compressed
0
10,00m
Gray
/home/sindre/catkin_ws/src/turtlebot3_autorace_2020/turtlebot3_autorace_...
/hom...
rosco...
sindr...

```
[INFO] [1676021076.196531]: detect intersection sign
[INFO] [1676021076.247335]: detect intersection sign
[INFO] [1676021076.484069]: detect intersection sign
[INFO] [1676021076.541144]: detect intersection sign
[INFO] [1676021076.888668]: detect intersection sign
[INFO] [1676021077.165309]: detect intersection sign
[INFO] [1676021077.396506]: detect intersection sign
[INFO] [1676021077.698116]: detect intersection sign
[INFO] [1676021077.864970]: detect intersection sign
[INFO] [1676021078.202270]: detect intersection sign
[INFO] [1676021078.452785]: detect intersection sign
[INFO] [1676021078.561377]: detect intersection sign
[INFO] [1676021078.736265]: detect intersection sign
[INFO] [1676021078.817064]: detect intersection sign
[INFO] [1676021078.952392]: detect intersection sign
```

Figure 6.2: False detection of intersection sign

7 Reinforcement Learning for Obstacle Avoidance

This chapter focuses on using reinforcement learning (RL) and Deep Q-Networks (DQN) to enhance the autonomous driving capabilities of the TurtleBot3.

7.1 Introduction to Reinforcement Learning and Deep Q-Networks

Reinforcement learning (RL) is a subfield of machine learning that focuses on training a model to make decisions based on interactions with its environment. The model tries to find the optimal actions to maximize the reward over time. Q-learning is a popular RL algorithm that learns the optimal action-value function (Q-value) to find the best move in a given state. Deep Q-Network (DQN) extends Q-learning by using a deep neural network as a function approximator for the Q-value. The function approximator helps to generalize learning across different states, enabling the agent to learn effectively in high-dimensional state spaces. DQN is a suitable RL algorithm for challenging tasks such as autonomous driving, as it allows the agent to understand complex and high-dimensional state spaces.

7.2 The Learning Process

The learning process for the machine learning model is done in Gazebo to speed up and simplify the process. As with the real robot, the simulated robot uses 360 Lidar samples to observe its environment, which helps it build a detailed state representation of its surroundings. The DQN utilizes a deep neural network with several layers, including input, hidden, and output layers, as shown in Figure 7.1. The input states for the DQN model include the 360 Lidar values, distance to the goal, and angle to the goal. There are four hidden layers with three dense layers and one dropout layer. The model gives five outputs, as listed below:

1. Move forward: The robot moves straight ahead at a constant speed.
2. Turn left: The robot rotates counterclockwise in place, changing its orientation.
3. Turn right: The robot rotates clockwise in place, changing its orientation.
4. Move forward and left: The robot moves forward while turning left, following a curved path to the left.
5. Move forward and right: The robot moves forward while turning right, following a curved path to the right.

The reward function encourages the agent to reach the goal while avoiding obstacles. The agent receives positive rewards when moving closer to the destination and negative rewards when moving away from it. Additionally, the agent receives a significant positive reward upon reaching the goal and a substantial negative reward when colliding with an obstacle.

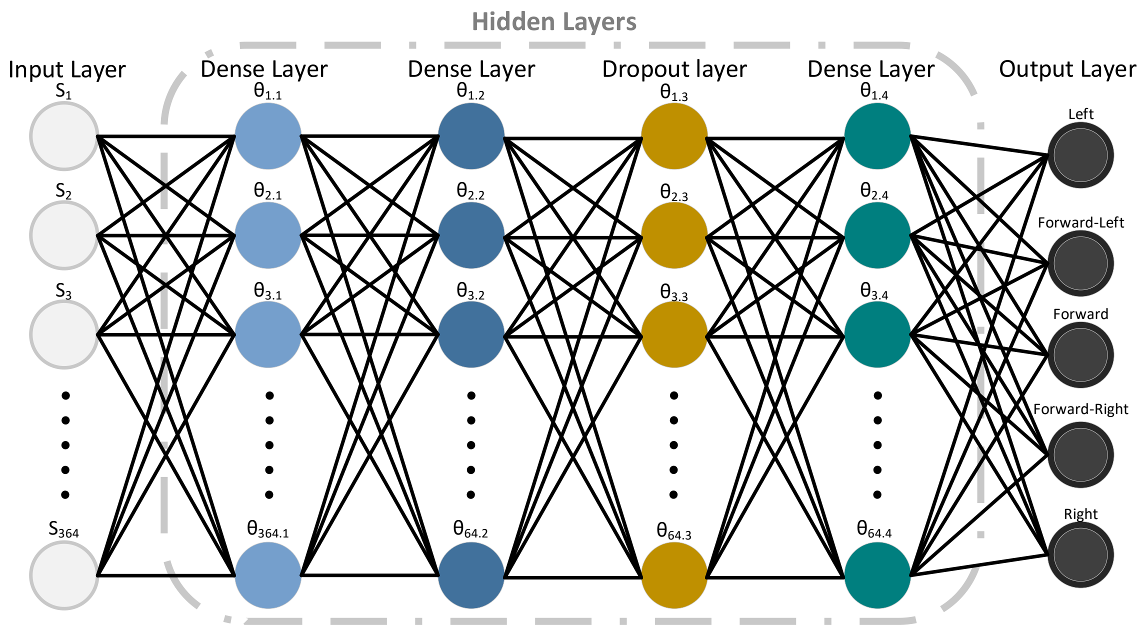


Figure 7.1: Deep Q-Network (DQN) architecture adapted for the TurtleBot3.

7.3 Results and analysis of reinforcement learning

The reinforcement learning algorithm was assessed in four stages of training, as depicted in Figure 7.2. Each stage presented a more challenging scenario, making the model adapt its behavior. Unfortunately, the graphs for stages 2, 3, and 4 were lost. However, they were largely similar to stage 1, with the primary difference being an increased number of iterations. This section provides an in-depth analysis of Stage 1, with both Total Reward and Average Max Q-value, as shown in Figure 7.3.

Stage 1 is a 4x4 map enclosed in a box without any obstacles. It took the model 150 iterations before it could consistently reach its target in this stage. Initially, the robot had no prior information, resulting in a low total reward during early episodes and difficulty reaching its target. As the model learned from its experiences, the total reward, which is the cumulative sum of rewards received for taking actions in each episode, increased as the robot got better at reaching its goal. Once the total reward reached a high value, the robot could navigate to its goal consistently, albeit not through the most efficient path.

