Bachelor thesis report

Engineering & Computer Science, Degree project 15hp

Virtual Validation of Autonomous Vehicles

Virtualizing an Electric Cabin Scooter

Halmstad University, Semcon

Halmstad, 2023-06-04

Jakob Andersson & Christoffer Arvidsson

HÖGSKOLAN
I HALMSTAD

Abstract

This thesis report presents a study on the virtualization of an Electric Cabin Scooter used to validate the feasibility of converting it into an autonomous vehicle. The project aimed to design, develop, and test a virtual model of the car that can navigate from points A to B while avoiding obstacles. The report describes the methodology used in the project, which includes setting up the workspace, construction of the virtual model, implementation of ROS2 controllers, and integration of SLAM and Navigation2. The thesis report also describes and discusses related work, as well as the theoretical background of the project. Results show a successfully developed working virtual vehicle model, which provides a solid starting point for future work.

Keywords: ROS2, Navigation2, SLAM, Gazebo, Virtual Environment, Ackermann steering, Autonomous Vehicle

Sammanfattning

Detta examensarbete presenterar en studie om virtualiseringen av en elektrisk kabinscooter. Den virtuella modellen används för att validera genomförbarheten av att omvandla den till ett autonomt fordon. Projektet syftade till att designa, utveckla och testa en virtuell modell av bilen som kan navigera från punkt A till B medan den undviker hinder. Rapporten beskriver metodiken som används i projektet, vilket inkluderar att sätta upp arbetsytan, konstruktion av den virtuella modellen, implementering av ROS2-kontroller och integration av SLAM och Navigation2. Rapporten diskuterar även relaterat arbete, samt teoretisk bakgrund till arbetet. Resultaten visar en framgångsrikt utvecklad fungerande virtuell fordonsmodell, som ger en solid utgångspunkt för framtida arbete.

Nyckelord: ROS2, Navigation2, SLAM, Gazebo, Virtual Environment, Ackermann steering, Autonomous Vehicle

Acknowledgments

We want to express our deepest gratitude to Semcon AB, particularly Mikael Wiktorell and Magnus Andersson, for their invaluable support and collaboration throughout this project. Their expertise, guidance, and resources have been instrumental in the success of this study.

We would also like to thank our academic supervisor at Halmstad University, Felipe Eduardo Valle Quiroz, for their invaluable advice, feedback, and encouragement throughout the research process and report writing.

Table of contents

1 Introduction

Modern robotics systems are complex entities equipped with a plethora of devices, sensors, and computers, which are often controlled by equally complex software. These systems need to be able to perform their tasks under different environments and dynamically changing conditions. However, testing the functionality of these systems under all potential different settings is not only time-consuming and expensive but also many times simply impossible.

For this reason, there has been increased interest in the use of so-called virtual environments [1]. Virtual environments are, broadly speaking, simulations that are created using computer software with the goal of testing the functionality and/or feasibility of a system in a controlled environment. Performing tests in a virtual environment allows one to execute several tests at the same time, in conditions that are, in general, unreproducible in a laboratory environment, while preventing any risks of damage to the real equipment in case of failure. Therefore, by making use of a well-developed virtual environment, the development process can be made to be more exhaustive and cost-effective. This virtualization process is the basis of this thesis work.

1.1 Purpose

This thesis work will be done in collaboration with the company Semcon [2] in Gothenburg, which has recently acquired a small electric car with space for one driver (see Figure 1). Their ambition is to turn this vehicle into an autonomous one, however, instead of testing the car’s autonomous functions in real life. They wish to create a virtual model of the car along with a virtual environment in which to perform simulated testing. The virtual model should be equipped with the corresponding virtual sensors to the real car, in corresponding positions, to mimic the real car and its functionalities.

This work will focus on designing, developing, and testing said virtual environment, as well as making necessary changes to the configurations and settings of the avatar model.

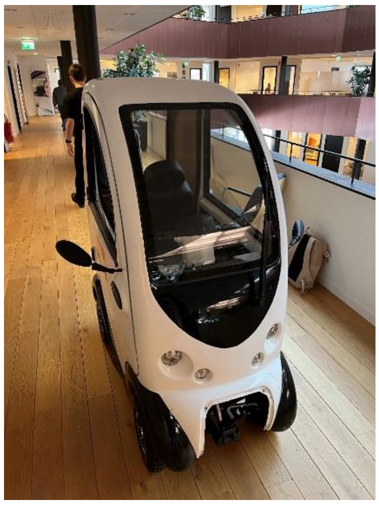


Figure 1- The electric vehicle acquired by Semcon

1.2 Goals & Delimitations

This thesis work aims to create a virtual environment and model on which the autonomous functions of the vehicle can be tested. To achieve this goal, a set of subgoals need to be achieved:

- Create a virtual model of the vehicle, which accurately represents the mobility and collision characteristics of the real vehicle.
- Implement necessary packages and features to turn the model autonomous.

To achieve these goals, certain delimitations will be necessary. Specifically, this thesis work will:

- Focus exclusively on the development of the virtual model and the autonomous functions, without consideration of other features like suspension or actuators.
- Assume ideal operating conditions for the virtual model, not taking features like weather or road conditions into account.
- Not involve physical testing of the real vehicle, which will be subject of work based on the result of this thesis.

1.3 Requirements

- The virtual model should be able to navigate from point A to B in a known environment, while avoiding pre-defined as well as unknown static obstacles.
- If avoiding collision is deemed impossible, the vehicle should stop.

2 Background

2.1 Related Work

Due to its ability to test and validate the behavior of autonomous vehicles, virtual validation has become an increasingly popular technique during recent years [3]. For example, researchers in [4] describe the process of modeling a vehicle in URDF [5] (Unified Robot Description Format), an XML format used to describe the kinematic and visual properties of a robot for simulation and visualization. They implemented Ackermann steering and a lidar sensor on the model and imported it into a virtual environment. However, in the report we have access to, no testing was performed to test the navigation capabilities of the vehicle. While in this work, the testing part is key to correctly determine the feasibility of automating the vehicle.

Another similar project is found in [6] where the researchers attempted to create an autonomous golf cart. They created a virtual version of the car using URDF much like the previous work described. However, in this study, virtual testing is performed on the navigation characteristics. While testing, the developers found that the already existing algorithms for navigation were more suited for a differential drive robot rather than their implemented Ackermann steering. Upon modification, the navigation package could be used for the intended usage. A physical prototype was also developed, with corresponding sensors to the final car, to test the steering characteristics in the real world.

The researchers in [1] present a simulation environment that will be used for testing mobile robots using ROS (Robot Operating System) and the simulation program Gazebo [7]. The simulation environment enables researchers and developers to test mobile robots in a virtual environment before deploying and testing them in real-world scenarios, thus reducing costs and risks. The researchers describe the design and implementation of their simulation environment, which includes the mobile robot model, sensor models, and controller models. They demonstrate the effectiveness of the simulation environment by testing the mobile robot in its simulated environment and comparing the results with real-world experiments. The paper illustrates the importance of simulation environments in robotics R&D and shows how ROS and Gazebo can be used to develop such environments. This work will make use of several of those software tools that have been developed and used in state-of-the-art research, including ROS2 [8], Gazebo, and Navigation2 [9].

2.2 The Vehicle

The inherited vehicle used in this work is an electric cabin scooter from the manufacturer Blimo [10]. To simulate a vehicle, certain key components and systems of the car must be replicated virtually. In this case, that would be the vehicle's mass, inertia, and wheel parameters, such as friction, radius, and width. The sensors need to be replicated to ensure that the autonomous functions of the car perform similarly in the real world.

2.2.1 Physical properties and measurements

Creating an accurate model of the vehicle in Gazebo requires some important measurements. Due to limited documentation on the car, we have acquired some of the following measurements ourselves.

Vehicle width (cm)	71(117 including mirrors)
Vehicle length (cm)	155
Vehicle height (cm)	160
Vehicle weight (kg)	242
Track width (cm)	60
Wheelbase (cm)	99
Tire radius (cm)	18
Ground clearance (cm)	9 (lowest point)
Turning radius (cm)	235

The inertial properties can be calculated with the help of Xacro [11], which is an XML macro language. Xacro will be explained in more detail later in the report.

3D-scanning [12] is a method that can be used to create a precise virtual copy of the vehicle’s chassis. This method works by using a laser to create a point cloud from the vehicle.

2.2.2 Sensors

The vehicle will be equipped with a lidar sensor to help navigate the car. The sensor (Velarray M1600) comes from Velodyne and is a Solid-State lidar (SSL) [13]. By being an SSL, it uses solid-state components such as photodetectors and lasers instead of mechanical components to measure distance. This also makes the sensor lighter, so it is often used in applications where size and weight are important factors, such as autonomous vehicles. One downside of using an SSL instead of a mechanical lidar is that the field of view, FOV, can be smaller. Velarray M1600 has a FOV of 120° instead of 360°, which a mechanical lidar has.

2.2.3 Ackermann Steering

Four-wheeled robots have several different steering mechanisms, depending on how the vehicle is designed. Commonly used in robots is differential drive or skid steering because robots usually have fixed wheels. In this case, an Ackermann-type steering is most suitable. Ackermann steering mechanism [14] is used when both front wheels should be able to turn with minimal tire slippage, see Figure 2. A steering linkage connects the front wheels, and turning the wheel causes the wheels to turn at a different angle, to maintain a constant turning radius. The maximum angle of the inner and outer wheels can be calculated by using the formulas:

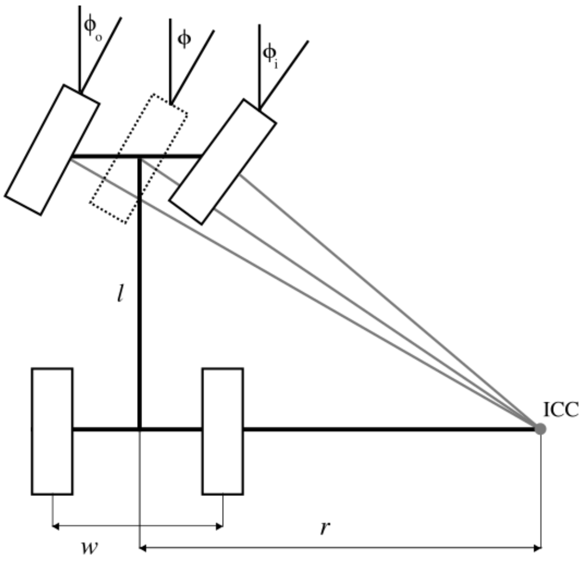


Figure 2 – Ackermann steering kinematics [15]

$$\phi_i = tan^{-1}\left(\frac{l}{r - \frac{w}{2}}\right) \tag{1}$$

$$\phi_o = tan^{-1}\left(\frac{l}{r + \frac{w}{2}}\right) \tag{2}$$

Where *l* is the wheelbase, *r* is the turning radius, and *w* is the track width. ϕ_i is the maximum inner wheel angle and ϕ_0 is the maximum outer wheel angle, in radians.

2.3 Simulation & Software

Autonomous vehicles have been widely studied in recent years [16,17]. Much of the research in autonomous vehicles is done via simulators, mainly due to high costs and safety precautions [3]. Most of these simulations are built on the open-source software platform ROS2 (Robot Operating System 2) [8]. ROS2 is a framework that provides a different set of tools and libraries that makes it easier to construct a robotic system. ROS2 calls the processes of robot system *nodes* [18], each responsible for a specific task (e.g., controlling a motor or collecting data from a sensor). Nodes can communicate with other nodes through three similar, but distinct ways: t*opics* [19] which provide a way for nodes to exchange information with each other in a publish/subscribe type of communication; *services* [20] which are similar to topics but use a call/response communication instead (request a service from another node and receive a response like HTTP); and *actions* [21] which again are similar to services, but the communication through actions enables feedback during the execution of the task, this also allows for task cancellation (e.g., moving a robotic leg while sending feedback back to the requesting node so it can cancel the task if necessary).

Despite its name, ROS2 is not an operating system but instead a type of software which is called "middleware." This means that the software acts as a bridge between an operating system, different applications, and hardware components allowing them to communicate and exchange data with each other [22]. Using ROS2 as a middleware allows us to connect the various parts of the robot, including hardware and software elements, and enable them to communicate and work together seamlessly. This makes it easier for developers to focus on writing the code for the actual tasks rather than worrying about the details of how the different components will interact and work together.

One of ROS2 useful packages is *ros2_control* [23] which is a framework designed to control robot hardware components, such as actuators, sensors, and controllers. The package makes it easy to switch between hardware configurations, as well as to simulate hardware in a virtual environment. *Ros2_control* supports different controller plugins, such as *Joint_trajectory_controller* and *diff_drive_controller*.

Diff_drive_controller is a controller for differential drive robots, which is a type of robot that uses wheels that can rotate independently. A robot like this can drive each wheel both forwards and backward, which enables turning in place. The controller gets an input of body velocity commands which are then translated to wheel commands for the differential drive base. The *diff_drive_controller* also calculates the robot's odometry, which estimates its position and orientation in the simulation environment. The odometry information is important for navigation tasks, and it can be used by other nodes in the ROS2 system to make decisions and plan trajectories. Odometry is about using data from sensors to estimate the position of the model [24]. The sensors can be wheel encoders or a lidar, or in our case, the */joint_states* topic provides us with information.

Joint_trajectory_controller is a controller which can control multiple joints by taking a desired trajectory as input, typically in the form of a sequence of joint positions or angles, and then generates the necessary commands to achieve the desired position. This controller is good to use on multi-joint robots.

Another useful package in ROS2 is the *geometry_msg* [25] package which provides a collection of message types that nodes can use to exchange information with each other. A common message type is a Twist message, which represents the linear and angular velocities of a robot. This is commonly used in ROS2 to control the movement of a robot.

Two other essential and common packages are the *robot_state_publisher* [26] and *joint_state_publisher* package [27], which are responsible for publishing a robot's state and joint information, respectively.

Robot_state_publisher calculates the positions and orientations of all the robot's links based on the joint states, which are usually provided by a URDF file. This information is published on topics that other nodes in the ROS2 system can use to understand the robot's configuration and perform tasks such as motion planning, sensor data processing, and visualization.

Joint_state_publisher publishes information about the positions, velocities, and efforts of all a robot's joints to a specific topic that nodes in the ROS2 system can access.

2.3.1 Navigation2

ROS2 comes equipped with a navigation stack (Navigation2/Nav2) [9], that provides most features needed to navigate the vehicle from point A to B, without contacting obstacles in the path. Navigation2 is a set of different plugins that adds different navigation functionalities, such as global and local planners, cost map generators and recovery behaviors if a robot gets stuck or detects a collision, for example. These navigation functionalities are coordinated by a behavior tree which is a powerful and flexible control system for designing decision-making processes in robotics.

With the use of the behavior tree, the global and local planner, cost map generators, and recovery behaviors are effectively integrated and ensures seamless navigation, illustrated in Figure 3. The global planner generates an initial path from the robot's current position to the desired goal. After the initial global path is set, a controller generates a local path, and velocity commands in the form of a twist message are sent to the robot, which enables the robot to follow the global path and avoid obstacles in real-time.

If the robot encounters a potential collision, then the behavior tree triggers a corresponding recovery behavior to handle the situation. Recoveries can be rotating in place, reversing, clearing the cost map, etc. This enables custom recovery strategies tailored to the robot's capabilities and application requirements.

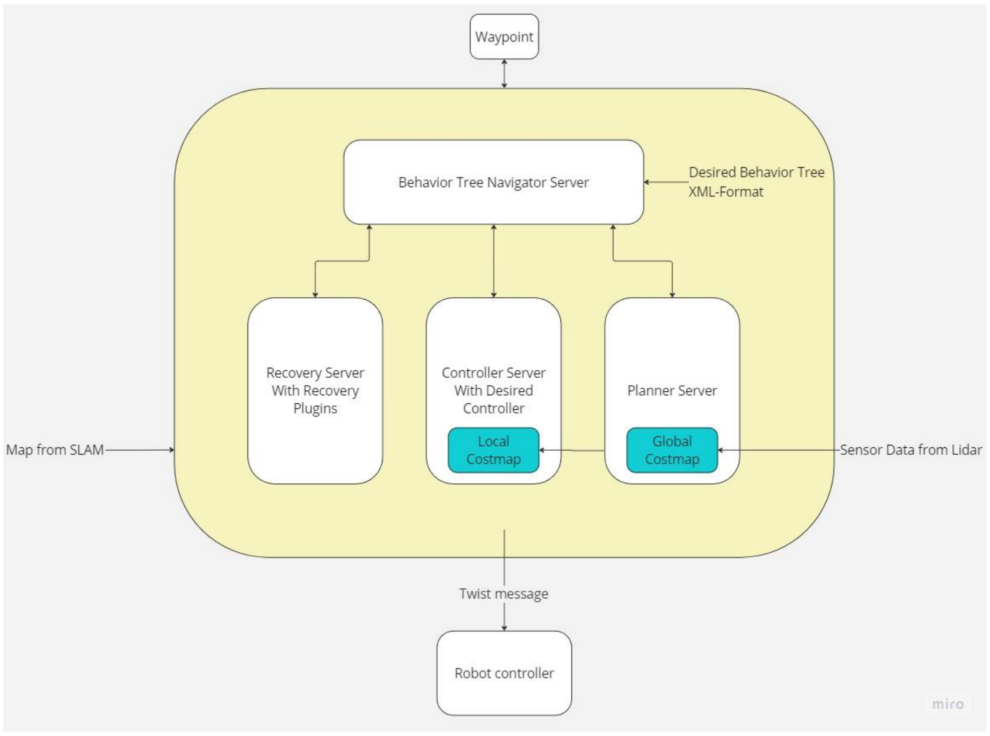

Waypoint
Behavior Tree Navigator Server
Desired Behavior Tree XML-Format
Recovery Server With Recovery Plugins
Controller Server With Desired Controller
Planner Server
Local Costmap
Global Costmap
Map from SLAM
Sensor Data from Lidar
Twist message
Robot controller
miro

Figure 3 – Block diagram explaining the structure of the Navigation 2 stack

2.3.2 SLAM – Simultaneous Localization and Mapping

For the robot to navigate in an environment. It needs to be able to localize itself and map the surroundings. For that, SLAM [28] is used. ROS2 provides support for multiple different packages, such as SLAM_toolbox [29], Hector Slam [30], and Gmapping [31]. SLAM uses data from sensors such as lidars and RGB cameras to localize the position of the robot by extracting important features. These features can be used to identify landmarks or objects in the environment. The features are then matched with previous sensor readings to determine which features belong together. These landmarks or features help build a map incrementally. Revisiting the same areas improves the accuracy of the map due to the SLAM functionality of loop closing, which recognizes previous scans and patches the map together, correcting any faulty estimates.

2.3.3 Rviz

Rviz (ROS Visualization) [32] is a 3D visualization software tool for robots, sensors, and algorithm. The software communicates with ROS2 through the publish/subscribe pattern which is the core communication system of the ROS2 ecosystem. It provides a graphical interface for visualizing and interacting with data produced by different ROS2 nodes, such as sensor data, robot states, and planned paths. Rviz helps to monitor, debug, and analyze the behaviors of robot systems in real time.

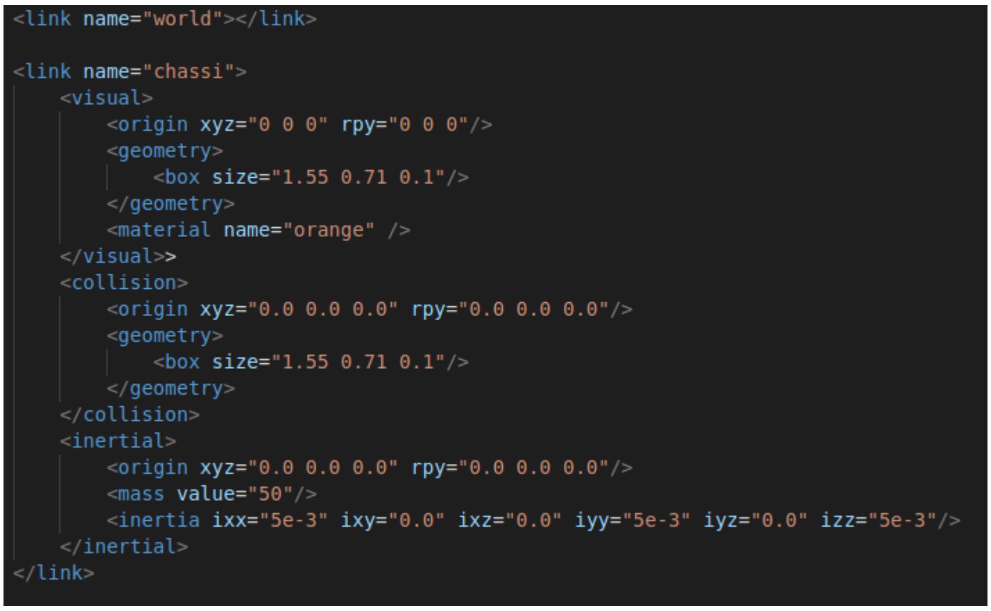
The interface in Rviz is highly configurable and it lets the user add different interaction tools to suit the specific needs. For instance, it is possible to insert tools that enable SLAM and Navigation2 configuration.

Integrating Navigation2 into Rviz enables the visualization of the global and local paths generated by the Nav2 planners. It also enables the user to publish information on specific topics. For instance, when setting a navigation goal, Rviz publishes a goal message to a specific topic to which the Navigation2 node is subscribed to.

2.3.4 Model

A common way to create a virtual avatar of a robot is to describe it in an XML format using URDF, which describes the physical properties of a robot, such as mass, size, and shape as well as the visual properties, color, and shape. The URDF is built up of several tags and sub-tags, with the joint tag and link tag considered the most central ones.

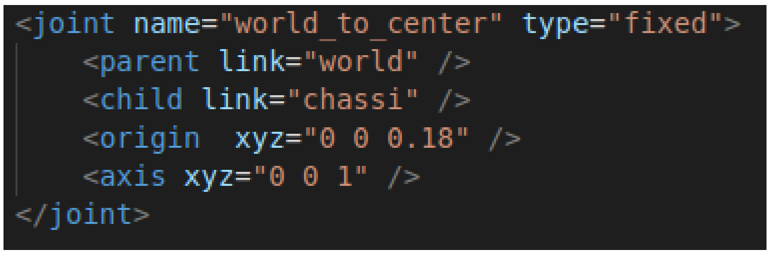
Links describe the visual and physical properties of a body part in the robot. To implement these properties, several *sub-tags* are used, these being <visual>, <inertia>, and <collision>. The visual tag defines the appearance of the body part. The color, size, and origin parameters for the body part are set in this section. To be able to simulate the robot in Gazebo or similar simulation programs, inertia and collision need to be defined.



```
<link name="world"></link>

<link name="chassi">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="1.55 0.71 0.1"/>
        </geometry>
        <material name="orange" />
    </visual>>
    <collision>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
        <geometry>
            <box size="1.55 0.71 0.1"/>
        </geometry>
    </collision>
    <inertial>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
        <mass value="50"/>
        <inertia ixx="5e-3" ixy="0.0" ixz="0.0" iyy="5e-3" iyz="0.0" izz="5e-3"/>
    </inertial>
</link>
```

Figure 4 - Example of two links in an URDF file format, the "world" link is a base link, meaning that it has no parent. The "chassi" link is a fully defined link, with visual, inertial, and collision properties

To connect two links, a *joint* is used, see Figure 5. The joint defines the relation between the two links, the parent link, and the child link. The joint tag sets limitations of movement as well as effort and velocity. Joints can be constructed with different types of motion allowed. Some of the most common joints being fixed, revolute, continuous, prismatic, and planar. A fixed joint allows no movement, and the link becomes part of the parent link. Revolute allows the child link to rotate around a specific axis with limits on the range. Continuous joints allow rotation around a single axis, like a revolute joint. However, no limit is applied to a continuous joint, making it useful in the context of building a car. Planar and prismatic joints allow the links to move along one and two axes, respectively.



```
<joint name="world_to_center" type="fixed">
    <parent link="world" />
    <child link="chassi" />
    <origin  xyz="0 0 0.18" />
    <axis xyz="0 0 1" />
</joint>
```

Figure 5- Example of a fixed joint in an URDF file format, which connects chassi to world

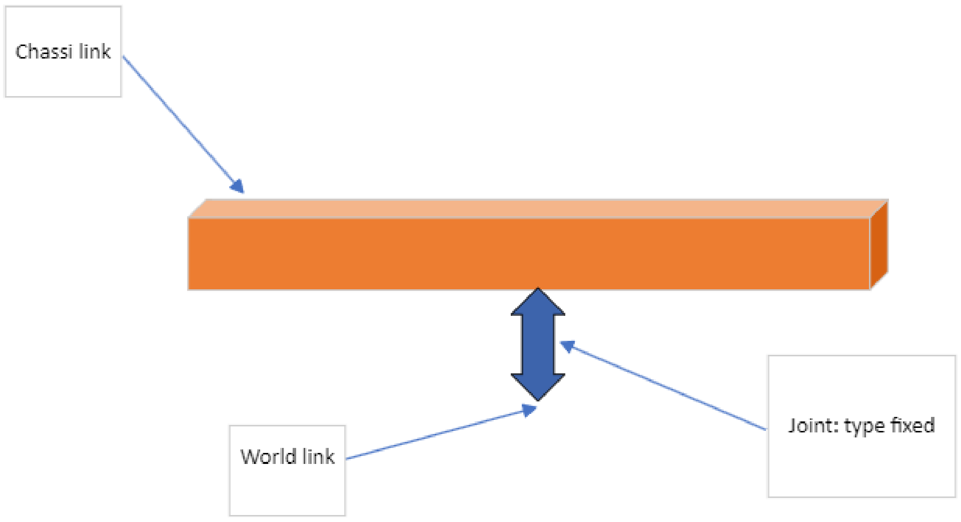


Figure 6 – Illustration of previously described URDF structure where the world link, which has no visual element and is used as a reference point for the world, is the parent and the chassi is the child link

During the development of the model, the robot can be visualized in Rviz, a graphical interface that uses plugins for various topics to visualize the model, transforms, as well as sensor data. Thus, making it easy for a developer to review recent changes in real time. To get an understanding of the connection between the nodes and topics in a project, the user can utilize the graph plugin within Rqt (ROS Qt) [33]. Rqt is a graphical interface plugin framework that provides numerous tools to help developers understand the structure of the project, see Figure 7.

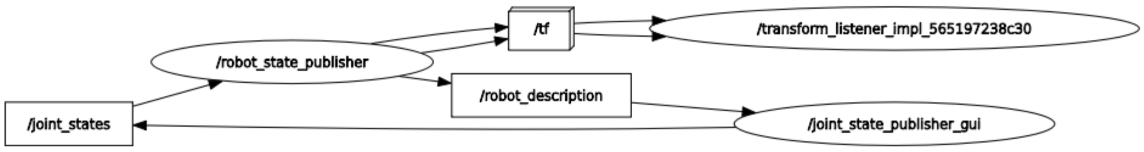

/tf
/transform_listener_impl_565197238c30
/robot_state_publisher
/robot_description
/joint_states
/joint_state_publisher_gui

Figure 7- Simple Rqt graph describing relation between topics and nodes, where the squares are topics, and the ovals are nodes. In this case, the "/joint_state_publisher_gui" subscribes to the "/robot_description" topic and publishes to the "/joint_states" topic

2.3.5 Simulation

After developing a complete URDF model, simulations are required to ensure functionality. The simulations can be done within Gazebo [7], a 3D robot simulator plugin for the ROS2 platform. Gazebo utilizes plugins to be able to simulate steering and control of a robot model. With correctly setup link and joint attributes, Gazebo can accurately simulate the behavior of the robot in a controlled environment. As mentioned previously in the report, aspects like the steering, battery consumption, and sensor performance can be simulated in Gazebo. To do this Gazebo utilizes plugins, a software component that allows the user to extend the functionality of the program. A plugin can be downloaded or created by the user, enabling the developer to create custom simulation environments for the intended usage [34]. The main reason for choosing Gazebo over other similar simulators such as Webots [35], Coppeliasim [36] or MORSE [37] is that the same developers that developed ROS also developed Gazebo [38]. This, along with the large community supporting Gazebo, which can help by providing information through forums and tutorials, make this plug-in the easy choice for this project.

3 Method

In the following section, strategies and tools that have been implemented will be described. The first part of the project time was spent on familiarizing us with the tools and programs, as well as reading up on previous similar work. The project's main focus was creating a virtual model and environment and establishing a working setup of nodes and packages to turn the model autonomous.

3.1 Planification Stage

During initial meetings with the company, potential scopes and plans for the project were discussed based on time availability. The original plan conducted was to construct the model and implement Navigation 2 and SLAM. When this was finished, the performance of the model was to be tested in randomized worlds created by AWS World Forge [39]. Testing would then be performed using a CI/CD pipeline [39], if time allows it, to automate and speed up the testing.

However, if issues arose with the AWS RoboMaker [39], then testing could be performed through manually made virtual environments. However, constructing manual environments in Gazebo is less efficient and more time-consuming. This will mean that, in this case the model would only be tested using a limited number of environments, however, this will at least ensure that testing is performed. The amount of time available in this case would ultimately determine the number of environments and their complexity.

3.2 Project Workspace Setup

The project was supplied with a virtual machine copy by the company which runs on Linux Ubuntu 22.04 and contains all the software programs that are relevant to the project, ROS2 Humble [40], and Gazebo 11 being the main ones. ROS2 includes support for essential features and tools for the project, such as Rviz and Navigation2.

ROS2 provides support primarily for Python and C++, in this case we decided to use Python as the primary language to work in because of our familiarity with it and its simplicity with respect to C++. Python is a free high-level programming language that is designed to be easy to learn and use, making it great when developing robotics applications. It also has a large community base and extensive 3rd party library support, so there are many open-source resources and tools available to use with ROS2 [40].

In addition, a helpful tool when writing URDF files is Xacro [11], an XML macro language that reduces the amount of code that needs to be written by allowing the developer to create macros that can be used to represent commonly used parts in a robot model. It improves the readability of the code and allows for faster modification of the model.

The ROS2 workspace was created and edited with VSCode, an open-source code editor [41]. The reason behind choosing VSCode is that it supports multiple different code formats such as Python and XML, which allow us, the developers to mix freely when building the workspace. It also offers integration with GitHub [42], which simplifies commits to the online repository as well as cloning the code to the local repository.

3.3 Software Tool Familiarization

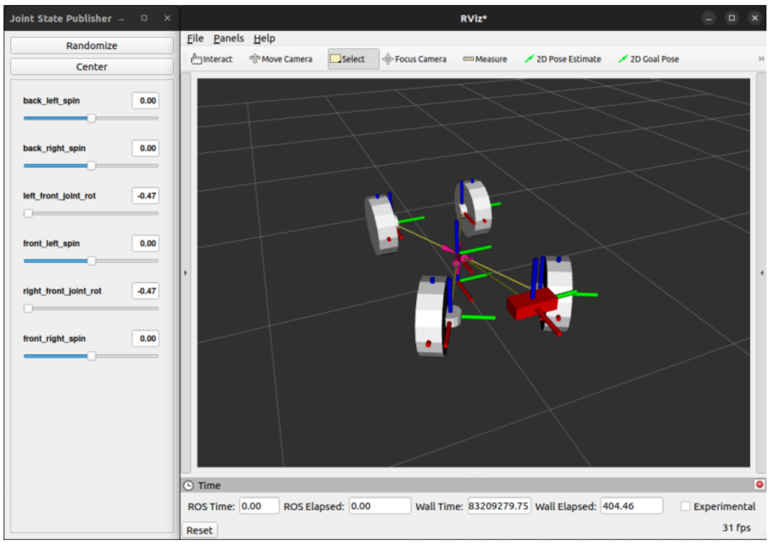
Due to our limited experience with ROS2 and simulation in virtual environments. The first weeks were spent reading about previous and similar work as well as familiarizing ourselves with the structure and principles of ROS2. As mentioned before, the ROS2 environment runs on a virtual box supplied by the company. The virtual box used Linux Ubuntu 22.04 [43] as the operating system, which was another component that we needed to get used to. It is worth mentioning that to familiarize ourselves with ROS2, we followed several tutorials regarding the handling of topics, nodes, and structuring a workspace, among other important features. Learning the foundations of the software set the project back a couple of weeks.

3.4 Virtual Model Construction and Spawning in Gazebo

Once the project's workspace was set up correctly and we had familiarized ourselves with ROS2 and its communication system, we created a URDF file for the vehicle using the dimensions taken from the real car at the company's headquarters. These measurements include the wheelbase dimension, tire radius, and ground clearance, but also the external dimensions of Wille's chassis. We also used a 3D scanner to obtain a visual mesh, see Figure 9, as well as the collision properties of the vehicle. The mesh obtained by 3D scanning the vehicle was saved as a .stl file, which was imported into Gazebo to transfer the physical properties of the chassi into the simulation.

The process began by constructing the URDF file via the creation of a chassis link in the center, which we then use to attach every other part, including the 3D scanned mesh. The two front wheel links referred to as *"left_front_wheel"* and "*right_front_wheel*" were attached to two rotational links *"front_left_rot"* and *"front_right_rot,"* through two continuous joints, which enables certain degrees of freedom around the z-axis based on the Ackermann steering calculation in equations 1 and 2, while enabling the wheels to spin. Similarly, the two rear wheels, *"left_back_wheel"* and *"right_back_wheel,"* were connected to *"rear_drive_left"* and *"read_drive_right"* via continuous joints.

We used Rviz to visualize the vehicle in a simple environment, see Figure 8. To be able to control the different joints, we also initialized the Joint State publisher graphical interface, which is a ROS2 package that publishes the state of a robot joint to the ROS2 system. This is done by reading the values of a joint and publishing them as a message on a the *“/joint_states”* topic. In addition to using Joint State publisher to control the joints, we also used the Robot State publisher package which reads the URDF file and computes the robot’s kinematic state and publishes the information on the *“/robot_description”* topic.



back_left_spin 0.00
back_right_spin 0.00
left_front_joint_rot -0.47
front_left_spin 0.00
right_front_joint_rot -0.47
front_right_spin 0.00

ROS Time: 0.00 ROS Elapsed: 0.00 Wall Time: 83209279.75 Wall Elapsed: 404.46 Experimental

Figure 8 – The vehicle visualized in Rviz with the use of Robot State Publisher and Joint State Publisher

Next, to simplify the launch of all the necessary nodes and spawn the vehicle in Gazebo we proceeded to develop a python launch file that initializes the following nodes automatically:

1. URDF Model: Loads the URDF model.
2. Robot State Publisher: Launches the robot state publisher node, which reads the URDF file and computes the robots kinematic state and publishes the information to the *“/robot_description”* topic.
3. Joint State Publisher: Initialization of the joint state publisher node which publishes positions, velocities, and efforts of each joint as messages on the *"/joint_states"* topic.
4. Gazebo simulation environment: Initializes the Gazebo node and opens the desired Gazebo world, see Figure 9.
5. Rviz visualization: Launches the Rviz node to be able to inspect and interact with the joints in real time.

As the project progresses and new nodes need to be created, we can easily extend and update this launch file to integrate new functionalities and features into our vehicle model.

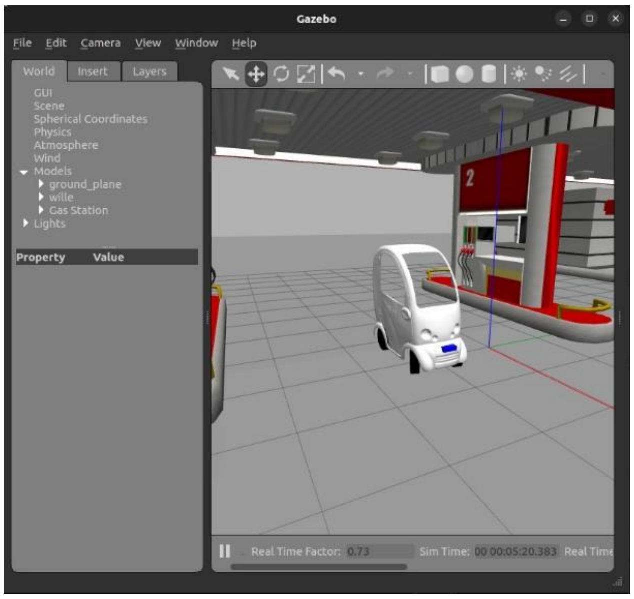


Figure 9 – The vehicle visualized in a Gazebo world, with the mesh from previous 3D-scan incorporated, providing collision properties for the vehicle

3.5 Implementation of Controllers

The next step was the implementation of the control systems. Controllers are the key component that enable the movement of several joints at the same time and allow the user to control the vehicle in a way that closely resembles and mimics an actual car. After exploring the different possible controllers available in ROS2 Humble, we chose to implement the steering with a *diff_drive_controller* controlling the rear wheels. As well as the *joint_trajectory_controller* to control the rotation of the front wheels. These were later configured to replicate the Ackermann steering characteristics. Along with these two controllers, a joint broadcaster was implemented, which keeps track of the state of the joints controlled by the controllers. This helped when visualizing the robot in Rviz, to see if the joints moved as expected.