The Average Max Q-value for Stage 1 also demonstrated the model's progress in understanding optimal actions. The initial fluctuations in the Average Max Q-value signified the model's uncertainty. The Average Max Q-value, representing the highest Q-value for each state-action pair averaged over all episodes, stabilized and gradually increased as the model learned. This increase indicates a more precise estimation of optimal actions and continuous refinement of the decision-making process.

Stages 2 and 3 featured a 4x4 map, with four static cylindrical obstacles for stage 2 and four moving cylindrical obstacles for stage 3. Stage 2 required 600 iterations, and stage 3 required 1000 iterations to consistently reach its target while avoiding the obstacles. In both stages, the total reward and Average Max Q-value graphs were mostly similar to stage 1, starting low and unstable before stabilizing at a positive number towards the end.

Stage 4 consisted of a larger 5x5 map with walls and two moving cylindrical obstacles. This stage was the most different as the total reward fluctuated between positive and negative values. At episode 1100, the model started showing promising results and sometimes managed to reach its goal without crashing. However, after some more training, it got stuck and started driving in a circle, focusing more on not crashing than reaching its goal. When no progress was made, the training was restarted from episode 1100 with an increased learning rate to help the model learn faster and a higher epsilon decay to improve the model's convergence. The model was then trained until episode 3000, before it consistently reached its target.

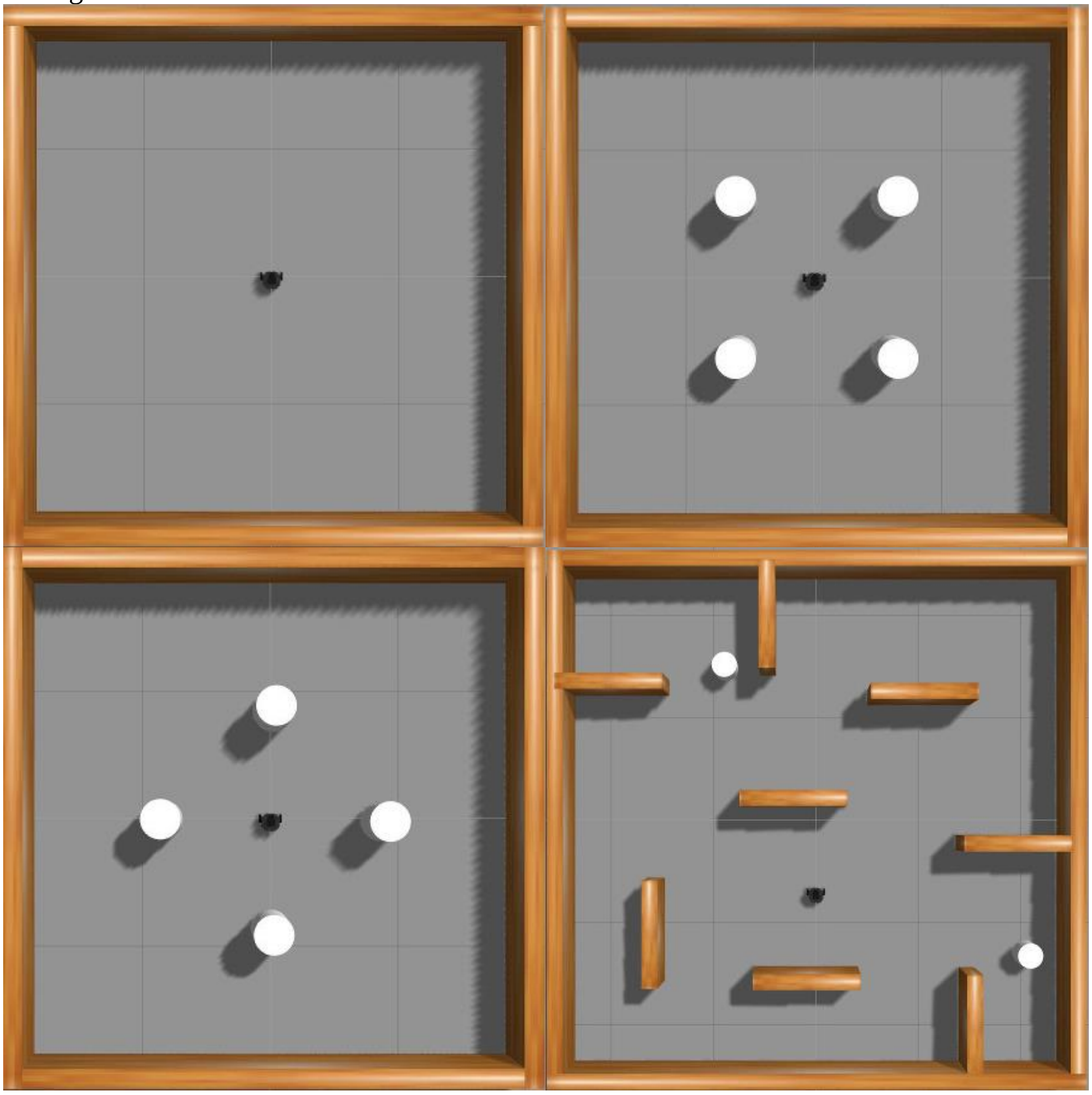


Figure 7.2: 4 stages for training the machine learning model.