To move the car manually and send input to both controllers, a script that uses the *teleop_twist_keyboard* [44] was constructed. The *teleop_twist_keyboard* takes key presses as inputs and generates a twist message that can be sent to the controllers. The custom script used key presses of “w” for forward movement, “s” for stopping, and “x” for reversing. Turning was handled by pressing “a” or “d.” Each key press generated a twist message.

One issue with this solution was that the two controllers needed input on two different formats. The *diff_drive_controller* takes twist messages, which this script generates, and the *joint_trajectory_controller* takes position messages. Due to this, another node, “*twist-to-joint”,* was created that subscribed to the custom script created earlier and converted the angular part of the twist message into a position message. This message was then sent topic to which the *joint_trajectory_controller* subscribes. The structure of this conversion can be seen in Figure 10.

Apart from converting the twist message to a position format, the *"twist-to-joint"* node converted the desired steering angle from the twist message into two individual angles for the joints depending on if the message was negative or positive, which indicated which way the car was turning. The conversion was based on the formula described in equations 1 and 2, but to find out the specific angle, the result was multiplied with a factor: received angle/maximum angle. Since the formula only generates the maximum angle for each wheel, multiplying by this factor gives the Ackermann angle based on the input.

$$\phi_{inner} = \phi_i \ * \frac{\phi}{\phi_{max}} \tag{3}$$

$$\phi_{outer} = \phi_o \ * \frac{\phi}{\phi_{max}} \tag{4}$$

Where ϕ_i and ϕ_o are the maximum angles for respective wheels, as per Figure 2. ϕ is the received angle and ϕ_{max} is the maximum angle that the input can generate. This conversion is done due to the fact that Navigation 2 generates different values based on the desired linear speed.

3.6 Implementing SLAM and Nav2

As discussed in the background, the model needs to be able to first localize itself and map the surroundings to plan a safe path to a target before starting to navigate. In this work, the implementation was done using a 2D lidar sensor model. This was due to the fact that using a 3D lidar in the simulations would require a large amount of processing power. In addition, during our preliminary investigation we concluded that several state-of-the-art projects seem to use 2D lidar [45, 46, 47]. This meant that there is more information sources and channels available to us in case there is a need of troubleshooting. The lidar configurations regarding resolution and range also had to be modified to preserve processing power. Once we had mapped a demo world, we decided to implement the Nav2 package and tune the parameters for Nav2 by testing in the example world.

3.6.1 Writing Custom Nodes and Scripts

To run the previously designed steering script simultaneously to the Nav2 and to avoid having to change topics for our previously written nodes, we decided to create two muxes using the "*twist_mux*" package in ROS2. A mux, short for multiplexer, is a device or component which allows for selection of an output from multiple inputs [48]. The two muxes listened to the two topics generating messages to each controller. The mux regarding the front wheels listened to two topics, one published from the Nav2 node and one where the message was published with the steering script. The advantage of using a mux, besides being able to listen to inputs from two topics, is that the developer can order them in priority. In our case, we put a higher priority on the message generated by the steering script which enabled the ability to intervene with the manual control in case the model got stuck. The mux sent the prioritized info through the previously created node to convert the message to a position format and to get the Ackermann angle.

The mux regarding the rear wheels needed more configuration. Since Nav2 generates both angular and linear messages, the angular part of the message needs to be filtered away. Sending both linear and angular messages to the diff drive controller would cause the rear wheels to move at different speeds. Another node, "*Twist-to-linear*", was then created which only sent the linear part of the twist message to the topic on which the controller subscribed. Apart from that implementation, the mux regarding the rear wheels was implemented in the same way, with the Nav2 message being prioritized. Figure 10 shows the relationship between the different system components used in our ROS2 configuration

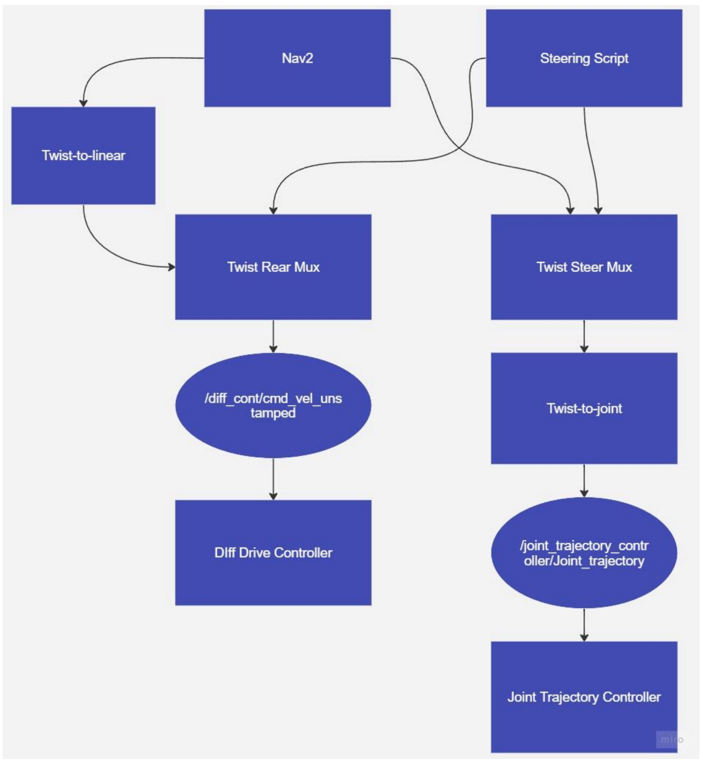

Nav2
Steering Script
Twist-to-linear
Twist Rear Mux
Twist Steer Mux
/diff_cont/cmd_vel_unstamped
Twist-to-joint
DIff Drive Controller
/joint_trajectory_controller/Joint_trajectory
Joint Trajectory Controller

Figure 10 – Overview of the remapping and restructuring of the messages generated by Nav2 and the Steering Script.

Upon testing the navigation, we noticed that the model in Rviz and Gazebo moved differently. We decided to investigate the odometry settings and realized that only the diff drive controller published odometry, causing the model to move strangely in Rviz. To correct this, we needed to create a custom node that calculated the odometry and published it as a fixed frame.

The created node retrieved information about the wheels and joints by subscribing to the "*/joint_states*" topic. The topic holds information about the velocity and position of every wheel as well as the front steering joints. In order to estimate the position and direction of the vehicle, a steering angle and linear velocity needed to be obtained. The steering angle was calculated by using the mean of the two front steering joint's positions. The linear velocity was obtained by calculating the mean of the velocity of the rear wheel. The main reason behind only using the rear wheels was that we had an issue with the front wheels that kept spinning even though the vehicle stopped. This is not something that affected the performance of the model, so the most direct way of avoiding the potential problem was to use the rear wheels for estimating the speed. The linear velocity was then used to calculate the angular velocity.

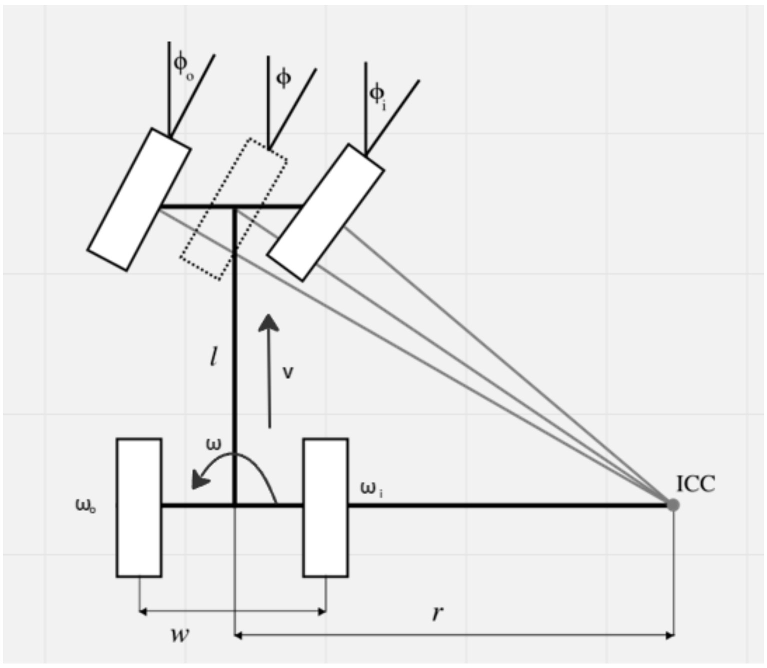


Figure 11- Ackermann steering kinematics with angular and linear velocities [15].

$$angle = \frac{(\phi_{\text{inner}} + \phi_{outer})}{2} \tag{5}$$

$$v = \frac{\omega_i + \omega_o}{2} * wheel\ radius \tag{6}$$

$$\omega = v * atan(angle) \tag{7}$$

Where ϕ_{inner} is the angle of the right wheel, and ϕ_{outer} is the angle of the left wheel, as explained in equations 3 and 4. ω_i, ω_o is the rotational velocity of the right, respectively left rear wheels. The linear velocity is *v* in this case and ω is the angular velocity.

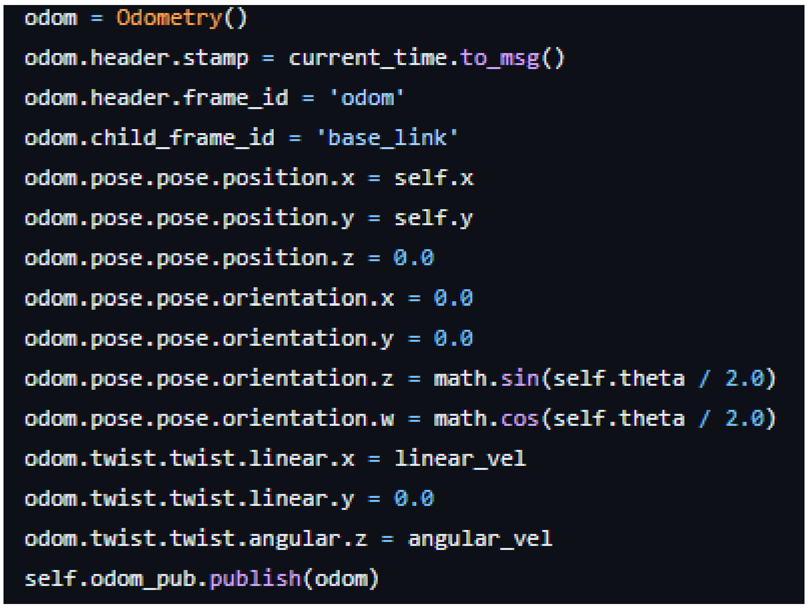
Furthermore, to update the position of the model, the following equations were used:

$$theta = \omega * dt \tag{8}$$

$$x = v * \cos(theta) * dt \tag{9}$$

$$y = v * sin(theta) * dt \tag{10}$$

Where dt is the time elapsed between each callback of the code, the position and orientation are then published on the topic "*/my_odom*".



```python
odom = Odometry()
odom.header.stamp = current_time.to_msg()
odom.header.frame_id = 'odom'
odom.child_frame_id = 'base_link'
odom.pose.pose.position.x = self.x
odom.pose.pose.position.y = self.y
odom.pose.pose.position.z = 0.0
odom.pose.pose.orientation.x = 0.0
odom.pose.pose.orientation.y = 0.0
odom.pose.pose.orientation.z = math.sin(self.theta / 2.0)
odom.pose.pose.orientation.w = math.cos(self.theta / 2.0)
odom.twist.twist.linear.x = linear_vel
odom.twist.twist.linear.y = 0.0
odom.twist.twist.angular.z = angular_vel
self.odom_pub.publish(odom)
```

Figure 12– Code snippet generating the odometry message published on the "/my_odom" topic

To use the updated odometry obtained from the wheel speeds and positions, we then needed to create a translator component between the odometry message and the base link. So, a new node was created that subscribed to the odometry message and broadcasted a transformed version of it between the base link of the world and the odometry, thus enabling a fixed frame to use.

3.6.2 Setting Up Nav2

Once the odometry was updated to get a more accurate representation, it was time to continue with the SLAM and Nav2 implementation. We ran the SLAM on the same environment as before and noticed that the estimation of the position of the model was improved, which also led to a more accurate scan. Nav2, however, proved more complex to get an improvement on. The navigation stack contains several launch files and a parameter file, where the developer can change the parameters of the currently used plugins or replace them with other ones. Nav2 was originally configured to work with a differential drive robot, which meant that the planning and behavior of the robot in case of a detected collision differed from what was desired.

Thus, for the model to be able to perform the desired actions, some plugins had to be modified:

1. The original controller plugin, *"DWB Controller"* [49], was swapped out for the "Regulated Pure Pursuit" [50] controller, which is more suitable for an Ackermann-like steering [51].
2. The original planner plugin *"NavFn Planner"* [52] was swapped with the *"SmacPlannerHybrid"* [53]. The main difference between the planners is that the *NavFn planner* finds a straight plan and disregards the orientation of the vehicle since differential drive robots can turn in place. However, the *SmacPlannerHybrid* generates a global plan that takes the goal direction into account.
3. Finally, the standard recovery behaviors were changed. Originally, three commands would be sent to the model in case of an error or an impending collision. One to spin the vehicle around, one to reverse the vehicle, and one to wait. In our case, we would just like the vehicle to stop in case of a collision or an error, so the two first behaviors were removed.

Once the plugins had been changed to fit our setup better, parameter tuning could be performed with regards to speed, distance to obstacles, and goal precision.

3.7 Experimental Evaluation

To analyze the performance of our implementation of SLAM and Navigation2, multiple experiments were conducted. The experiments aimed to evaluate the performance of both the SLAM and Navigation2 node configurations. As stated before, both SLAM and Navigation2 are configured with two different parameter files. Since there are millions of potential parameter configurations, the experiments were not aimed at finding the optimal parameter configuration (if it exists), but rather to determine if our chosen parameters are able to comply with the project's requirements.

3.7.1 SLAM Performance

To evaluate the performance of SLAM, different configurations of the simulated lidar were tested. Since our simulations run on a virtual machine with limited RAM, we needed to take processing power into consideration. With this in mind, the goal was to find a value for the range of the lidar that was sufficient for navigating in an indoor environment and required relatively low computational power to simulate. Afterwards, when an adequate lidar range had been determined, parameter tuning could be performed on the chosen configuration. Parameter tuning would then be performed using a mostly heuristic approach using the initial configuration as a starting point.

3.7.2 Navigation2 Performance

To test the Navigation2 performance and check whether the requirements of the project were met, we conducted three different tests:

- Trajectory matching test
- Replan
- Collision

The *Trajectory matching test* is based on computing the mean distance and standard deviation from the desired destination and vehicle when moving from point A to B while avoiding obstacles. The test also showed the average distance from the vehicle to the walls, obstacles, and the goal at selected points in Figure 13.

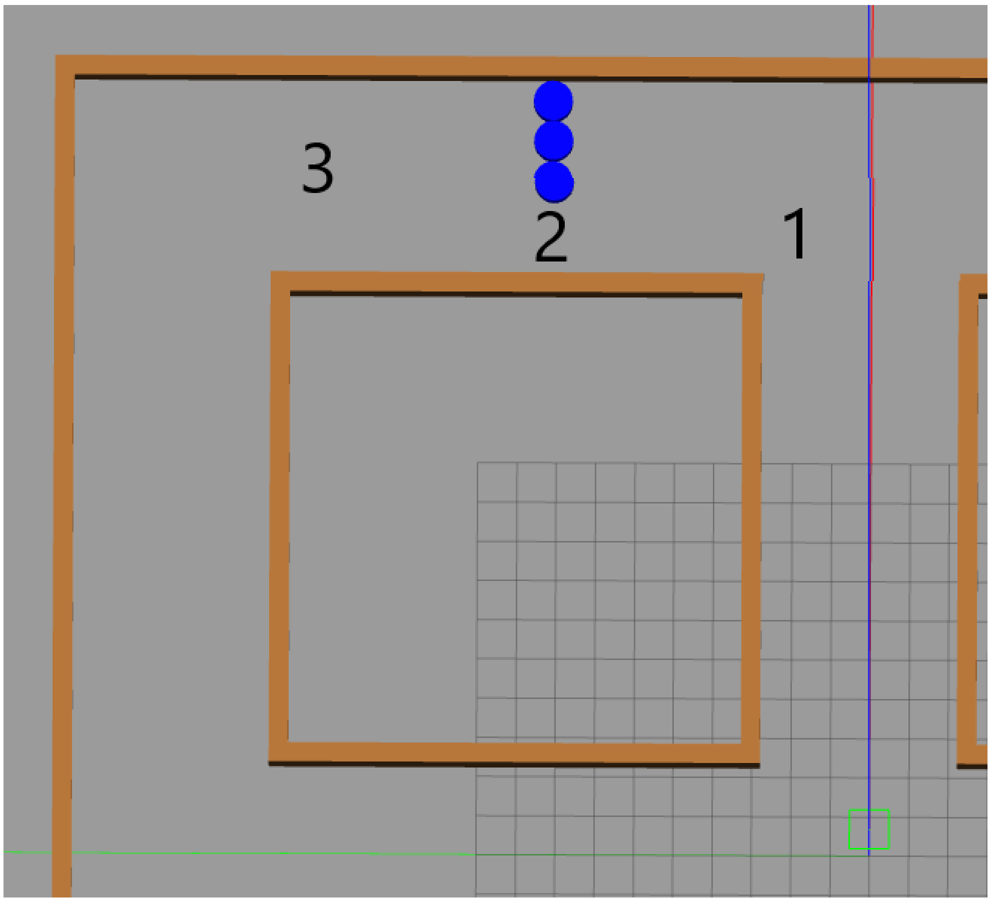


Figure 13 – Visualization of the three points where measurements will be taken of the position of the vehicle in relation to either obstacles or the final goal.

Five simulation runs were performed, where the goal was set at point number 3 in Figure 13. Meanwhile, the distance from the vehicle to the closest obstacle was measured at points number 2 and 1. The distances were measured from the center of the vehicle. As well as distances to the goal and obstacles, the time of each run was measured. The model ran at 1 m/s with an acceleration of 0.5 m/s.

The *replan* test focused on determining the shortest distance to an obstacle at which the vehicle recognized the need to replan its trajectory and successfully navigate around the obstacle. The testing was performed by adding an obstacle in the planned path at different distances from the lidar sensor located at the front of the vehicle, see Figure 19. If the model was able to pass, the closest distance to the obstacle was measured.

The *collision* test builds upon the results in the *replan* test, distances shorter than the shortest distance avoidable in the *replan* test were tested here. In the *collision* test, we tested the brake distance for the model by adding an obstacle at different distances in front of the vehicle. The longest distance was 1 meter and then it was incrementally lowered by 0.1 meters until the vehicle had a collision with the obstacle.

4 Results and Analysis

This section aims to describe the outcome of the experiments outlined in the methodology section, as well as the overall development of the project.

4.1 SLAM Performance

Different settings and parameters were tested during the implementation part of the project to optimize performance. One significant change that was made was reducing the range of the lidar. The full list of parameters and their specific values can be found in Appendix C.

By decreasing the lidar range, the amount of data that needed to be processed was reduced. This resulted in faster processing speeds and more efficient operation of the SLAM algorithm, which made it easier to drive around, map up the environment, and improve the overall performance. The 15-meter distance was found to be sufficient for our application by incrementally decreasing the 30-meter range by 5 meters and comparing the performance. 15 meters provided a good balance between processing speed and the ability to detect obstacles at a reasonable distance considering the maximum speed of the vehicle being 1 m/s.

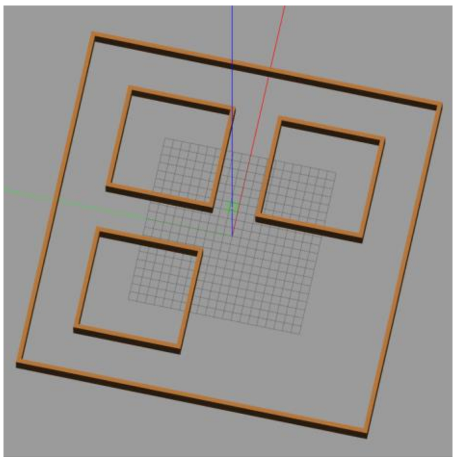


Figure 14 - Demo map in Gazebo

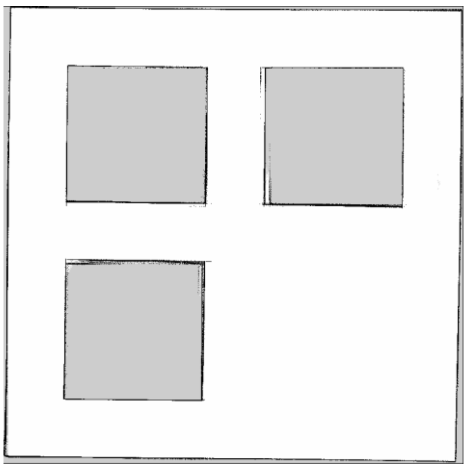


Figure 15 - Gazebo-map produced by SLAM

4.2 Navigation2 Performance

This section will showcase the results of the experiments described in 3.7.2

Upon completion of the project, the model can move from point A to B while considering the obstacles in its path. As shown in the experiments, the model can plan an initial path to the goal, as well as identify obstacles and replan in case of obstruction.

In terms of obstacle avoidance and path planning, some features were more important than others, these being:

Global cost map: 2D grid representation of the robot's environment used for long-term planning. The cost map assigns a cost to each cell in the map based on the likelihood of a collision with an obstacle.

Cost_travel_multiplier: Used to determine how much the model should steer away from high-cost areas.

Global footprint: represents the physical shape of the model and is used for global planning.

We wanted the model to be able to avoid obstacles but not steer away from paths that were possible to take. To do this, the global cost map inflation radius was increased from 0.55 to 1, causing the global planner to plan further away from obstacles. The *"cost_scaling_factor"* was decreased from 3.0 to 2.0, which makes it easier for the planner to plan through higher-cost areas if needed. Furthermore, the global footprint was set to match the car's shape, with minimal padding, which enables planning closer to the obstacles. Combining these three changes allows the model to take the safer path when possible but allows for paths closer to obstacles when needed. These values were produced by trial and error and changed until we were satisfied with the result.

In the documentation for Nav2 regarding the tuning of the parameters, the chosen value of 2.0 for *Cost_travel_multiplier* is described as a tradeoff value when deciding whether to prioritize tight turns close to an obstacle or wide turns to avoid a collision.

The discussed changes have been performed by changing parameters in a configuration file for Nav2. No programming has been made that directly affects the navigation, rather nodes and scripts that enable the car to move and locate.

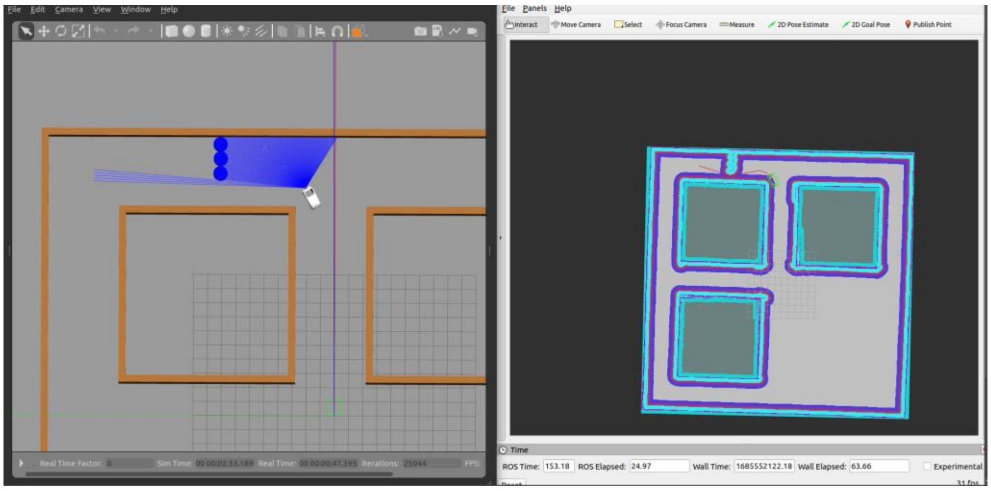
4.2.1 Trajectory matching test

The test was performed as described in section 3.7.2; the test generated the following results.

Table 1: Mean and standard deviation calculated from measurements from the center of the vehicle. Measurements from each simulation run can be found in appendix E.

	Vehicle position turn (m) (1)	Vehicle position obstacle (m) (2)	Vehicle position destination (m) (3)	Time (s)
Mean	1,10	0,63	0,28	32
Standard Deviation	0,27	0,07	0,10	1,87

The results show that the parameter tuning discussed in the previous section had a noticeable effect. The car prefers to navigate further from obstacles, as seen in the measurements from point 1. However, the measurements from point 2 shows that it is capable of navigating closer to obstacles if needed.

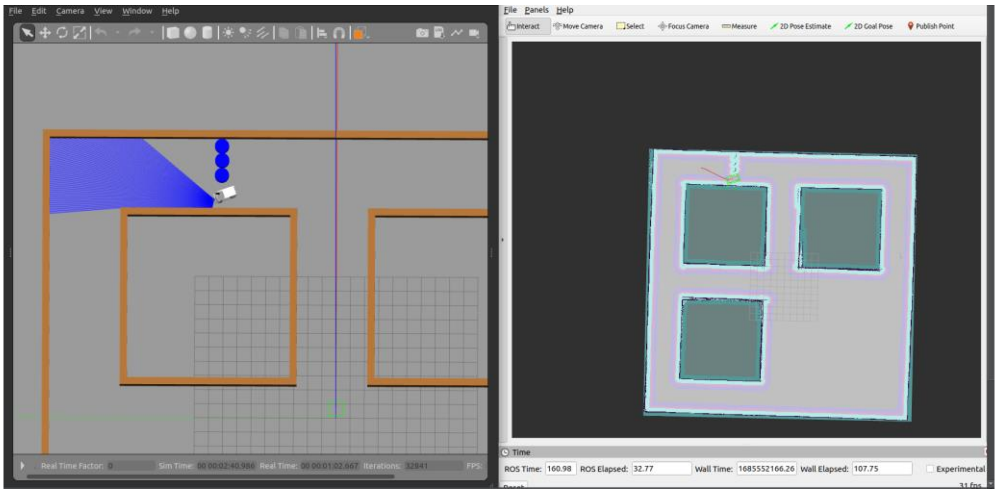


Real Time Factor: 0 Sim Time: 00 00:02:33.189 Real Time: 00 00:00:47.395 Iterations: 25044 FPS:

ROS Time: 153.18 ROS Elapsed: 24.97 Wall Time: 1685552122.18 Wall Elapsed: 63.66 Experimental

31 fps

Figure 16 – Measurement point 1. Measuring the distance from the center of the vehicle to the closest obstacle, which in this case is the wall. The planned path to the goal is visualized in the right window of the picture.

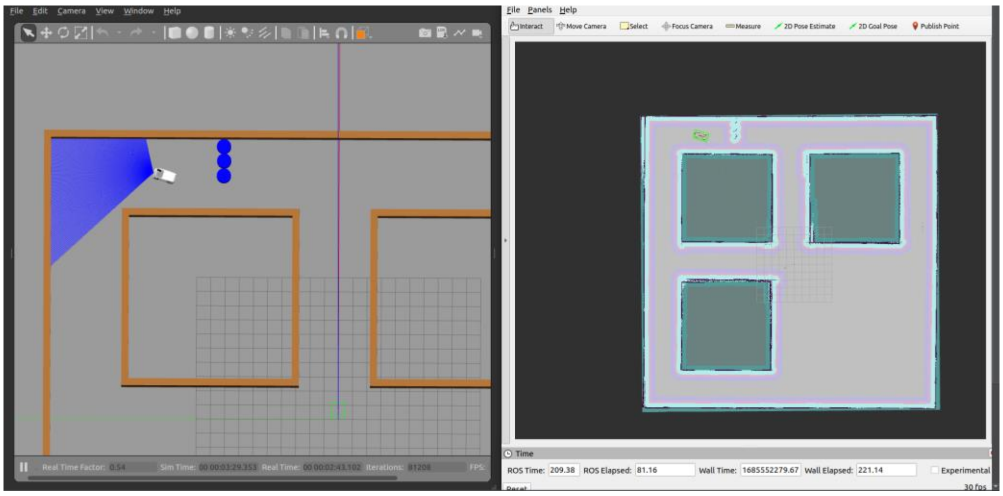


Real Time Factor: 0 Sim Time: 00 00:02:40.986 Real Time: 00 00:01:02.667 Iterations: 32841 FPS:

ROS Time: 160.98 ROS Elapsed: 32.77 Wall Time: 1685552166.26 Wall Elapsed: 107.75 Experimental

31 fps

Figure 17 – Measurement point 2. Measuring the distance from the center of the vehicle to the closest obstacle, either the cylindrical obstacles or the wall. The planned path is visualized in the right window of the picture.



Real Time Factor: 0.54 Sim Time: 00 00:03:29.353 Real Time: 00 00:02:43.102 Iterations: 81208 FPS:

ROS Time: 209.38 ROS Elapsed: 81.16 Wall Time: 1685552279.67 Wall Elapsed: 221.14 Experimental

30 fps

Figure 18 – Measurement point 3. Measuring the distance from the initial goal to the actual position of the vehicle. The distance here is measured from the center of the vehicle as well.

4.2.2 Replan

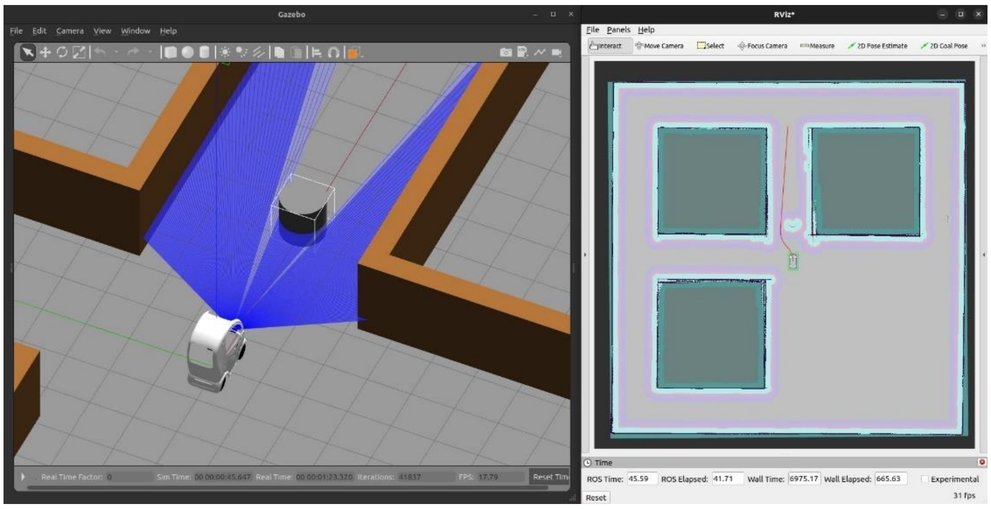
The replan test, shown in Figure 19 was focused on determining the minimum distance to an obstacle at which the vehicle recognized the need to replan its trajectory and successfully navigate around the obstacle.

As shown in Table 2, at the distance of 5 meters from the front of the vehicle to the obstacle, the vehicle successfully replanned the route and reached the goal. The closest distance between the vehicle and the obstacle was 1.14m.

Similarly, at 4 meters, the vehicle successfully replanned the path and reached the goal with the closest distance of 1.43m to the obstacle. Likewise, as the distance to the obstacle decreased to 3 meters, the vehicle replanned its path and avoided the obstacle with the closest distance of 0.90m to the target.

However, when the distance was reduced to 2.5 meters and below, the vehicle was unable to replan and execute the path to avoid the obstacle, which resulted in the vehicle stopping.

These results show that the vehicle successfully recognized the need to replan its trajectory and navigate around the obstacle when the obstacle distance was 5, 4, and 3 meters. But, at distances of 2.5 meters and below, the vehicle failed to replan the path. This is probably due to the vehicle's mobility constraints, the turning radius of 2.35 meters combined with the speed and effectiveness of the replanning algorithm not being optimal.



Real Time Factor: 0 Sim Time: 00 00:00:45.647 Real Time: 00 00:01:23.320 Iterations: 41837 FPS: 17.79

ROS Time: 45.59 ROS Elapsed: 41.71 Wall Time: 6975.17 Wall Elapsed: 665.63

31 fps

Figure 19 – Replan test with 3 m distance to the obstacle in Gazebo (Left) and Rviz (Right)

Table 2: Test results of the replanning test where the distance to the obstacle is measured from the lidar located at the front, closest distance while passing is measured from the center of the vehicle.

Distance to Obstacle (m)	Closest distance while passing (m)	Target reached
5	1,14	Yes
4	1,43	Yes
3	0,90	Yes
2,5	N/A	No
2	N/A	No

4.2.3 Collision

The collision test aimed to evaluate the brake distance of the model by introducing obstacles at different distances in front of the vehicle. As can be seen in Table 3, the vehicle managed to stop without a collision with distances over 60cm from the vehicle to the obstacle. This is probably due to the deacceleration value of 0.5 m/s, a value which was chosen by taking into account the velocity of 1 m/s. The update frequency of the costmap could also be an issue because it provides information about the available paths and ensures that the robot has updated information about obstacles and available free space.

Table 3: The results from the collision test, the distance to the obstacle is measured from the lidar sensor which is located at the front of the vehicle.

Distance to Obstacle (m)	Remaining distance (m)	Crash (Yes/No)
1	0,27	No
0,9	0,17	No
0,8	0,14	No
0,7	0,06	No
0,6	0	Yes

5 Discussion

This section introduces a discussion on some of the challenges faced during the development of the project, as well as the limitations regarding the accuracy and scope of the finalized vehicle model.

5.1 SLAM Performance

The result of the experiment resulted in a 2D-lidar configuration and range of 15m, with computational requirements in mind. The selected configuration is specified to run on the limited system that we ran the simulation on and can be changed in case of access to more processing power and a need for more range. The parameters of SLAM were tuned according to the lidar range. During development, it was found that the odometry of the vehicle had a bigger impact on localization when mapping with SLAM than the parameters chosen.

5.2 Navigation Performance

5.2.1 Trajectory matching test

As discussed briefly in the results section, the model has a preferred distance to obstacles when navigating, but, if necessary, it can navigate closer to the obstacles to reach the goal. On average, the model missed the target by 0.28 meters, with a standard deviation of 0.1 meters which in our case is an acceptable distance. This result was expected since the tolerance was set to 0.25 meters in the Nav2 parameters, which means that the planner will consider the model to have reached the goal when in this range. Reducing the tolerance could improve the results, however, it would put higher demands on the other parameters to make sure that the navigation is accurate enough. If the tolerance is too low, the model wouldn't consider itself to have reached the goal, despite being at an acceptable distance considering the size of the vehicle.

5.2.2 Replan

The replanning test was a test of both the ability to navigate past an obstacle as well as a test of the mobility constraints of the robot. One issue encountered here was that, since two paths were available to pass the object with similar lengths and cost, the planner sometimes alternated between them. This caused the robot to delay the avoidance maneuver. This can be seen in the results, where it would be expected that the distance to the obstacle would be greater or at least similar for the simulation run with a 5-meter distance to the obstacle compared to the one with a 4-meter distance. This issue could potentially be avoided by further tuning the parameters, with emphasis on the replanning frequency. This would allow the planner to choose one path and by the time replanning is considered, the paths are no longer the same length, causing one of them to be chosen instead of alternating.

5.2.3 Collision

The results of the collision test show that the vehicle was able to stop and avoid collisions at distances of 1 meter, 0.9 meters, 0.8 meters, and 0.7 meters. At a distance of 0.6 meters, the vehicle was unable to stop before colliding with the obstacle.

Several different factors could contribute to this distance. Like said before, the deacceleration of 0.5 m/s could potentially be an issue. By increasing this the vehicle should be able to come to a stop faster. Additionally, friction between the tires and the ground could also influence the stopping distance.

The update frequency of the costmap could also contribute to the stopping distance because of its key role to create cost areas around obstacles. If the frequency is set too low, the vehicle may not be aware of a suddenly appeared obstacle. To address this issue, it is important to ensure that the frequency is set properly to match the environment and the vehicles dynamics.

However, increasing the update frequency of the costmap means increasing load on processing power which could put a strain on the system. Therefore, it is necessary to find a good balance between the update frequency and controller frequency.