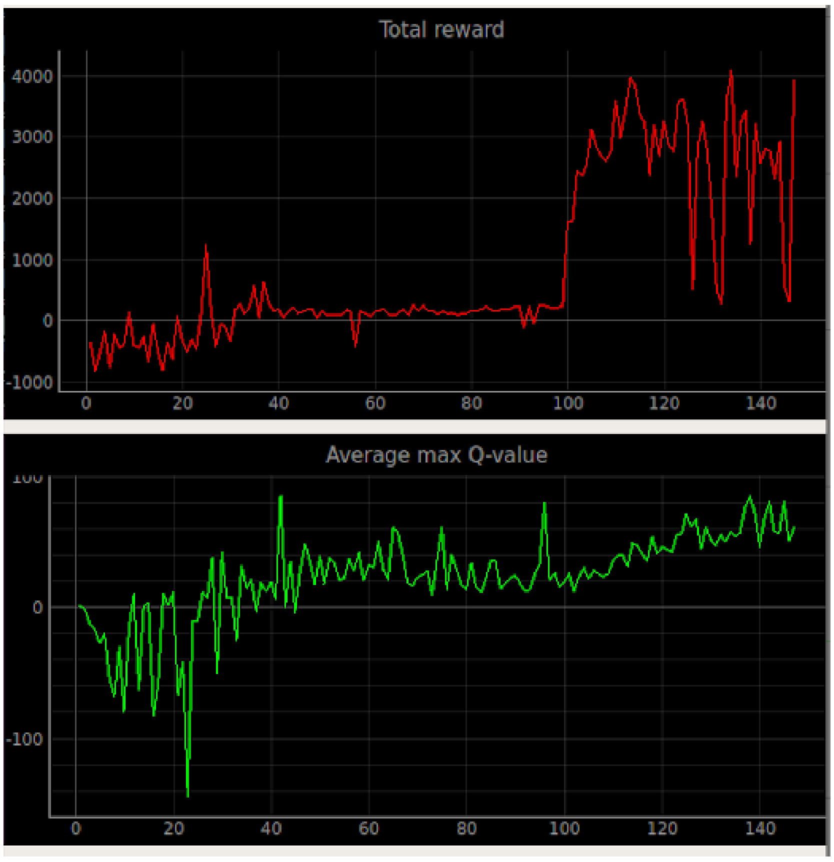


Figure 7.3: Total Reward and Average Max Q-value over time for stage 1

7.4 Future work and implementation on TurtleBot3

The successful simulation of the reinforcement learning algorithm in Gazebo paves the way for implementing the trained model on the actual TurtleBot3. Future work will focus on applying the behaviors learned in the simulated environment to the physical robot, ensuring its robustness in real-world environments. Moreover, the model can be further optimized by exploring alternative reward functions, improving the neural network architecture, or employing other RL algorithms to enhance the autonomous driving capabilities of the TurtleBot3.

8 Discussion

This chapter discusses the challenges, limitations, and implications of the autonomous driving system developed for the TurtleBot3 platform.

8.1 Setup, Configuration, and Navigation

The TurtleBot3 was successfully set up and configured for two different models, the TurtleBot3 Burger and the Custom TurtleBot3. Initially, the Custom TurtleBot3, which was designed with different dimensions than the Burger model, faced challenges in stopping accurately at target points while navigating. The robot would spin around indefinitely, trying to reach the exact target position. This issue was resolved by adjusting the xy and yaw goal tolerances to higher values, allowing the robot to stop further from the target without continually attempting to reach the precise location.

SLAM and navigation worked effectively for the most part. The system was fine-tuned to allow the robot to navigate closer to objects and through small openings, such as tunnel entrances and exits. However, improper parameters led the robot to stop or oscillate while searching for a path. To overcome this issue, the costmap parameters were adjusted by setting the inflation_radius to 0.2 and cost_scaling_factor to 3. These changes enabled the robot to approach obstacles more closely without colliding. Overall, the project's setup, configuration, and navigation aspects were successful, providing a solid foundation for the autonomous driving system.

8.2 Camera Selection and Positioning

The choice of camera and its positioning proved to be a challenge during the project. The standard Raspberry Pi camera without a fisheye lens was insufficient for effective lane tracking, particularly during turns, as the limited field of view restricted the robot's visibility of the road. Although lowering the camera position improved results somewhat, it was still not enough to enable the robot to perform lane tracking successfully around the entire track.

In retrospect, a Raspberry Pi camera with a fisheye lens would have been more suitable for the project, as it would have provided a 180-degree field of view and facilitated more effective lane tracking. Future research should investigate the impact of different camera types and mounting positions on the system's performance.

8.3 Traffic Sign Detection

Traffic sign detection faced difficulties in the real-world environment, as the robot struggled to detect all traffic signs. Detection improved slightly when using the Korean traffic signs the robot was trained on, but only under specific lighting conditions, angles, and with a black background. Additionally, the system frequently generated false detections of the intersection sign even when no signs were present. Attempts to resolve this problem by updating the reference images in the detection folder with new photos of both the 3D-printed European-standard signs and the Korean signs yielded minimal improvements. This highlights the need for a more adaptable algorithm to manage various real-world conditions and diverse training data to ensure reliable and accurate sign detection across different regions.

One possible reason for the detection issues could be that the SIFT algorithm, which was used for traffic sign detection, may have been trained using a fisheye lens. If this were the case, it would explain why the robot struggled to detect signs accurately, as the standard Raspberry Pi camera without a fisheye lens would capture images with different perspectives and distortions compared to the training data. The discrepancy between the training data and the real-world images captured by the robot's camera could significantly affect the sign detection performance, emphasizing the importance of camera selection and compatibility with the chosen algorithm for successful traffic sign detection.

Future work should explore alternative algorithms that are more robust and adaptable to various conditions, such as deep learning-based approaches that have shown promise in object recognition tasks. Expanding the training dataset to include a broader range of traffic signs from different regions and under varying conditions would likely improve the system's ability to detect signs accurately in real-world environments. Addressing these limitations will have important implications for the development of autonomous driving systems, emphasizing the significance of camera selection, algorithm adaptability, and diverse training data in achieving reliable performance.

8.4 Simulated Environment

In the Kinetic version with Autorace 2019, the ground_plane was missing, causing the robot to fall through the environment. Manually installing the ground_plane from GIT resolved this issue. In the simulated environment, the autonomous driving system demonstrated promising results, as the robot successfully navigated the track and detected traffic signs.

Another issue encountered in the simulation was that the TurtleBot3 sometimes failed to generate a path to the goal when faced with the tunnel obstacle. A quick fix for this during the simulation, without addressing the root cause, was to manually set the navigation goal again. The cause of this problem is not entirely clear, but one possibility could be limitations in the path planning algorithm, which may struggle to find a valid path in certain complex scenarios.

9 Conclusion

The purpose of this project was to investigate and develop an autonomous driving system for the Turtlebot3 platform, with the goal of creating a research and education platform using Robot Operating System (ROS). This included installation and testing of different ROS packages, completing the racing rig, developing and evaluating autonomous driving in both real and simulated environment, and exploring the use of reinforcement learning for obstacle avoidance. The findings from this project show the challenges in transitioning from simulated to real-world scenarios, such as choosing the correct camera and detection algorithms.

Future work for this project includes testing with a fisheye lens camera, exploring alternative traffic sign detection algorithms, and integrating aspects from the reinforcement learning section to enhance overall performance.

References

Durrant-Whyte, H. &. (2006). *Simultaneous localization and mapping: part I.* IEEE Robotics & Automation Magazine.

Educative. (2023). *educative.* Hentet fra What is the A* algorithm?: https://www.educative.io/answers/what-is-the-a-star-algorithm

Fox, D., Burgard, W., & Thrun, S. (1997). *The dynamic window approach to collision avoidance.* IEEE Robotics & Automation Magazine.

Granosik, G. &. (2016). *USING ROBOT OPERATING SYSTEM FOR AUTONOMOUSCONTROL OF ROBOTS IN EUROBOT, ERC AND ROBOTOURCOMPETITIONS.* Acta Polytechnica CTU Proceedings.