5.2.4 Comparative Analysis

In [54] the authors introduced a methodology aimed at acting as a guide for the necessary steps required to deploy Navigation2 on an existing custom-built robot operating with ROS2. The Navigation2 configuration was tested in a virtual environment, Gazebo, which was modeled to replicate the real-world environment.

To evaluate the transferability of the setup, the authors of the paper ran a similar test on both a real-life robot as well as a simulated one in Gazebo. The two tests in both a simulated environment and a real environment demonstrated that the same Navigation2 configuration resulted in the same maneuvers being carried out. Furthermore, in this project we have followed roughly the same steps highlighted in [54] and the main differences are mostly in the configuration parameters which are highly dependent upon the actual vehicle one desires to simulate. This shows that transferring and implementing a Nav2 configuration into a real physical system is entirely plausible and in fact has already been achieved by following roughly the same methodology. Thus, we can conclude that we have proven the feasibility of automating the real electric vehicle.

5.3 Limitations

Although the project produced a working virtual model that can move from point A to B while avoiding obstacles, the model still has some limitations worth noting.

As mentioned previously in the report, the model cannot reverse, which means that the car's mobility is somewhat limited. The reason behind not allowing the car to reverse was that the project was supplied with one physical lidar aimed to be mounted at the front. This meant it would only be able to detect objects in front of the car. Due to this, it was thought that, from a safety standpoint, it was reasonable not to allow the car to reverse.

Not allowing reversing also brings another limitation to the performance. If the model was to deviate from the target when close to the goal, it cannot correct it. So, either it must find a way to loop around to reach the target or the goal tolerance of the navigation stack can be increased to allow more deviation from the final goal.

Another limitation of the model is maneuvering in open space without obstacles or walls nearby. Due to the range being set at 15m, the SLAM will not be provided with any lidar scans if there are no obstacles within a radius of 15m from the sensor. This will cause the model to localize only with the help of odometry, which isn't perfect. This causes some confusion for the planner at times and leads to a decrease in precision.

Currently, the model can only navigate in a previously mapped environment. However, this does not pose much of an issue since the vehicle is intended to be used in closed areas.

The model would need complementing to be used as a base for the real vehicle, as several integral parts would need changing to control it:

Steering the robot has been performed by sending commands directly to the joints, as opposed to the real vehicle, which will be controlled by actuators. To further replicate the real vehicle, these actuators would need to be simulated as well.

The model is only configured to map and navigate using a 2D lidar, which is not what will be used in the vehicle. This was a way of saving time due to the quicker configuration regarding the limited time of the project, as well as saving processing power.

5.4 Challenges

As described previously in the report, the project was supplied with a virtual box containing ROS2 Humble, which is the latest release of ROS2. While this has advantages in terms of longevity, it also presents some challenges.

One advantage of it is the latest release is that it is likely to have a longer life span than the older distributions. This is good, considering the community of the older distributions may become inactive and stop providing updates to packages.

One issue with this, however, is that all the necessary packages for performing our project are not yet developed for Humble. Previous distributions offer support for an Ackermann controller, which makes steering and odometry calculating a lot easier for the user. The Ackermann controller will be provided to ROS2 Humble in due time, so the configuration of this project might seem outdated by then.

Overall, our experience using ROS2 Humble highlighted both the advantages and disadvantages of using the latest ROS2 distribution. While it offers longevity and access to up-to-date features, it may not always be the best choice, depending on the type of project that is to be conducted.

5.5 Sustainable Development

This final section aims to analyze the project from different sustainability perspectives.

Economical

Performing virtual testing is considered cheaper since it is less time-consuming and brings a lower risk of any hardware being damaged. Since the testing can be performed virtually, there is no need to transport the vehicle or the people working on it.

Environmental

As discussed in the economic part, virtual testing decreases the transport necessary and eventual hardware reparations. Furthermore, creating a virtual environment and model enables the developers to try different configurations without needing to order new parts.

Safety

The biggest safety risk when developing autonomous cars is that the vehicle performs in an erratic and unexpected way. Virtually simulating the performance before reduces this risk. However, it is a risk to assume that the vehicle performs exactly like the virtual model since a lot of parameters and conditions need to be simulated correctly.

6 Conclusion

This project explored the virtualization of an electric vehicle, and collaboration with the company has allowed the authors to design, develop, and test this virtual avatar of the car.

This thesis work has proven that although it isn't ideal to create a custom control setup for an Ackermann steering robot, it is possible in a simulated environment. Through custom-written nodes to convert messages to the right message types as well as updating the odometry from wheel speeds and positions, a working autonomous model was created.

The developed model was able to achieve the goals set out in the beginning of the work, by navigating from point A to B while simultaneously avoiding obstacles. The results proved that navigation was performed with an error margin of 0.28 meters, which is deemed a success considering the size of the vehicle. Furthermore, if avoiding an obstacle was deemed impossible, the car would come to a stop if the obstacle appeared more than 0.6 meters away from the lidar sensor.

Although the model has some limitations, including the inability to reverse, due to safety constraints, and maneuver in open spaces without obstacles or walls nearby, it provides a solid starting point for future work.

For potential future work, the authors recommend using a previous distribution of ROS2 or awaiting further controller support for Ackermann-steer robots. In addition, to enable the vehicle reverse drive, it is recommended to incorporate a lidar or RGB camera located at the rear.

The identified areas for future development present promising opportunities for significant advancements in this field. Moreover, the assumption that the virtual setup will translate into real-life scenarios is supported by the work conducted by [54]. Their successful implementation of Nav2 on a real robot showcases the potential practical application of our virtual testing setup.

Overall, this project has demonstrated the feasibility of virtual testing for autonomous vehicles and has identified several areas for potential future development, which could lead to significant advancements in this field.

7 References

1. Takaya, K., Asai, T., Kroumov, V., & Smarandache, F. (2016, October). Simulation environment for mobile robots testing using ROS and Gazebo. In *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)* (pp. 96-101). IEEE.
2. Semcon, "About Us." https://semcon.com/uk/about-us/. Accessed 31 January 2023.
3. Yao, S., Zhang, J., Hu, Z., Wang, Y., & Zhou, X. (2018). Autonomous-driving vehicle test technology based on virtual reality. The Journal of Engineering, 2018(16), 1768-1771.
4. Shabalina, K., Sagitov, A., Su, K. L., Hsia, K. H., & Magid, E. (2019). Avrora unior car-like robot in gazebo environment. In *International Conference on artificial life and robotics* (pp. 116-119).
5. S. Fu, C. Zhang, W. Zhang and X. Niu, "Design and Simulation of Tracked Mobile Robot Path Planning," 2021 IEEE 4th International Conference on Big Data and Artificial Intelligence (BDAI), Qingdao, China, 2021, pp. 86-90, doi: 10.1109/BDAI52447.2021.9515251.
6. Shimchik, I., Sagitov, A., Afanasyev, I., Matsuno, F., & Magid, E. (2016). Golf cart prototype development and navigation simulation using ROS and Gazebo. In *MATEC Web of Conferences* (Vol. 75, p. 09005). EDP Sciences.
7. A. AbdelHamed, G. Tewolde and J. Kwon, "Simulation Framework for Development and Testing of Autonomous Vehicles," 2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS), Vancouver, BC, Canada, 2020, pp. 1-6, doi: 10.1109/IEMTRONICS51293.2020.9216334.
8. S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," Science Robotics vol. 7, May 2022.
9. S. Macenski, F. Martín, R. White, J. Clavero. The Marathon 2: A Navigation System. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2020.
10. Blimo Kabinscooter Plus. https://www.blimo.se/elfordon/promenadscooter/blimo-kabinscooter-plus Accessed 6 March 2023
11. Xacro Albergo, N., Rathi, V., & Ore, J. P. (2022, May). Understanding Xacro Misunderstandings. In *2022 International Conference on Robotics and Automation (ICRA)* (pp. 6247-6252). IEEE.
12. Daneshmand, M., Helmi, A., Avots, E., Noroozi, F., Alisinanoglu, F., Arslan, H. S., ... & Anbarjafari, G. (2018). 3d scanning: A comprehensive survey. *arXiv preprint arXiv:1801.08863*.

13. D. V. Nam and K. Gon-Woo, "Solid-State LiDAR based-SLAM: A Concise Review and Application," 2021 IEEE International Conference on Big Data and Smart Computing (BigComp), Jeju Island, Korea (South), 2021, pp. 302-305, doi: 10.1109/BigComp51126.2021.00064.
14. Zhang, H., Zhang, Y., Liu, C., & Zhang, Z. (2023). Energy efficient path planning for autonomous ground vehicles with ackermann steering. *Robotics and Autonomous Systems*, 104366.
15. Eisele R. Ackerman Steering • Open Source is Everything [Internet]. Xarg.org. 2019. Available from: https://www.xarg.org/book/kinematics/ackerman-steering/
16. Friedrich, B. (2016). The effect of autonomous vehicles on traffic. Autonomous Driving: Technical, Legal and Social Aspects, 317-334.
17. Faisal, A., Kamruzzaman, M., Yigitcanlar, T., & Currie, G. (2019). Understanding autonomous vehicles. Journal of transport and land use, 12(1), 45-72.
18. Robot Operating System Documentation, "Understanding Nodes." https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html . Accessed 30 January 2023.
19. Robot Operating System Documentation, "Understanding Topics." https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html. Accessed 30 January 2023.
20. Robot Operating System Documentation, "Understanding Services." https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html. Accessed 30 January 2023.
21. Robot Operating System Documentation, "Understanding Actions." https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html. Accessed 30 January 2023.
22. Bakken, D. (2001). Middleware. Encyclopedia of Distributed Computing, 11.
23. Chitta, S., Marder-Eppstein, E., Meeussen, W., Pradeep, V., Tsouroukdissian, A. R., Bohren, J., ... & Perdomo, E. F. (2017). ros_control: A generic and simple control framework for ROS. *The Journal of Open Source Software*, *2*(20), 456-456.
24. Nourani-Vatani, N., Roberts, J., & Srinivasan, M. V. (2008, January). IMU aided 3D visual odometry for car-like vehicles. In *Proceedings of the 2008 Australasian Conference on Robotics and Automation, ACRA 2008* (pp. 1-8). Australian Robotics and Automation Association.

25. Rico, F. M. (2022). *A Concise Introduction to Robot Programming with ROS2*. CRC Press.
26. ROS Index [Internet]. index.ros.org. [cited 2023 May 8]. Available from: https://index.ros.org/p/robot_state_publisher/
27. ROS Index [Internet]. index.ros.org. [cited 2023 May 8]. Available from: https://index.ros.org/p/joint_state_publisher/
28. M. G. Ocando, N. Certad, S. Alvarado and Á. Terrones, "Autonomous 2D SLAM and 3D mapping of an environment using a single 2D LIDAR and ROS," *2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR)*, Curitiba, Brazil, 2017, pp. 1-6, doi: 10.1109/SBR-LARS-R.2017.8215333.
29. Macenski, S., & Jambrecic, I. (2021). SLAM Toolbox: SLAM for the dynamic world. Journal of Open Source Software, 6(61), 2783.
30. Kohlbrecher, S., Von Stryk, O., Meyer, J., & Klingauf, U. (2011, November). A flexible and scalable SLAM system with full 3D motion estimation. In 2011 IEEE international symposium on safety, security, and rescue robotics (pp. 155-160). IEEE.
31. Grisetti, G., Stachniss, C., & Burgard, W. (2007). Improved techniques for grid mapping with rao-blackwellized particle filters. IEEE transactions on Robotics, 23(1), 34-46.
32. A. Mohan and A. R. Krishnan, "Design and Simulation of an Autonomous Floor Cleaning Robot with Optional UV Sterilization," *2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*, Mysuru, India, 2022, pp. 1-6, doi: 10.1109/MysuruCon55714.2022.9972558.
33. Robot Operating System Documentation, "Overview and usage or RQt" http://docs.ros.org/en/humble/Concepts/About-RQt.html Accessed 6 March 2023
34. Chen, H. (2022). Development of teaching material for Robot Operating System (ROS): creation and control of robots.
35. Cyberbotics., "Introduction to Webots," Cyberbotics Ltd., [Online]. Available from: https://cyberbotics.com/doc/guide/introduction-to-webots. Accessed 7 May 2023.
36. Coppelia Robotics. CoppeliaSim [Internet]. [cited 07-05-2023]. Available from: https://www.coppeliarobotics.com/
37. Echeverria, G., Lassabe, N., Degroote, A., & Lemaignan, S. (2011, May). Modular open robots simulation engine: Morse. In *2011 ieee international conference on robotics and automation* (pp. 46-51). IEEE.
38. Farley, A., Wang, J., & Marshall, J. A. (2022). How to pick a mobile robot simulator: A quantitative comparison of CoppeliaSim, Gazebo, MORSE and Webots with a focus on accuracy of motion. *Simulation Modelling Practice and Theory*, *120*, 102629.

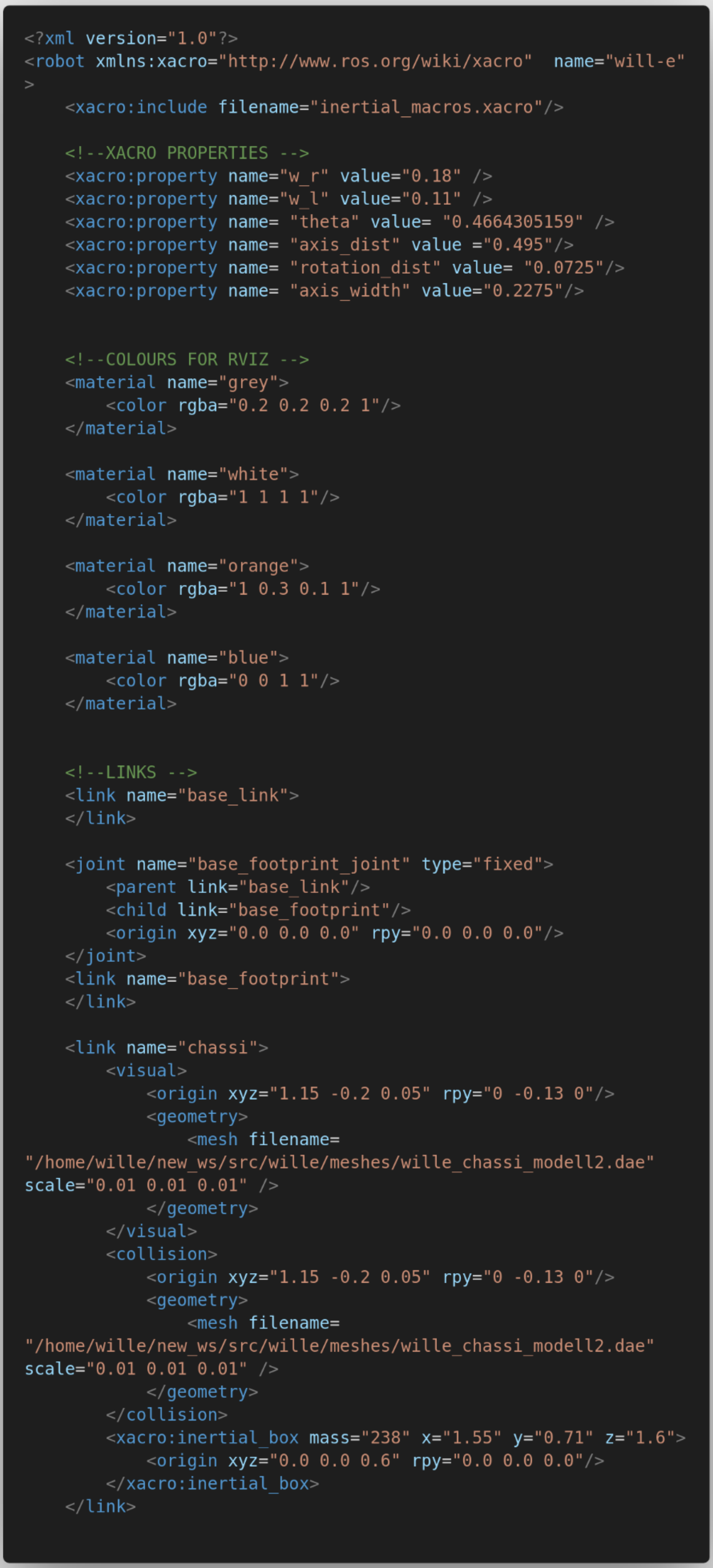
39. S. Teixeira, R. Arrais and G. Veiga, "Cloud Simulation for Continuous Integration and Deployment in Robotics," 2021 IEEE 19th International Conference on Industrial Informatics (INDIN), Palma de Mallorca, Spain, 2021, pp. 1-8, doi: 10.1109/INDIN45523.2021.9557476.
40. Kortelainen, M. (2023). A short guide to ROS 2 Humble Hawksbill.
41. Microsoft. Visual Studio Code [Internet]. Visualstudio.com. Microsoft; 2016. Available from: https://code.visualstudio.com/
42. GitHub. GitHub [Internet]. GitHub. Available from: https://github.com/
43. 22.04 LTS [Internet]. Ubuntu. [cited 2023 May 8]. Available from: https://ubuntu.com/blog/tag/22-04-lts
44. ROS Index [Internet]. index.ros.org. [cited 2023 May 8]. Available from: https://index.ros.org/p/teleop_twist_keyboard/github-ros2-teleop_twist_keyboard/
45. De Rose, M. (2021). *LiDAR-based Dynamic Path Planning of a mobile robot adopting a costmap layer approach in ROS2* (Doctoral dissertation, Politecnico di Torino).
46. Zhang, X., Lai, J., Xu, D., Li, H., & Fu, M. (2020). 2d lidar-based slam and path planning for indoor rescue using mobile robots. *Journal of Advanced Transportation*, *2020*, 1-14.
47. Jiang, S., Wang, S., Yi, Z., Zhang, M., & Lv, X. (2022). Autonomous Navigation System of Greenhouse Mobile Robot Based on 3D Lidar and 2D Lidar SLAM. *Frontiers in Plant Science*, *13*.
48. twist_mux - ROS Wiki [Internet]. wiki.ros.org. [cited 2023 May 5]. Available from: http://wiki.ros.org/twist_mux
49. DWB Controller — Navigation 2 1.0.0 documentation [Internet]. navigation.ros.org. [cited 2023 Jun 3]. Available from: https://navigation.ros.org/configuration/packages/configuring-dwb-controller.html
50. Regulated Pure Pursuit — Navigation 2 1.0.0 documentation [Internet]. navigation.ros.org. [cited 2023 Jun 3]. Available from: https://navigation.ros.org/configuration/packages/configuring-regulated-pp.html
51. Tuning Guide — Navigation 2 1.0.0 documentation [Internet]. navigation.ros.org. Available from: https://navigation.ros.org/tuning/index.html
52. NavFn Planner — Navigation 2 1.0.0 documentation [Internet]. navigation.ros.org. [cited 2023 Jun 3]. Available from: https://navigation.ros.org/configuration/packages/configuring-navfn.html
53. Smac Hybrid-A* Planner — Navigation 2 1.0.0 documentation [Internet]. navigation.ros.org. [cited 2023 Jun 3]. Available from:

https://navigation.ros.org/configuration/packages/smac/configuring-smac-hybrid.html

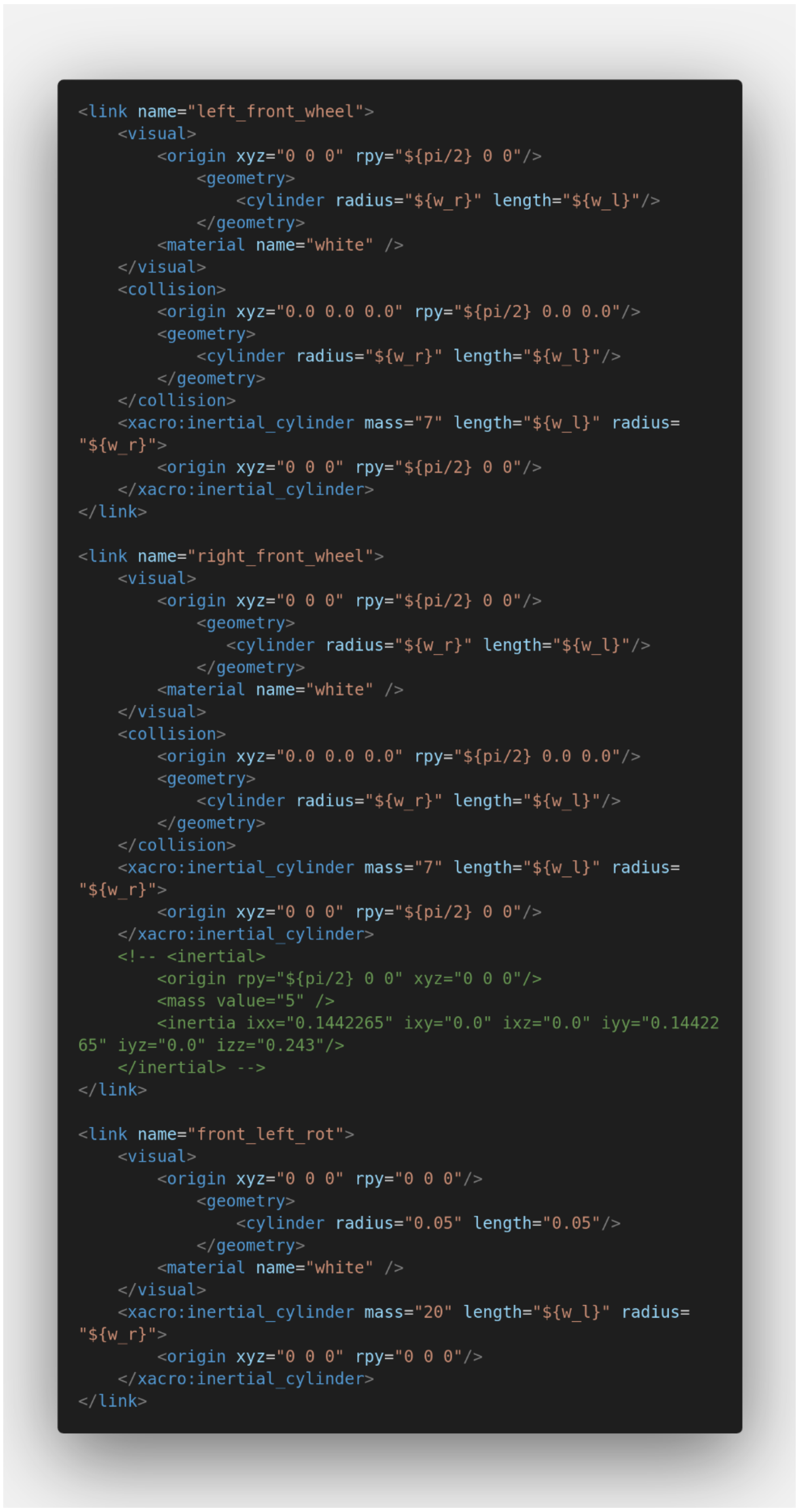
54. Horelican, T. (2022). Utilizability of Navigation2/ROS2 in Highly Automated and Distributed Multi-Robotic Systems for Industrial Facilities. IFAC-PapersOnLine, 55(4), 109-114.

8 Appendices

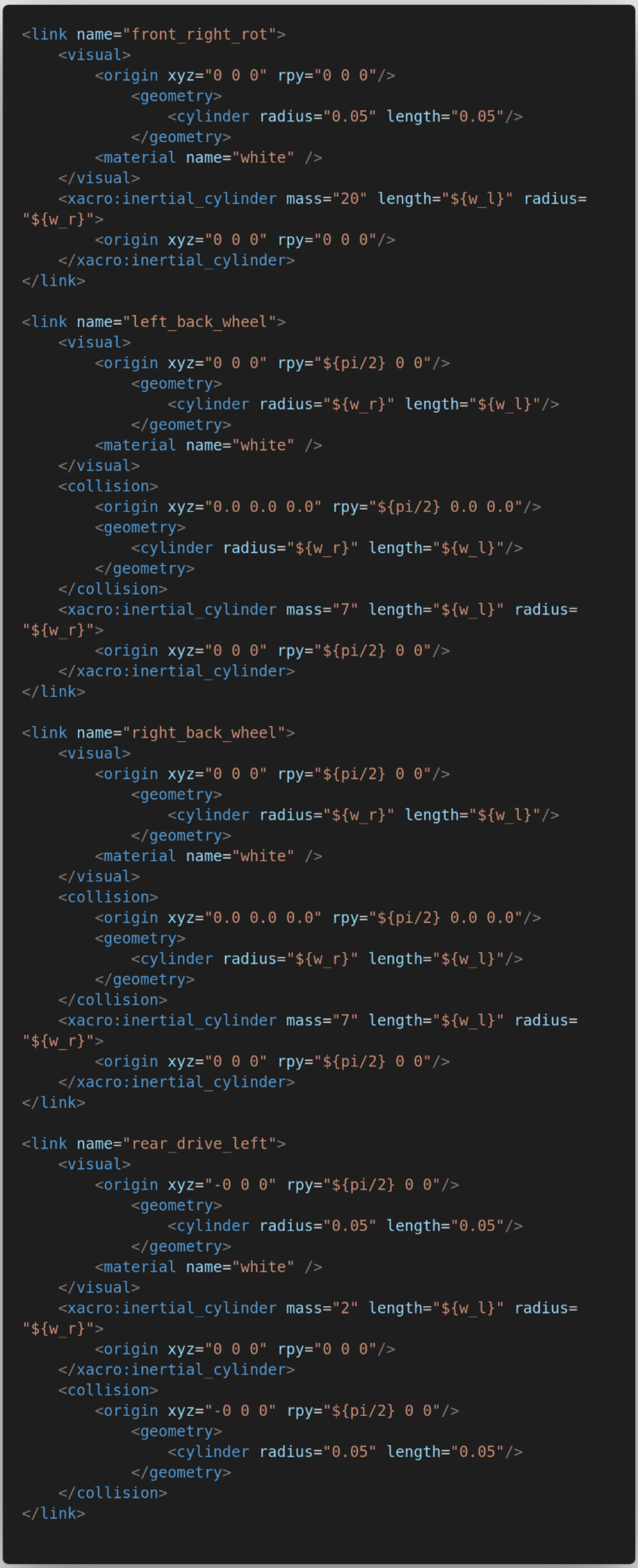
Appendix A – URDF code describing the vehicle



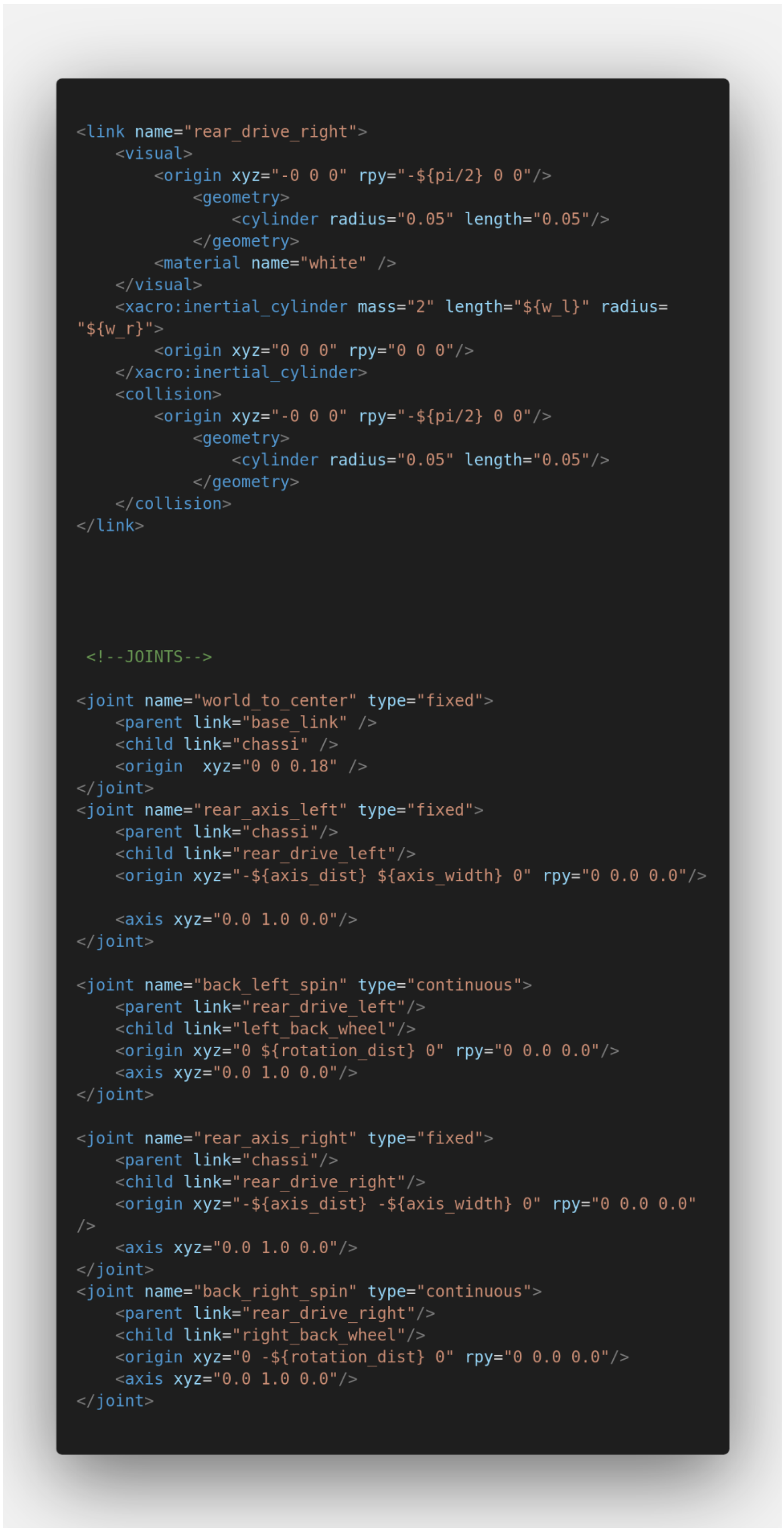
```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro"  name="will-e"
>
    <xacro:include filename="inertial_macros.xacro"/>

    <!--XACRO PROPERTIES -->
    <xacro:property name="w_r" value="0.18" />
    <xacro:property name="w_l" value="0.11" />
    <xacro:property name= "theta" value= "0.4664305159" />
    <xacro:property name= "axis_dist" value ="0.495"/>
    <xacro:property name= "rotation_dist" value= "0.0725"/>
    <xacro:property name= "axis_width" value="0.2275"/>


    <!--COLOURS FOR RVIZ -->
    <material name="grey">
        <color rgba="0.2 0.2 0.2 1"/>
    </material>

    <material name="white">
        <color rgba="1 1 1 1"/>
    </material>

    <material name="orange">
        <color rgba="1 0.3 0.1 1"/>
    </material>

    <material name="blue">
        <color rgba="0 0 1 1"/>
    </material>


    <!--LINKS -->
    <link name="base_link">
    </link>

    <joint name="base_footprint_joint" type="fixed">
        <parent link="base_link"/>
        <child link="base_footprint"/>
        <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
    </joint>
    <link name="base_footprint">
    </link>

    <link name="chassi">
        <visual>
            <origin xyz="1.15 -0.2 0.05" rpy="0 -0.13 0"/>
            <geometry>
                <mesh filename=
"/home/wille/new_ws/src/wille/meshes/wille_chassi_modell2.dae"
scale="0.01 0.01 0.01" />
            </geometry>
        </visual>
        <collision>
            <origin xyz="1.15 -0.2 0.05" rpy="0 -0.13 0"/>
            <geometry>
                <mesh filename=
"/home/wille/new_ws/src/wille/meshes/wille_chassi_modell2.dae"
scale="0.01 0.01 0.01" />
            </geometry>
        </collision>
        <xacro:inertial_box mass="238" x="1.55" y="0.71" z="1.6">
            <origin xyz="0.0 0.0 0.6" rpy="0.0 0.0 0.0"/>
        </xacro:inertial_box>
    </link>
```



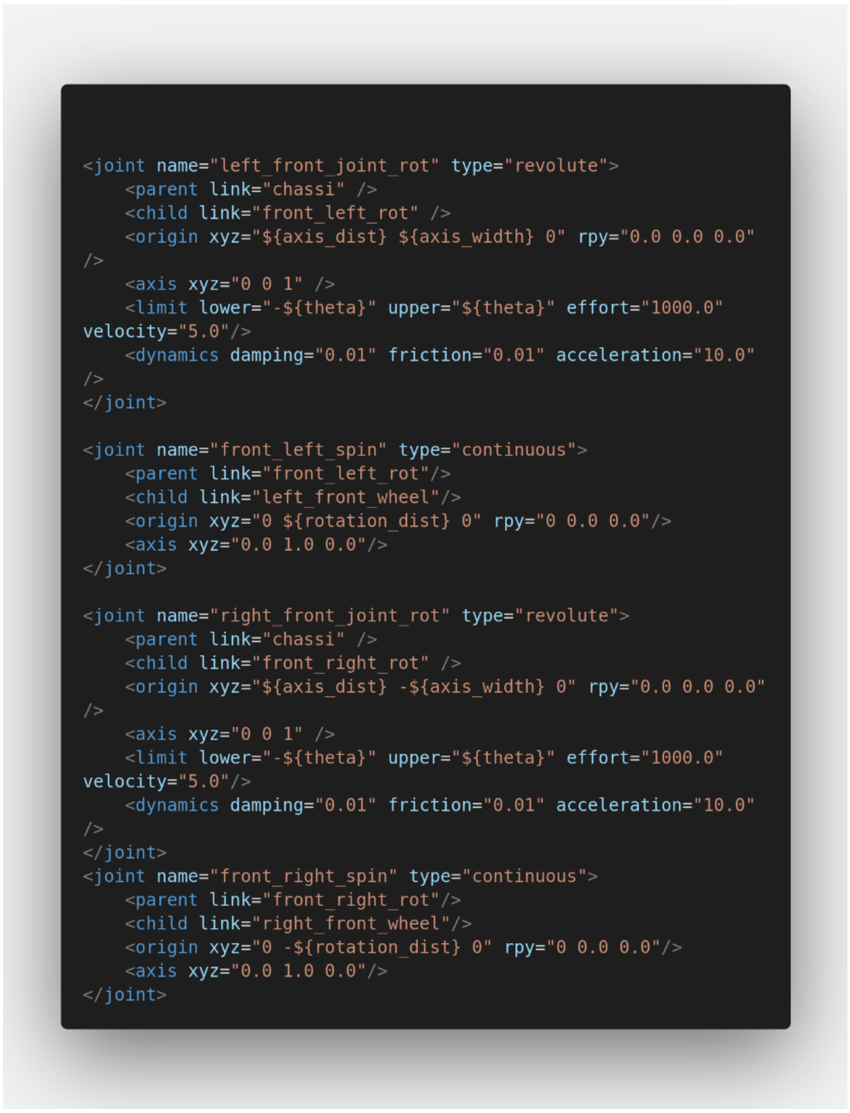
```xml
<link name="left_front_wheel">
    <visual>
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
            <geometry>
                <cylinder radius="${w_r}" length="${w_l}"/>
            </geometry>
        <material name="white" />
    </visual>
    <collision>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0.0 0.0"/>
        <geometry>
            <cylinder radius="${w_r}" length="${w_l}"/>
        </geometry>
    </collision>
    <xacro:inertial_cylinder mass="7" length="${w_l}" radius=
"${w_r}">
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
    </xacro:inertial_cylinder>
</link>

<link name="right_front_wheel">
    <visual>
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
            <geometry>
               <cylinder radius="${w_r}" length="${w_l}"/>
            </geometry>
        <material name="white" />
    </visual>
    <collision>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0.0 0.0"/>
        <geometry>
            <cylinder radius="${w_r}" length="${w_l}"/>
        </geometry>
    </collision>
    <xacro:inertial_cylinder mass="7" length="${w_l}" radius=
"${w_r}">
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
    </xacro:inertial_cylinder>
    <!-- <inertial>
        <origin rpy="${pi/2} 0 0" xyz="0 0 0"/>
        <mass value="5" />
        <inertia ixx="0.1442265" ixy="0.0" ixz="0.0" iyy="0.14422
65" iyz="0.0" izz="0.243"/>
    </inertial> -->
</link>

<link name="front_left_rot">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
            <geometry>
                <cylinder radius="0.05" length="0.05"/>
            </geometry>
        <material name="white" />
    </visual>
    <xacro:inertial_cylinder mass="20" length="${w_l}" radius=
"${w_r}">
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </xacro:inertial_cylinder>
</link>
```