Haugseter, S., & Lauritzen, C. (2022). *Setup, configuration and simulation of Turtlebot3 Waffle Pi mobile rover for autonomous driving.* Porsgrunn.

Lauttia, T. (2022). *Adaptive Monte Carlo Localization in ROS.* Faculty of Information Technology and Communication Sciences.

Lin, S.-Y., & Chen, Y.-C. (2011). *SLAM and navigation in indoor environments*. Hentet 4 6, 2023 fra https://link.springer.com/content/pdf/10.1007/978-3-642-25367-6_5.pdf

Open Robotics. (2022). *ROS*. Hentet fra http://wiki.ros.org/

ROBOTIS. (2022). Hentet fra Hardware Assembly: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/#overview

ROS. (2023). *Setup the Navigation Stack for TurtleBot.* Hentet November 18, 2022 fra https://wiki.ros.org/turtlebot_navigation/Tutorials/Setup%20the%20Navigation%20Stack%20for%20TurtleBot

Šarić, R. (2018). *thingiverse*. Hentet fra https://www.thingiverse.com/thing:3010541

SCHAETZEN, R. D. (2019). Hentet fra Configuring the ROS Navigation Stack on a new robot: https://blog.zhaw.ch/icclab/configuring-the-ros-navigation-stack-on-a-new-robot/

Wang, H. H. (2020). *A Quantitative Analysis on Gmapping Algorithm Parameters Based on Lidar in Small Area Environment.* Chinese Intelligent Systems Conference.

Zhang, Z. (2016). *Camera Parameters (Intrinsic, Extrinsic).* Boston: Springer.

Appendices

Appendix A: FMH606 Master's Thesis

University of South-Eastern Norway

Faculty of Technology, Natural Sciences and Maritime Sciences, Campus Porsgrunn

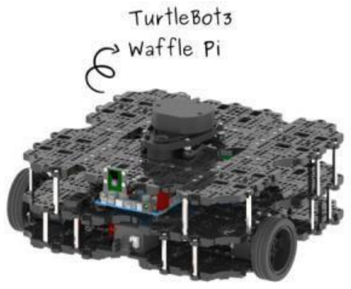
Title: Autonomous driving and machine learning with TurtleBot3 Waffle Pi mobile rover.

USN supervisor: Associate Professor Roshan Sharma (USN), Senior Engineer Fredrik Hansen

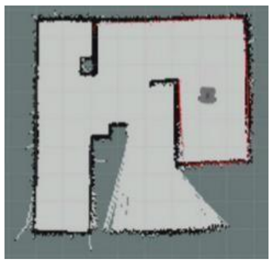
(USN)

Task background:

TutleBot3 Waffel Pi is a programmable mobile rover that is based on ROS (Robot Operating System) and has been widely used in research and education in many universities all around the world. A schematic of TurtleBot3 Waffle Pi is shown in Figure 1(a). It has two wheels for driving around together with a Raspberry Pi camera at the front for acquiring images and videos. But what makes this mobile rover interesting is that, it is also equipped with 360 degree distance sensor which is a LIDAR sensor (a rotating laser sensor for 3D scanning). This makes it useful for generating or building maps of the local area where the rover is moving around by making use of algorithms such as SLAM (simultaneous localization and mapping). An example of such a map is shown in Figure 1(b). Generation of local maps is a crucial step required for autonomous driving. USN has recently purchased several units of a turtlebot 3 waffle pi mobile rovers. These are planned to be used in teaching and research activities here at USN. The rover is driven by the Robotis DYNAMIXEL smart servo motors which are highly customizable.



(a)



(b)

Figure 1: (a) A turtlebot3 waffel pi (b) Example of a map of a room creating using SLAM.

Aim:

It is of interest to use this mobile rover with ROS platform for creating an autonomous driving vehicle research/educational platform here at USN. To motivate you have a look at this youtube video at https://www.youtube.com/watch?v=O1mTL5VBfGs

Task description:

The following are the main tasks:

a) Installation of ROS in Linux platform, followed by installation of various ROS packages for turtlebot3 waffle pi rover.
b) Connect turtlebot3 waffle pi and test its basic functionality such as (i) being able to use SLAM for creating dynamic map of a local area and localize the position of the rover, (ii) command it to automatically navigate from any chosen point A to another point B within the local map, (ii) being able to acquire the camera image/video etc. The student should also try to change the values of the tuning parameters for these tasks and try to improve both the SLAM and navigation.
c) Another task will be to complete the construction of a racing rig (working support from USN staff) that is suitable for waffle pi. The racing rig should have (i) racing track (straight and curves) with start and finish markings, (ii) parking area, (iii) traffic light (iv) traffic signs with arrows showing direction etc. This task will also require the student to 3D print different elements used in the racing track.
d) The most important task would be to use the racing rig for testing the turtlebot 3 waffle pi rover for autonomous driving both in simulator and with real device. For this, the rover should follow the rules setup for autonomous driving like (i) drive along the road (ii) stop at red light and continue at green light, (iii) read the traffic sign and decide which direction to drive, (iv) park at the right spot, (v) navigate through the tunnel etc.
e) Using tools like Tensorflow, Keras, Python etc., use reinforcement learning for obstacle avoidance during navigation. For this task, Gazebo and ROS can be used as simulation platform.
f) Document the work in a report. Presentation of the work.

Student category: Reserved

Reserved for IIA campus student Sindre Haugseter.

Practical arrangements:

Turtlebot3 waffle pi will be provided to the students. In addition materials needed for construction of the racing rig will also be available to the student. The workplace is campus Porsgrunn.

Signatures:

Supervisor (date and signature): [illegible]
01.02.2023

Student (write clearly in all capitalized letters): SINDRE HAUGSETER

Students (date and signature): Sindre Haugseter
01.02.2023

Appendix B: Arduino Traffic Light Code

```
int GREEN = 4;
int YELLOW = 3;
int RED = 2;
int DELAY_GREEN = 8000;
int DELAY_YELLOW = 2000;
int DELAY_RED = 8000;

void setup() {
  pinMode(GREEN, OUTPUT);
  pinMode(YELLOW, OUTPUT);
  pinMode(RED, OUTPUT);
}
void loop() {
  green_light();
  delay(DELAY_GREEN);
  yellow_light();
  delay(DELAY_YELLOW);
  red_light();
  delay(DELAY_RED);
  yellow_red_light();
  delay(DELAY_YELLOW);
}
void green_light()
{
  digitalWrite(GREEN, HIGH);
  digitalWrite(YELLOW, LOW);
  digitalWrite(RED, LOW);
}
void yellow_light()
{
  digitalWrite(GREEN, LOW);
  digitalWrite(YELLOW, HIGH);
  digitalWrite(RED, LOW);
}
void yellow_red_light()
{
  digitalWrite(GREEN, LOW);
  digitalWrite(YELLOW, HIGH);
  digitalWrite(RED, HIGH);
}
void red_light()
{
  digitalWrite(GREEN, LOW);
  digitalWrite(YELLOW, LOW);
  digitalWrite(RED, HIGH);
}
```