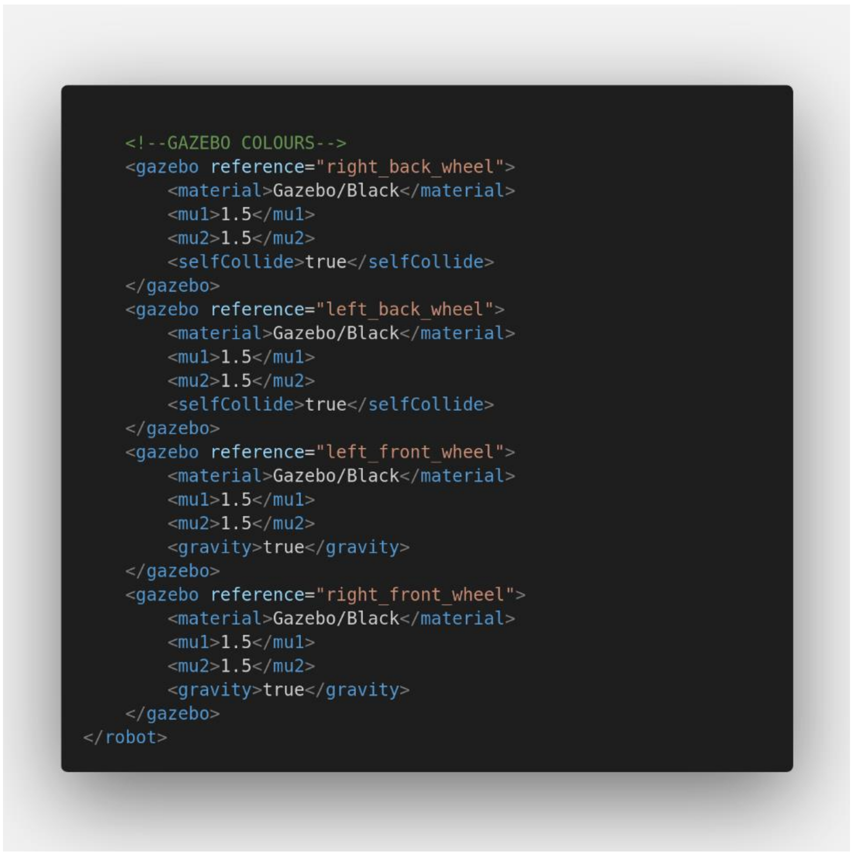
```
<link name="front_right_rot">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
            <geometry>
                <cylinder radius="0.05" length="0.05"/>
            </geometry>
        <material name="white" />
    </visual>
    <xacro:inertial_cylinder mass="20" length="${w_l}" radius=
"${w_r}">
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </xacro:inertial_cylinder>
</link>

<link name="left_back_wheel">
    <visual>
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
            <geometry>
                <cylinder radius="${w_r}" length="${w_l}"/>
            </geometry>
        <material name="white" />
    </visual>
    <collision>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0.0 0.0"/>
        <geometry>
            <cylinder radius="${w_r}" length="${w_l}"/>
        </geometry>
    </collision>
    <xacro:inertial_cylinder mass="7" length="${w_l}" radius=
"${w_r}">
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
    </xacro:inertial_cylinder>
</link>

<link name="right_back_wheel">
    <visual>
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
            <geometry>
                <cylinder radius="${w_r}" length="${w_l}"/>
            </geometry>
        <material name="white" />
    </visual>
    <collision>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0.0 0.0"/>
        <geometry>
            <cylinder radius="${w_r}" length="${w_l}"/>
        </geometry>
    </collision>
    <xacro:inertial_cylinder mass="7" length="${w_l}" radius=
"${w_r}">
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
    </xacro:inertial_cylinder>
</link>

<link name="rear_drive_left">
    <visual>
        <origin xyz="-0 0 0" rpy="${pi/2} 0 0"/>
            <geometry>
                <cylinder radius="0.05" length="0.05"/>
            </geometry>
        <material name="white" />
    </visual>
    <xacro:inertial_cylinder mass="2" length="${w_l}" radius=
"${w_r}">
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </xacro:inertial_cylinder>
    <collision>
        <origin xyz="-0 0 0" rpy="${pi/2} 0 0"/>
            <geometry>
                <cylinder radius="0.05" length="0.05"/>
            </geometry>
    </collision>
</link>
```



```
<link name="rear_drive_right">
    <visual>
        <origin xyz="-0 0 0" rpy="-${pi/2} 0 0"/>
            <geometry>
                <cylinder radius="0.05" length="0.05"/>
            </geometry>
        <material name="white" />
    </visual>
    <xacro:inertial_cylinder mass="2" length="${w_l}" radius=
"${w_r}">
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </xacro:inertial_cylinder>
    <collision>
        <origin xyz="-0 0 0" rpy="-${pi/2} 0 0"/>
            <geometry>
                <cylinder radius="0.05" length="0.05"/>
            </geometry>
    </collision>
</link>




 <!--JOINTS-->

<joint name="world_to_center" type="fixed">
    <parent link="base_link" />
    <child link="chassi" />
    <origin  xyz="0 0 0.18" />
</joint>
<joint name="rear_axis_left" type="fixed">
    <parent link="chassi"/>
    <child link="rear_drive_left"/>
    <origin xyz="-${axis_dist} ${axis_width} 0" rpy="0 0.0 0.0"/>

    <axis xyz="0.0 1.0 0.0"/>
</joint>

<joint name="back_left_spin" type="continuous">
    <parent link="rear_drive_left"/>
    <child link="left_back_wheel"/>
    <origin xyz="0 ${rotation_dist} 0" rpy="0 0.0 0.0"/>
    <axis xyz="0.0 1.0 0.0"/>
</joint>

<joint name="rear_axis_right" type="fixed">
    <parent link="chassi"/>
    <child link="rear_drive_right"/>
    <origin xyz="-${axis_dist} -${axis_width} 0" rpy="0 0.0 0.0"
/>
    <axis xyz="0.0 1.0 0.0"/>
</joint>
<joint name="back_right_spin" type="continuous">
    <parent link="rear_drive_right"/>
    <child link="right_back_wheel"/>
    <origin xyz="0 -${rotation_dist} 0" rpy="0 0.0 0.0"/>
    <axis xyz="0.0 1.0 0.0"/>
</joint>
```

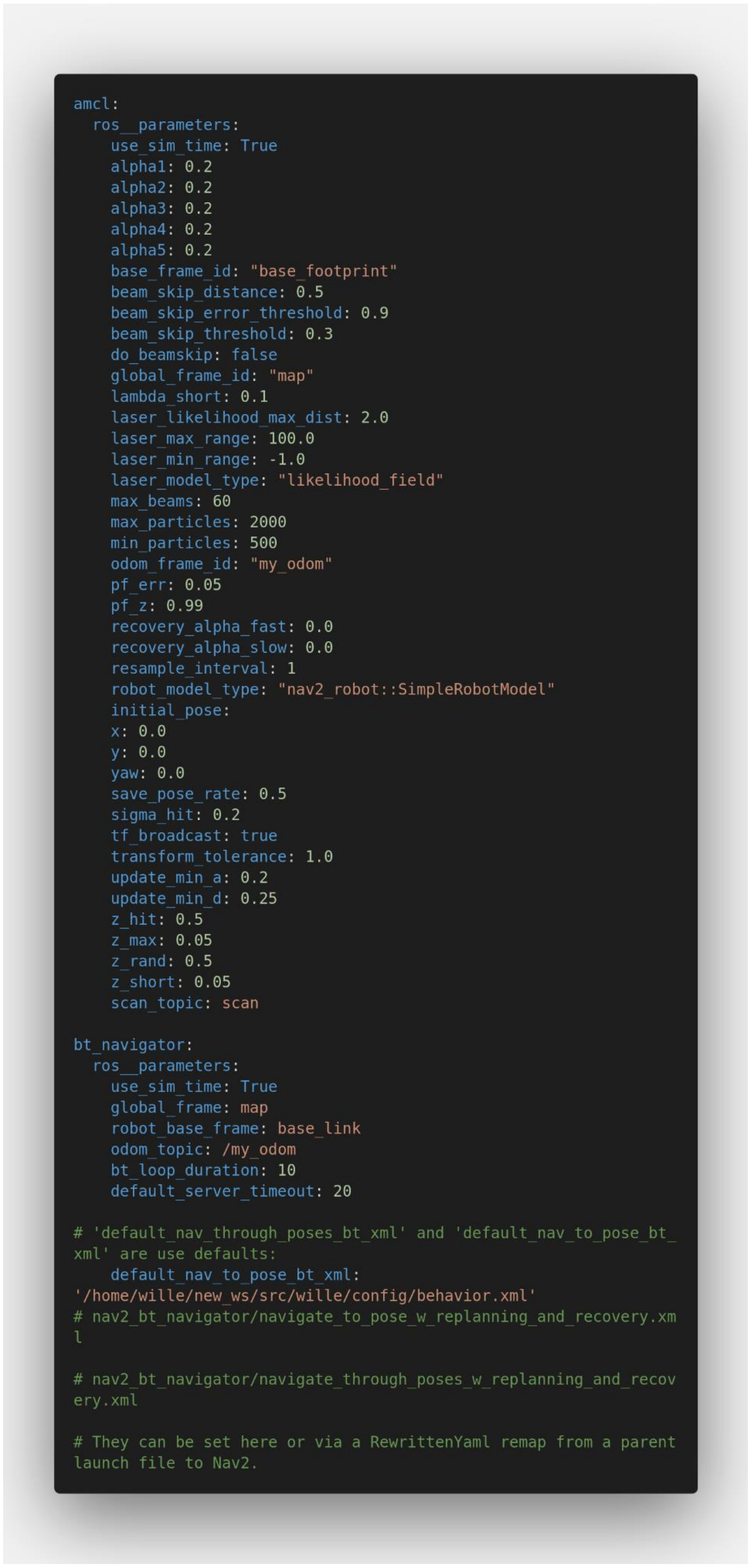


```
<joint name="left_front_joint_rot" type="revolute">
    <parent link="chassi" />
    <child link="front_left_rot" />
    <origin xyz="${axis_dist} ${axis_width} 0" rpy="0.0 0.0 0.0"
/>
    <axis xyz="0 0 1" />
    <limit lower="-${theta}" upper="${theta}" effort="1000.0"
velocity="5.0"/>
    <dynamics damping="0.01" friction="0.01" acceleration="10.0"
/>
</joint>

<joint name="front_left_spin" type="continuous">
    <parent link="front_left_rot"/>
    <child link="left_front_wheel"/>
    <origin xyz="0 ${rotation_dist} 0" rpy="0 0.0 0.0"/>
    <axis xyz="0.0 1.0 0.0"/>
</joint>

<joint name="right_front_joint_rot" type="revolute">
    <parent link="chassi" />
    <child link="front_right_rot" />
    <origin xyz="${axis_dist} -${axis_width} 0" rpy="0.0 0.0 0.0"
/>
    <axis xyz="0 0 1" />
    <limit lower="-${theta}" upper="${theta}" effort="1000.0"
velocity="5.0"/>
    <dynamics damping="0.01" friction="0.01" acceleration="10.0"
/>
</joint>
<joint name="front_right_spin" type="continuous">
    <parent link="front_right_rot"/>
    <child link="right_front_wheel"/>
    <origin xyz="0 -${rotation_dist} 0" rpy="0 0.0 0.0"/>
    <axis xyz="0.0 1.0 0.0"/>
</joint>
```

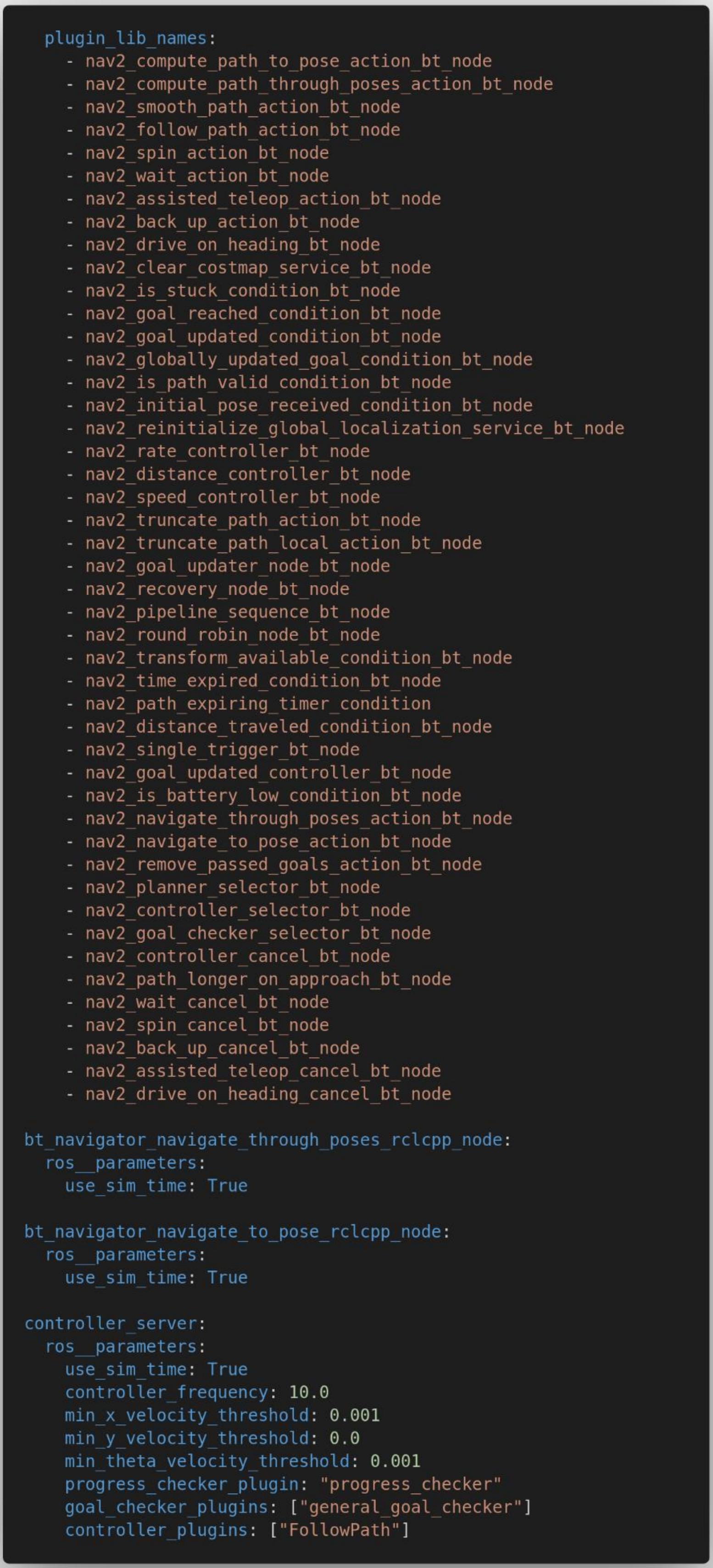


```
    <!--GAZEBO COLOURS-->
    <gazebo reference="right_back_wheel">
        <material>Gazebo/Black</material>
        <mu1>1.5</mu1>
        <mu2>1.5</mu2>
        <selfCollide>true</selfCollide>
    </gazebo>
    <gazebo reference="left_back_wheel">
        <material>Gazebo/Black</material>
        <mu1>1.5</mu1>
        <mu2>1.5</mu2>
        <selfCollide>true</selfCollide>
    </gazebo>
    <gazebo reference="left_front_wheel">
        <material>Gazebo/Black</material>
        <mu1>1.5</mu1>
        <mu2>1.5</mu2>
        <gravity>true</gravity>
    </gazebo>
    <gazebo reference="right_front_wheel">
        <material>Gazebo/Black</material>
        <mu1>1.5</mu1>
        <mu2>1.5</mu2>
        <gravity>true</gravity>
    </gazebo>
</robot>
```