Appendix C: Start Guide for TurtleBot3 (Kinetic)

B.1 PC Setup:

1. Install Ubuntu 16.04 LTS: Download Ubuntu 16.04 LTS Desktop image (64-bit) from https://releases.ubuntu.com/16.04.7/. Create a bootable USB drive and follow the installation process. You can choose to dual boot your PC with Ubuntu 16.04 and another operating system if needed.
2. Install ROS Kinetic and dependent ROS packages:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ wget https://raw.githubusercontent.com/ROBOTIS-
GIT/robotis_tools/master/install_ros_kinetic.sh
$ chmod 755 ./install_ros_kinetic.sh
$ bash ./install_ros_kinetic.sh
$ sudo apt-get install ros-kinetic-joy ros-kinetic-teleop-twist-joy \
  ros-kinetic-teleop-twist-keyboard ros-kinetic-laser-proc \
  ros-kinetic-rgbd-launch ros-kinetic-depthimage-to-laserscan \
  ros-kinetic-rosserial-arduino ros-kinetic-rosserial-python \
  ros-kinetic-rosserial-server ros-kinetic-rosserial-client \
  ros-kinetic-rosserial-msgs ros-kinetic-amcl ros-kinetic-map-server\
  ros-kinetic-move-base ros-kinetic-urdf ros-kinetic-xacro \
  ros-kinetic-compressed-image-transport ros-kinetic-rqt* \
  ros-kinetic-gmapping ros-kinetic-navigation ros-kinetic-
interactive-markers
```

3. Install TurtleBot3 Packages:

```
$ sudo apt-get install ros-kinetic-dynamixel-sdk
$ sudo apt-get install ros-kinetic-turtlebot3-msgs
$ sudo apt-get install ros-kinetic-turtlebot3
```

4. Set the default TURTLEBOT3_MODEL:

```
$ echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc
```

5. Network Configuration: Connect your PC to a Wi-Fi device and find the assigned IP address using the command:

```
$ ifconfig
```

6. Edit the ROS IP settings in the ~/.bashrc file:

```
$ nano ~/.bashrc
```

7. Modify the address of localhost in the ROS_MASTER_URI and ROS_HOSTNAME with the IP address of the PC. Save the file (Ctrl+S) and exit (Ctrl+X). Source the bashrc with the command:

```
$ source ~/.bashrc
```

B.2 SBC Setup

1. Prepare a microSD card (minimum 8GB) and a card reader.
2. Download the Raspberry Pi 3B+ ROS Kinetic image from the following link: https://www.robotis.com/service/download.php?no=2066

3. After the download is complete, unzip the downloaded image file to extract the .img file.
4. Install Raspberry Pi Imager (https://www.raspberrypi.org/software/) on your computer if you haven't already. Open Raspberry Pi Imager, click on "Choose OS," and select "Use custom" to browse and select the .img file you extracted earlier. Insert your microSD card into the card reader and connect it to your computer. In Raspberry Pi Imager, click on "Choose SD Card," and select your microSD card. Click on "Write" to burn the image file onto the microSD card.
5. Boot up the Raspberry Pi:
   - Connect the HDMI cable of the monitor to the HDMI port of Raspberry Pi.
   - Connect input devices (keyboard and mouse) to the USB ports of Raspberry Pi.
   - Insert the microSD card into the Raspberry Pi.
   - Connect the power supply to turn on the Raspberry Pi.
6. Configure the Raspberry Pi:
   - After the Raspberry Pi has booted up, connect it to the same Wi-Fi network as your PC.
   - Find the IP address of the Raspberry Pi by running **$ ifconfig** in its terminal (usually under the wlan0 section).
   - Connect to the Raspberry Pi using the SSH command (replacing **10.3.6.10** with your IP address):

     ```
     $ ssh ubuntu@10.3.6.10
     ```
   - When prompted, enter the default password: **turtlebot**
7. Configure ROS Network:
   - Edit the .bashrc file on the Raspberry Pi by typing the following command:

     ```
     $ nano ~/.bashrc
     ```
   - Find the **ROS_MASTER_URI** and **ROS_HOSTNAME** settings section in the .bashrc file.
   - Modify the IP addresses for **ROS_MASTER_URI** and **ROS_HOSTNAME** and replace **{IP_ADDRESS_OF_REMOTE_PC}** with the IP address of your computer running Ubuntu 20.04 and **{IP_ADDRESS_OF_RASPBERRY_PI}** with the IP address you noted in step 7.

     ```
     export ROS_MASTER_URI=http://{IP_ADDRESS_OF_REMOTE_PC}:11311
     export ROS_HOSTNAME={IP_ADDRESS_OF_RASPBERRY_PI_3}
     ```
   - Save the file by pressing Ctrl+S and exit the nano editor with Ctrl+X.
   - Apply the changes by typing the following command:

     ```
     $ source ~/.bashrc
     ```

B.3 OpenCR Setup

1. Connect the OpenCR board to the Raspberry Pi using a micro USB cable.
2. Install the required packages on the Raspberry Pi to upload the OpenCR firmware by entering the following commands in the terminal:

```
$ sudo dpkg --add-architecture armhf
$ sudo apt-get update
$ sudo apt-get install libc6:armhf
```

3. Set the OpenCR model name and port for your TurtleBot3 platform (either "Burger " or "waffle ") by typing the following commands in the terminal:

```
$ export OPENCR_PORT=/dev/ttyACM0
$ export OPENCR_MODEL=burger
$ rm -rf ./opencr_update.tar.bz2
```

4. Download the firmware and loader by entering the following command:

```
$ wget https://github.com/ROBOTIS-GIT/OpenCR-
Binaries/raw/master/turtlebot3/ROS1/latest/opencr_update.tar.bz2
$ tar -xvf opencr_update.tar.bz2
```

5. Extract the downloaded file by typing:

```
$ tar -xvf opencr_update.tar.bz2
```

6. Navigate to the "opencr_update" directory and upload the firmware to the OpenCR board:

```
$ cd ./opencr_update
$ ./update.sh $OPENCR_PORT $OPENCR_MODEL.opencr
```

7. Test the OpenCR board and the robot assembly by doing the following:

   - Power on the OpenCR board by connecting the power and turning on the power switch. The red Power LED will turn on.
   - Place the robot on a flat surface with a safety radius of at least 1 meter (40 inches).
   - Press and hold PUSH SW1 for a few seconds to command the robot to move 30 centimeters (about 12 inches) forward.
   - Press and hold PUSH SW2 for a few seconds to command the robot to rotate 180 degrees in place.

Appendix D: Start Guide for TurtleBot3 (Noetic)

C.1 PC Setup:

1. Install Ubuntu 20.04 LTS: Download Ubuntu 20.04 LTS Desktop image (64-bit) from https://releases.ubuntu.com/20.04/. Create a bootable USB drive and follow the installation process. You can choose to dual boot your PC with Ubuntu 20.04 and another operating system if needed.
2. Install ROS Noetic and dependent ROS packages:

```
$ sudo apt update
$ sudo apt upgrade
$ wget https://raw.githubusercontent.com/ROBOTIS-
GIT/robotis_tools/master/install_ros_noetic.sh
$ chmod 755 ./install_ros_noetic.sh
$ bash ./install_ros_noetic.sh
$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
  ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
  ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
  ros-noetic-rosserial-python ros-noetic-rosserial-client \
  ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
  ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
  ros-noetic-compressed-image-transport ros-noetic-rqt* ros-noetic-
rviz \
  ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-
markers
```

3. Install TurtleBot3 Packages:

```
$ sudo apt install ros-noetic-dynamixel-sdk
$ sudo apt install ros-noetic-TurtleBot3-msgs
$ sudo apt install ros-noetic-TurtleBot3
```

4. Network Configuration: Connect your PC to a Wi-Fi device and find the assigned IP address using the command:

```
$ ifconfig
```

5. Edit the ROS IP settings in the ~/.bashrc file:

```
$ nano ~/.bashrc
```

6. Modify the address of localhost in the ROS_MASTER_URI and ROS_HOSTNAME with the IP address of the PC. Save the file (Ctrl+S) and exit (Ctrl+X). Source the bashrc with the command:

```
$ source ~/.bashrc
```

C.2 SBC Setup

1. Prepare a microSD card (minimum 8GB) and a card reader.
2. Download the Raspberry Pi 4B (2GB or 4GB) ROS Noetic image from the following link: https://www.robotis.com/service/download.php?no=2066
3. After the download is complete, unzip the downloaded image file to extract the .img file.

4. Install Raspberry Pi Imager (https://www.raspberrypi.org/software/) on your computer if you haven't already. Open Raspberry Pi Imager, click on "Choose OS," and select "Use custom" to browse and select the .img file you extracted earlier. Insert your microSD card into the card reader and connect it to your computer. In Raspberry Pi Imager, click on "Choose SD Card," and select your microSD card. Click on "Write" to burn the image file onto the microSD card.
5. Resize the partition with GParted:
   - Install GParted (https://gparted.org/download.php) on your computer if you haven't already.
   - Insert the microSD card with the burned image into the card reader and connect it to your computer.
   - Open GParted and select the microSD card from the top-right menu (the device name may vary depending on your system, e.g., /dev/sdb).
   - In the GParted window, right-click on the yellow partition (this should be the largest partition on the microSD card).
   - Select "Resize/Move" from the context menu.
   - In the "Resize/Move" window, drag the right edge of the partition all the way to the right end to maximize the available space.
   - Click the "Resize/Move" button to apply the changes.
   - Click the "Apply All Operations" green check button at the top of the GParted window to finalize the resizing process.'
6. Configure the Wi-Fi Network Setting with the microSD card still connected to the PC::
   - Open a terminal window with Alt+Ctrl+T
   - Navigate to the netplan directory on the microSD card and edit the 50-cloud-init.yaml file with superuser permission using the following commands:

```
$ cd /media/$USER/writable/etc/netplan
$ sudo nano 50-cloud-init.yaml
```

   - In the editor, replace the **WIFI_SSID** and **WIFI_PASSWORD** placeholders with your Wi-Fi SSID and password.
   - Save the file by pressing Ctrl+S and exit the nano editor with Ctrl+X.
   - Safely eject the microSD card from your PC.
7. Insert the microSD card into the Raspberry Pi and power it on. The Raspberry Pi should connect to the Wi-Fi network automatically. To find its IP address and connect using SSH, follow the steps below:
   - Use a tool like Angry IP Scanner (https://angryip.org/download/) to find the IP address of your Raspberry Pi.
   - Open a terminal window on your Ubuntu PC.
   - Connect to the Raspberry Pi using the SSH command (replacing **10.3.6.10** with your IP address):

```
$ ssh ubuntu@10.3.6.10
```

- When prompted, enter the default password: **turtlebot**

8. Configure ROS Network:
   - Edit the .bashrc file on the Raspberry Pi by typing the following command:
     ```
     $ nano ~/.bashrc
     ```
   - Find the **ROS_MASTER_URI** and **ROS_HOSTNAME** settings section in the .bashrc file.
   - Modify the IP addresses for **ROS_MASTER_URI** and **ROS_HOSTNAME** and replace **{IP_ADDRESS_OF_REMOTE_PC}** with the IP address of your computer running Ubuntu 20.04 and **{IP_ADDRESS_OF_RASPBERRY_PI}** with the IP address you noted in step 7.
     ```
     export ROS_MASTER_URI=http://{IP_ADDRESS_OF_REMOTE_PC}:11311
     export ROS_HOSTNAME={IP_ADDRESS_OF_RASPBERRY_PI_3}
     ```
   - Save the file by pressing Ctrl+S and exit the nano editor with Ctrl+X.
   - Apply the changes by typing the following command:
     ```
     $ source ~/.bashrc
     ```

C.3 OpenCR Setup

1. Connect the OpenCR board to the Raspberry Pi using a micro USB cable.
2. Install the required packages on the Raspberry Pi to upload the OpenCR firmware by entering the following commands in the terminal:
   ```
   $ sudo dpkg --add-architecture armhf
   $ sudo apt-get update
   $ sudo apt-get install libc6:armhf
   ```
3. Set the OpenCR model name and port for your TurtleBot3 platform (either "Burger_noetic" or "waffle_noetic") by typing the following commands in the terminal:
   ```
   $ export OPENCR_PORT=/dev/ttyACM0
   $ export OPENCR_MODEL=Burger_noetic
   $ rm -rf ./opencr_update.tar.bz2
   ```
4. Download the firmware and loader by entering the following command:
   ```
   $ wget https://github.com/ROBOTIS-GIT/OpenCR-
   Binaries/raw/master/TurtleBot3/ROS1/latest/opencr_update.tar.bz2
   ```
5. Extract the downloaded file by typing:
   ```
   $ tar -xvf opencr_update.tar.bz2
   ```
6. Navigate to the "opencr_update" directory and upload the firmware to the OpenCR board:
   ```
   $ cd ./opencr_update
   $ ./update.sh $OPENCR_PORT $OPENCR_MODEL.opencr
   ```
7. Test the OpenCR board and the robot assembly by doing the following:
   - Power on the OpenCR board by connecting the power and turning on the power switch. The red Power LED will turn on.
   - Place the robot on a flat surface with a safety radius of at least 1 meter (40 inches).
   - Press and hold PUSH SW1 for a few seconds to command the robot to move 30 centimeters (about 12 inches) forward.

- Press and hold PUSH SW2 for a few seconds to command the robot to rotate 180 degrees in place.

Appendix E: Installing Autorace Packages

This appendix provides a step-by-step guide for installing the required packages and dependencies for Autorace on the Remote PC and the TurtleBot3 Single Board Computer (SBC).

D.1 Install Autorace Packages on Remote PC

1. Install the AutoRace 2020 meta package on the Remote PC by cloning the repository from GitHub into the catkin workspace source directory and then build the package using catkin_make. This package contains the necessary files and modules to run the Autorace:

```
$ cd ~/catkin_ws/src/
$ git clone -b noetic-devel https://github.com/ROBOTIS-
GIT/TurtleBot3_autorace_2020.git
$ cd ~/catkin_ws && catkin_make
```

2. Install additional dependent packages on the Remote PC. These packages provide functionality such as image transport, OpenCV support, and image processing:

```
$ sudo apt install ros-noetic-image-transport ros-noetic-cv-bridge
ros-noetic-vision-opencv python3-opencv libopencv-dev ros-noetic-
image-proc
```

D.2 Install Autorace Packages on TurtleBot3

1. Install the AutoRace package on both the Remote PC and the SBC (Single Board Computer) by cloning the feature-raspicam branch from the GitHub repository into the catkin workspace source directory and then build the package using catkin_make. This package contains the necessary files and modules to run the Autorace with Raspberry Pi camera support:

```
$ cd ~/catkin_ws/src/
$ git clone -b feature-raspicam https://github.com/ROBOTIS-
GIT/TurtleBot3_autorace_2020.git
$ cd ~/catkin_ws && catkin_make
```

2. Create a swap file on the SBC to prevent memory issues when building OpenCV. Allocate space for the swap file, set the appropriate permissions, create a swap area, and then enable the swap file. The swap file helps to avoid running out of memory during the build process.

```
$ sudo fallocate -l 4G /swapfile
$ sudo chmod 600 /swapfile
$ sudo mkswap /swapfile
$ sudo swapon /swapfile
```

3. Install required dependencies on the SBC. These dependencies include build tools, image processing libraries, and support for various video codecs and formats, which are necessary for running the Autorace and processing images from the camera.

```
$ sudo apt-get update
$ sudo apt-get install build-essential cmake gcc g++ git unzip pkg-
config
$ sudo apt-get install libjpeg-dev libpng-dev libtiff-dev libavcodec-
dev libavformat-dev libswscale-dev libgtk2.0-dev libcanberra-gtk*
libxvidcore-dev libx264-dev python3-dev python3-numpy python3-pip
libtbb2 libtbb-dev libdc1394-22-dev libv4l-dev v4l-utils libopenblas-
dev libatlas-base-dev libblas-dev liblapack-dev gfortran libhdf5-dev
libprotobuf-dev libgoogle-glog-dev libgflags-dev protobuf-compiler
```

4. Download, build and install OpenCV on the SBC with the opencv_contrib modules. Configure the build process by creating a CMake file with the appropriate options, then compile OpenCV using the make command and install the compiled binaries using the make install command. Update the shared library cache with the ldconfig command and clean up the build files with the make clean command. OpenCV is essential for image processing tasks in the Autorace.

```
$ cd ~
$ wget -O opencv.zip
https://github.com/opencv/opencv/archive/4.5.0.zip
$ wget -O opencv_contrib.zip
https://github.com/opencv/opencv_contrib/archive/4.5.0.zip

$ unzip opencv.zip
$ unzip opencv_contrib.zip

$ mv opencv-4.5.0 opencv
$ mv opencv_contrib-4.5.0 opencv_contrib
```

5. Create a CMake file for OpenCV on the SBC. Configure the build process by setting various options related to the build type, installation path, extra modules path, and others. This step is crucial for building OpenCV with the desired features and optimizations:

```
$ cd opencv
 mkdir build
$ cd build
$ cmake -D CMAKE_BUILD_TYPE=RELEASE \
        -D CMAKE_INSTALL_PREFIX=/usr/local \
        -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
        -D ENABLE_NEON=ON \
        -D BUILD_TIFF=ON \
        -D WITH_FFMPEG=ON \
        -D WITH_GSTREAMER=ON \
        -D WITH_TBB=ON \
        -D BUILD_TBB=ON \
        -D BUILD_TESTS=OFF \
        -D WITH_EIGEN=OFF \
        -D WITH_V4L=ON \
        -D WITH_LIBV4L=ON \
        -D WITH_VTK=OFF \
        -D OPENCV_ENABLE_NONFREE=ON \
        -D INSTALL_C_EXAMPLES=OFF \
        -D INSTALL_PYTHON_EXAMPLES=OFF \
        -D BUILD_NEW_PYTHON_SUPPORT=ON \
        -D BUILD_opencv_python3=TRUE \
        -D OPENCV_GENERATE_PKGCONFIG=ON \
        -D BUILD_EXAMPLES=OFF ..
```

6. Build and install OpenCV on the SBC. This process will take an hour or two to complete. After building, install the compiled binaries using the make install command, update the shared library cache with the ldconfig command, and clean up the build files with the make clean command.

```
$ cd ~/opencv/build
$ make -j4
$ sudo make install
$ sudo ldconfig
$ make clean
$ sudo apt-get update
```

7. Enable the Raspberry Pi camera by editing the config.txt file located in the system-boot partition of the microSD card. Turn off the Raspberry Pi, remove the microSD card, and add the line start_x=1 before the enable_uart=1 line in the config.txt file. Save the changes and re-insert the microSD card into the Raspberry Pi.
8. Install FFmpeg on the SBC. FFmpeg is a multimedia framework that can be used to process video and audio files, as well as capture images from video streams. This software is necessary for handling video data in the Autorace.

```
$ sudo apt install ffmpeg
$ ffmpeg -f video4linux2 -s 640x480 -i /dev/video0 -ss 0:0:2 -frames
1 capture_test.jpg
```

9. Install the ROS cv_camera package on the SBC. This package provides a node for capturing images from USB cameras and is necessary for obtaining images from the camera during the Autorace.

```
$ sudo apt install ros-noetic-cv-camera
```

10. Install additional dependent packages on the Remote PC, including packages for image transport, OpenCV support, and camera calibration. These packages provide essential functionality for processing and calibrating images captured during the Autorace.

```
$ sudo apt install ros-noetic-image-transport ros-noetic-image-
transport-plugins ros-noetic-cv-bridge ros-noetic-vision-opencv
python3-opencv libopencv-dev ros-noetic-image-proc ros-noetic-cv-
camera ros-noetic-camera-calibration
```

Appendix F: Camera Calibration Guide

E.1 Intrinsic Camera Calibration

1. Begin by printing the checkerboard pattern on an A4-sized paper. This pattern is essential for intrinsic camera calibration. The file can be found at *TurtleBot3_autorace_camera/data/checkerboard_for_calibration.pdf*.
2. To initiate the ROS core services, launch "roscore" on the Remote PC:
```
$ roscore
```
3. Activate the camera on the Single Board Computer (SBC) by running the "raspberry_pi_camera_publish.launch" file located in the "TurtleBot3_autorace_camera" package. SBC: $ roslaunch TurtleBot3_autorace_camera raspberry_pi_camera_publish.launch
```
$ roslaunch TurtleBot3_autorace_camera
raspberry_pi_camera_publish.launch
```
4. Start the intrinsic camera calibration by running the launch file. A calibration window will appear on the screen. Place the printed checkerboard in front of the camera at various angles and distances. Make sure that X, Y, Size, and Skew values turn green in the calibration window. Once they do, click "CALIBRATE":
```
$ roslaunch TurtleBot3_autorace_camera
intrinsic_camera_calibration.launch mode:=calibration
```
5. Save the intrinsic calibration data by clicking "Save". A compressed folder named "calibrationdata.tar.gz" will be generated in the "/tmp" directory. Extract this folder and locate the "ost.yaml" file, which contains the intrinsic calibration data.
6. Transfer the data from the "ost.yaml" file to the "camerav2_320x240_30fps.yaml" file to complete the intrinsic camera calibration process.

E.2 Extrinsic Camera Calibration

1. If not already running, launch "roscore" on the Remote PC to initiate the ROS core services:
```
$ roscore
```
2. If not already active, trigger the camera on the TurtleBot3:
```
$ roslaunch TurtleBot3_autorace_camera
raspberry_pi_camera_publish.launch
```
3. On the Remote PC, run the "intrinsic_camera_calibration.launch" file from the "TurtleBot3_autorace_camera" package, setting the "mode" parameter to "action". $ roslaunch TurtleBot3_autorace_camera intrinsic_camera_calibration.launch mode:=action
```
$ roslaunch TurtleBot3_autorace_camera
intrinsic_camera_calibration.launch mode:=action
```
4. Begin the extrinsic camera calibration by executing the "extrinsic_camera_calibration.launch" file from the "TurtleBot3_autorace_camera" package on the Remote PC. Set the "mode" parameter to "calibration". $ roslaunch TurtleBot3_autorace_camera extrinsic_camera_calibration.launch mode:=calibration
```
$ roslaunch TurtleBot3_autorace_camera
extrinsic_camera_calibration.launch mode:=calibration
```

5. Run "rqt" on the Remote PC and create multiple image view windows. Choose the `/camera/image_extrinsic_calib/compressed` and `/camera/image_projected_compensated` topics for each monitor. One window will display an image with a red rectangle box, while the other presents the ground projected view (Bird's eye view).

   ```
   $ rqt
   ```

6. Adjust the parameters for `/camera/image_extrinsic_calib/compressed` and `/camera/image_projected_compensated` using "rqt_reconfigure" on the Remote PC. . Fine-tune the parameters for `/camera/image_extrinsic_calib/compressed` until the red trapezoid aligns with the lane markings on the road. Make sure that the lane markings are clearly visible in the projected camera view, as it will be used for lane detection.

   ```
   $ rosrun rqt_reconfigure rqt_reconfigure
   ```

Appendix G: Lane Detection Guide

This appendix provides a step-by-step guide for lane detection calibration and execution using the TurtleBot3 with ROS Noetic.

F.1 Lane Detection Calibration

Follow these steps to calibrate the lane detection system:

1. Start the ROS core by running the following command in your terminal: roscore
2. On the TurtleBot3 Single Board Computer (SBC), launch the camera publisher by running: roslaunch TurtleBot3_autorace_camera raspberry_pi_camera_publish.launch
3. Launch the intrinsic camera calibration with the following command: roslaunch TurtleBot3_autorace_camera intrinsic_camera_calibration.launch mode:=action
4. Launch the extrinsic camera calibration by executing: roslaunch TurtleBot3_autorace_camera extrinsic_camera_calibration.launch mode:=action
5. Launch the lane detection calibration mode using the following command: roslaunch TurtleBot3_autorace_detect detect_lane.launch mode:=calibration
6. Start the RQT tool and select the following topics for each image view: /detect/image_yellow_lane_marker/compressed, /detect/image_lane/compressed, and /detect/image_white_lane_marker/compressed. You can do this by running: rqt
7. Run the RQT reconfigure tool to adjust the parameters for lane detection. You can fine-tune the values for hue, saturation, and lightness for both white and yellow lanes during the calibration process. To do this, open the RQT reconfigure tool by running: rosrun rqt_reconfigure rqt_reconfigure Adjust the parameters using the sliders while observing the three image views opened in step 6. This will help you to find the optimal values for accurate lane detection.

F.2 Lane Detection Execution

Follow these steps to execute lane detection:

1. On the remote PC, start the ROS core by running: roscore
2. On the TurtleBot3 SBC, launch the camera publisher by running: roslaunch TurtleBot3_autorace_camera raspberry_pi_camera_publish.launch
3. On the TurtleBot3 SBC, launch the TurtleBot3 bringup with the following command: roslaunch TurtleBot3_bringup TurtleBot3_robot.launch
4. Launch the intrinsic camera calibration by executing: roslaunch TurtleBot3_autorace_camera intrinsic_camera_calibration.launch mode:=action
5. Launch the extrinsic camera calibration with the following command: roslaunch TurtleBot3_autorace_camera extrinsic_camera_calibration.launch mode:=action
6. Launch the lane detection in action mode by running: roslaunch TurtleBot3_autorace_detect detect_lane.launch mode:=action
7. Launch the TurtleBot3 lane control to make the robot follow the detected lanes: roslaunch TurtleBot3_autorace_driving TurtleBot3_autorace_control_lane.launch