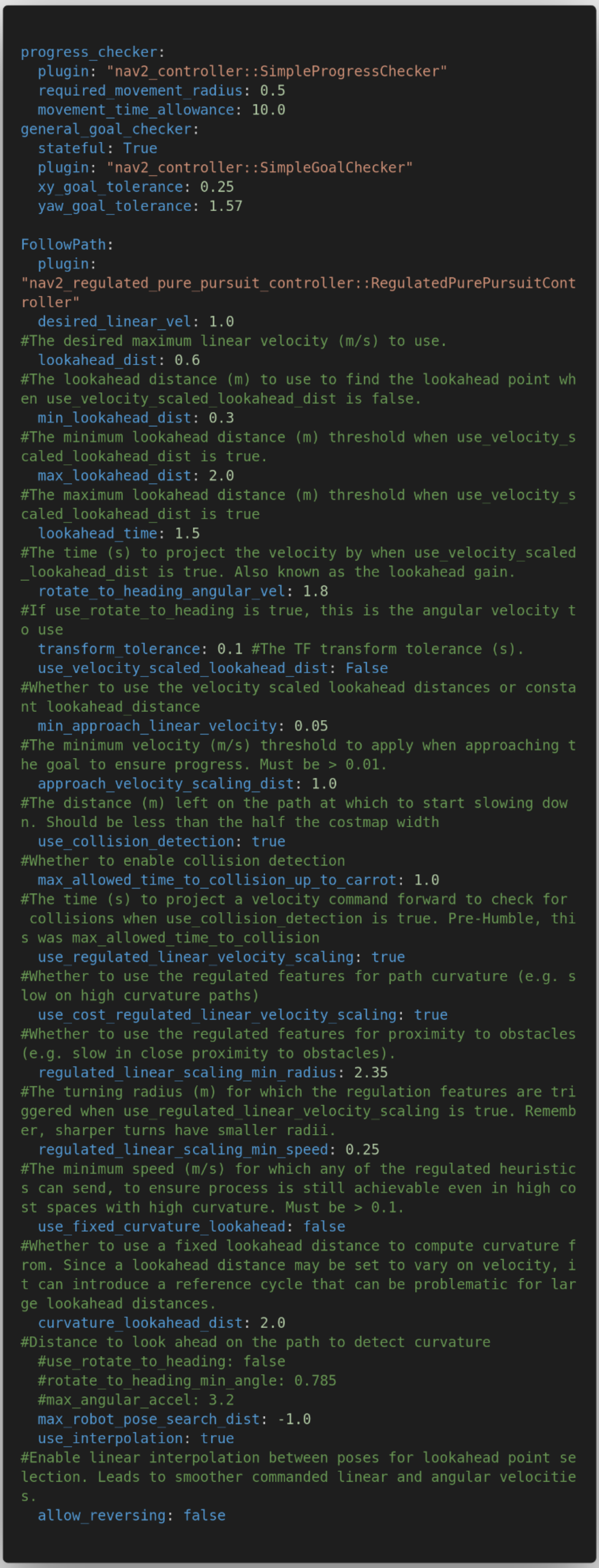
Appendix B – Nav2 Params



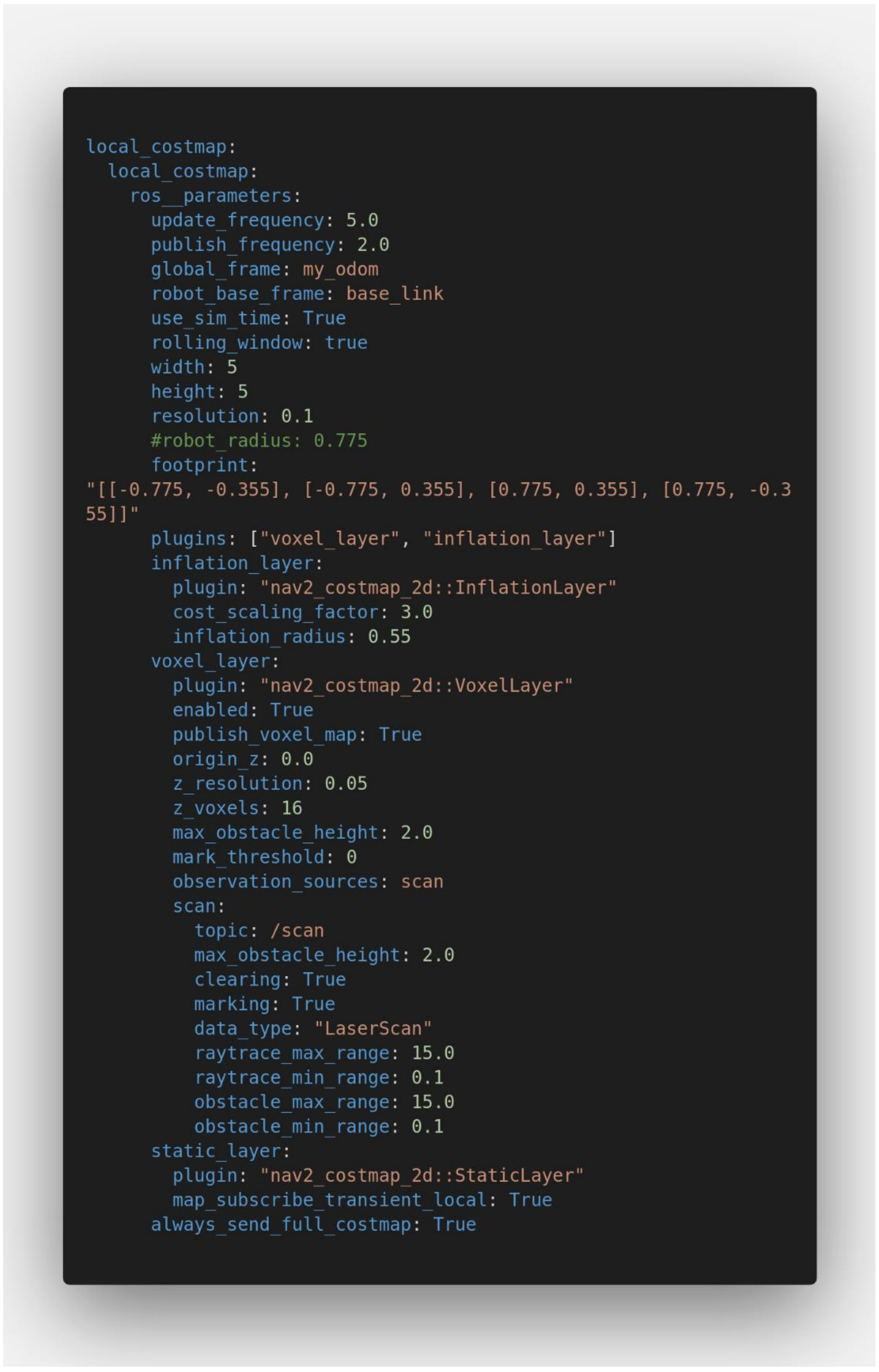
```yaml
amcl:
  ros__parameters:
    use_sim_time: True
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    alpha5: 0.2
    base_frame_id: "base_footprint"
    beam_skip_distance: 0.5
    beam_skip_error_threshold: 0.9
    beam_skip_threshold: 0.3
    do_beamskip: false
    global_frame_id: "map"
    lambda_short: 0.1
    laser_likelihood_max_dist: 2.0
    laser_max_range: 100.0
    laser_min_range: -1.0
    laser_model_type: "likelihood_field"
    max_beams: 60
    max_particles: 2000
    min_particles: 500
    odom_frame_id: "my_odom"
    pf_err: 0.05
    pf_z: 0.99
    recovery_alpha_fast: 0.0
    recovery_alpha_slow: 0.0
    resample_interval: 1
    robot_model_type: "nav2_robot::SimpleRobotModel"
    initial_pose:
    x: 0.0
    y: 0.0
    yaw: 0.0
    save_pose_rate: 0.5
    sigma_hit: 0.2
    tf_broadcast: true
    transform_tolerance: 1.0
    update_min_a: 0.2
    update_min_d: 0.25
    z_hit: 0.5
    z_max: 0.05
    z_rand: 0.5
    z_short: 0.05
    scan_topic: scan

bt_navigator:
  ros__parameters:
    use_sim_time: True
    global_frame: map
    robot_base_frame: base_link
    odom_topic: /my_odom
    bt_loop_duration: 10
    default_server_timeout: 20

# 'default_nav_through_poses_bt_xml' and 'default_nav_to_pose_bt_
xml' are use defaults:
    default_nav_to_pose_bt_xml:
'/home/wille/new_ws/src/wille/config/behavior.xml'
# nav2_bt_navigator/navigate_to_pose_w_replanning_and_recovery.xm
l

# nav2_bt_navigator/navigate_through_poses_w_replanning_and_recov
ery.xml

# They can be set here or via a RewrittenYaml remap from a parent
launch file to Nav2.
```



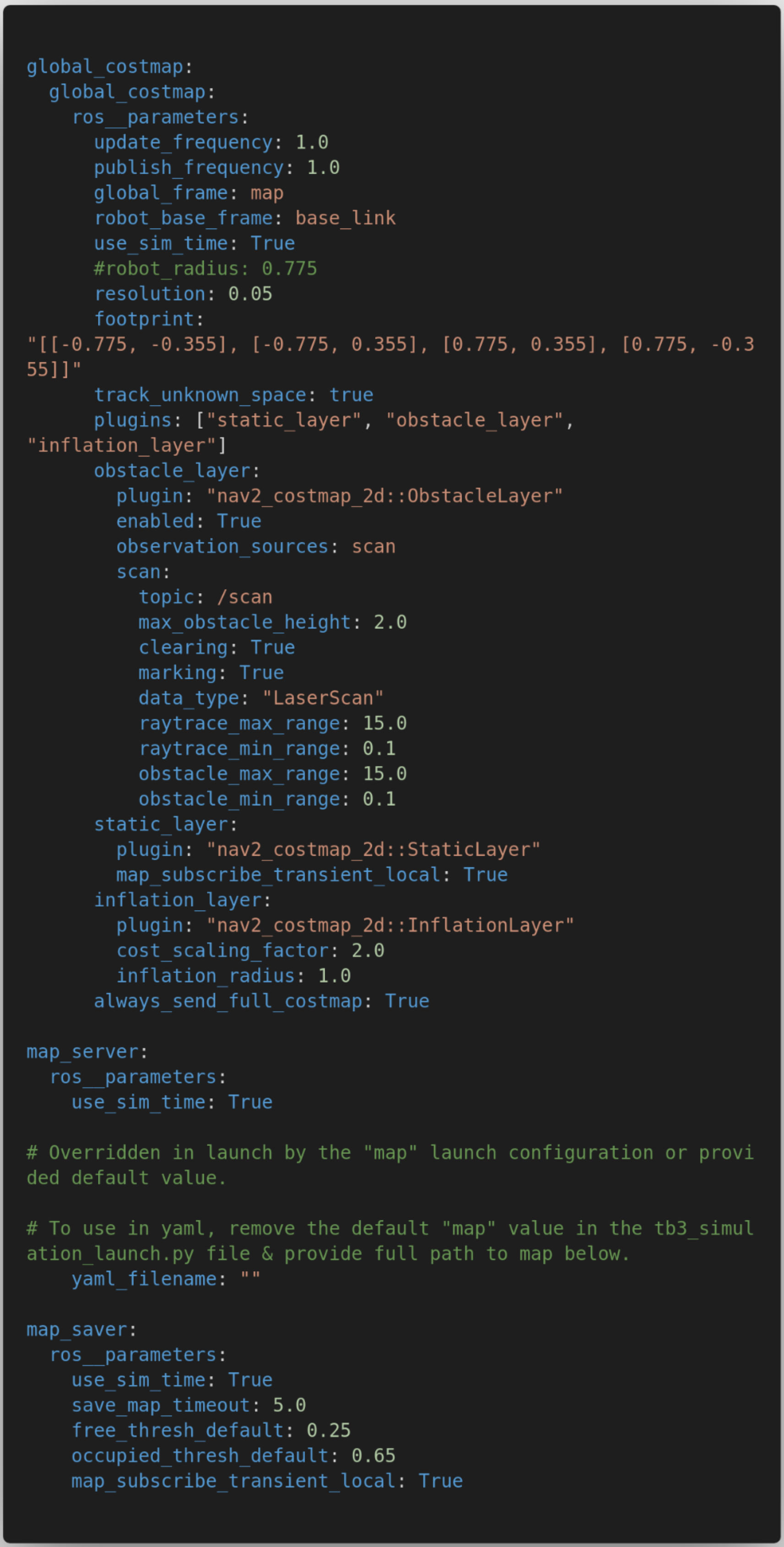
```yaml
  plugin_lib_names:
    - nav2_compute_path_to_pose_action_bt_node
    - nav2_compute_path_through_poses_action_bt_node
    - nav2_smooth_path_action_bt_node
    - nav2_follow_path_action_bt_node
    - nav2_spin_action_bt_node
    - nav2_wait_action_bt_node
    - nav2_assisted_teleop_action_bt_node
    - nav2_back_up_action_bt_node
    - nav2_drive_on_heading_bt_node
    - nav2_clear_costmap_service_bt_node
    - nav2_is_stuck_condition_bt_node
    - nav2_goal_reached_condition_bt_node
    - nav2_goal_updated_condition_bt_node
    - nav2_globally_updated_goal_condition_bt_node
    - nav2_is_path_valid_condition_bt_node
    - nav2_initial_pose_received_condition_bt_node
    - nav2_reinitialize_global_localization_service_bt_node
    - nav2_rate_controller_bt_node
    - nav2_distance_controller_bt_node
    - nav2_speed_controller_bt_node
    - nav2_truncate_path_action_bt_node
    - nav2_truncate_path_local_action_bt_node
    - nav2_goal_updater_node_bt_node
    - nav2_recovery_node_bt_node
    - nav2_pipeline_sequence_bt_node
    - nav2_round_robin_node_bt_node
    - nav2_transform_available_condition_bt_node
    - nav2_time_expired_condition_bt_node
    - nav2_path_expiring_timer_condition
    - nav2_distance_traveled_condition_bt_node
    - nav2_single_trigger_bt_node
    - nav2_goal_updated_controller_bt_node
    - nav2_is_battery_low_condition_bt_node
    - nav2_navigate_through_poses_action_bt_node
    - nav2_navigate_to_pose_action_bt_node
    - nav2_remove_passed_goals_action_bt_node
    - nav2_planner_selector_bt_node
    - nav2_controller_selector_bt_node
    - nav2_goal_checker_selector_bt_node
    - nav2_controller_cancel_bt_node
    - nav2_path_longer_on_approach_bt_node
    - nav2_wait_cancel_bt_node
    - nav2_spin_cancel_bt_node
    - nav2_back_up_cancel_bt_node
    - nav2_assisted_teleop_cancel_bt_node
    - nav2_drive_on_heading_cancel_bt_node

bt_navigator_navigate_through_poses_rclcpp_node:
  ros__parameters:
    use_sim_time: True

bt_navigator_navigate_to_pose_rclcpp_node:
  ros__parameters:
    use_sim_time: True

controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 10.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.0
    min_theta_velocity_threshold: 0.001
    progress_checker_plugin: "progress_checker"
    goal_checker_plugins: ["general_goal_checker"]
    controller_plugins: ["FollowPath"]
```



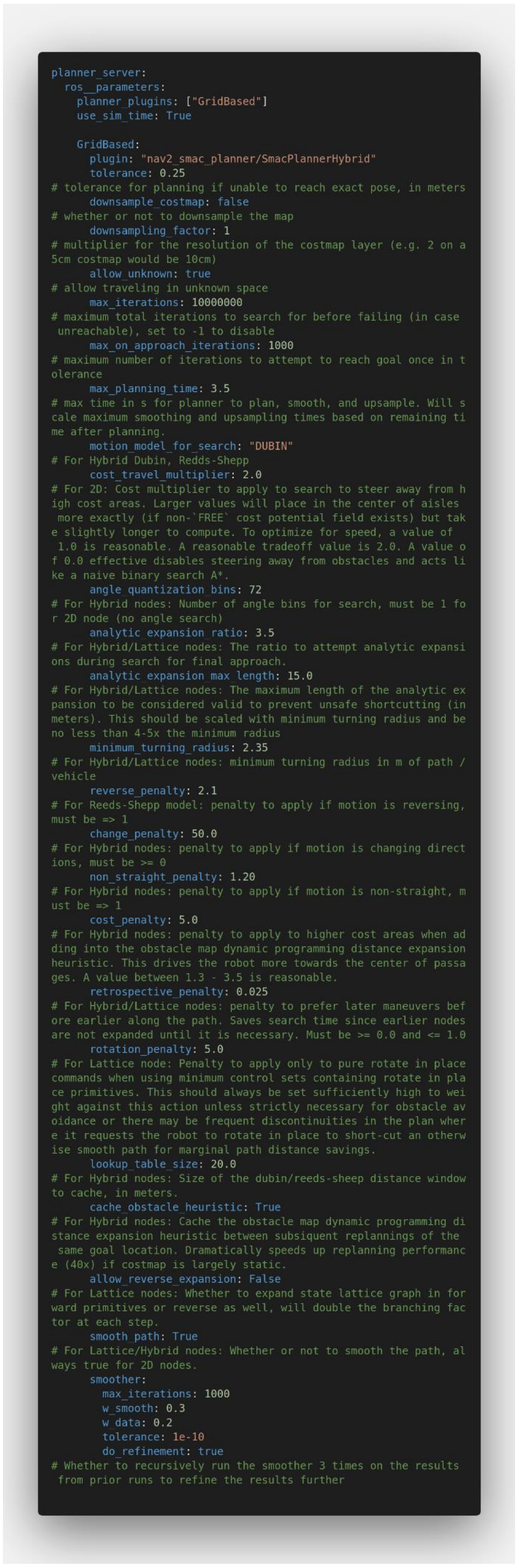
```
progress_checker:
  plugin: "nav2_controller::SimpleProgressChecker"
  required_movement_radius: 0.5
  movement_time_allowance: 10.0
general_goal_checker:
  stateful: True
  plugin: "nav2_controller::SimpleGoalChecker"
  xy_goal_tolerance: 0.25
  yaw_goal_tolerance: 1.57

FollowPath:
  plugin:
"nav2_regulated_pure_pursuit_controller::RegulatedPurePursuitCont
roller"
  desired_linear_vel: 1.0
#The desired maximum linear velocity (m/s) to use.
  lookahead_dist: 0.6
#The lookahead distance (m) to use to find the lookahead point wh
en use_velocity_scaled_lookahead_dist is false.
  min_lookahead_dist: 0.3
#The minimum lookahead distance (m) threshold when use_velocity_s
caled_lookahead_dist is true.
  max_lookahead_dist: 2.0
#The maximum lookahead distance (m) threshold when use_velocity_s
caled_lookahead_dist is true
  lookahead_time: 1.5
#The time (s) to project the velocity by when use_velocity_scaled
_lookahead_dist is true. Also known as the lookahead gain.
  rotate_to_heading_angular_vel: 1.8
#If use_rotate_to_heading is true, this is the angular velocity t
o use
  transform_tolerance: 0.1 #The TF transform tolerance (s).
  use_velocity_scaled_lookahead_dist: False
#Whether to use the velocity scaled lookahead distances or consta
nt lookahead_distance
  min_approach_linear_velocity: 0.05
#The minimum velocity (m/s) threshold to apply when approaching t
he goal to ensure progress. Must be > 0.01.
  approach_velocity_scaling_dist: 1.0
#The distance (m) left on the path at which to start slowing dow
n. Should be less than the half the costmap width
  use_collision_detection: true
#Whether to enable collision detection
  max_allowed_time_to_collision_up_to_carrot: 1.0
#The time (s) to project a velocity command forward to check for
 collisions when use_collision_detection is true. Pre-Humble, thi
s was max_allowed_time_to_collision
  use_regulated_linear_velocity_scaling: true
#Whether to use the regulated features for path curvature (e.g. s
low on high curvature paths)
  use_cost_regulated_linear_velocity_scaling: true
#Whether to use the regulated features for proximity to obstacles
(e.g. slow in close proximity to obstacles).
  regulated_linear_scaling_min_radius: 2.35
#The turning radius (m) for which the regulation features are tri
ggered when use_regulated_linear_velocity_scaling is true. Rememb
er, sharper turns have smaller radii.
  regulated_linear_scaling_min_speed: 0.25
#The minimum speed (m/s) for which any of the regulated heuristic
s can send, to ensure process is still achievable even in high co
st spaces with high curvature. Must be > 0.1.
  use_fixed_curvature_lookahead: false
#Whether to use a fixed lookahead distance to compute curvature f
rom. Since a lookahead distance may be set to vary on velocity, i
t can introduce a reference cycle that can be problematic for lar
ge lookahead distances.
  curvature_lookahead_dist: 2.0
#Distance to look ahead on the path to detect curvature
  #use_rotate_to_heading: false
  #rotate_to_heading_min_angle: 0.785
  #max_angular_accel: 3.2
  max_robot_pose_search_dist: -1.0
  use_interpolation: true
#Enable linear interpolation between poses for lookahead point se
lection. Leads to smoother commanded linear and angular velocitie
s.
  allow_reversing: false
```



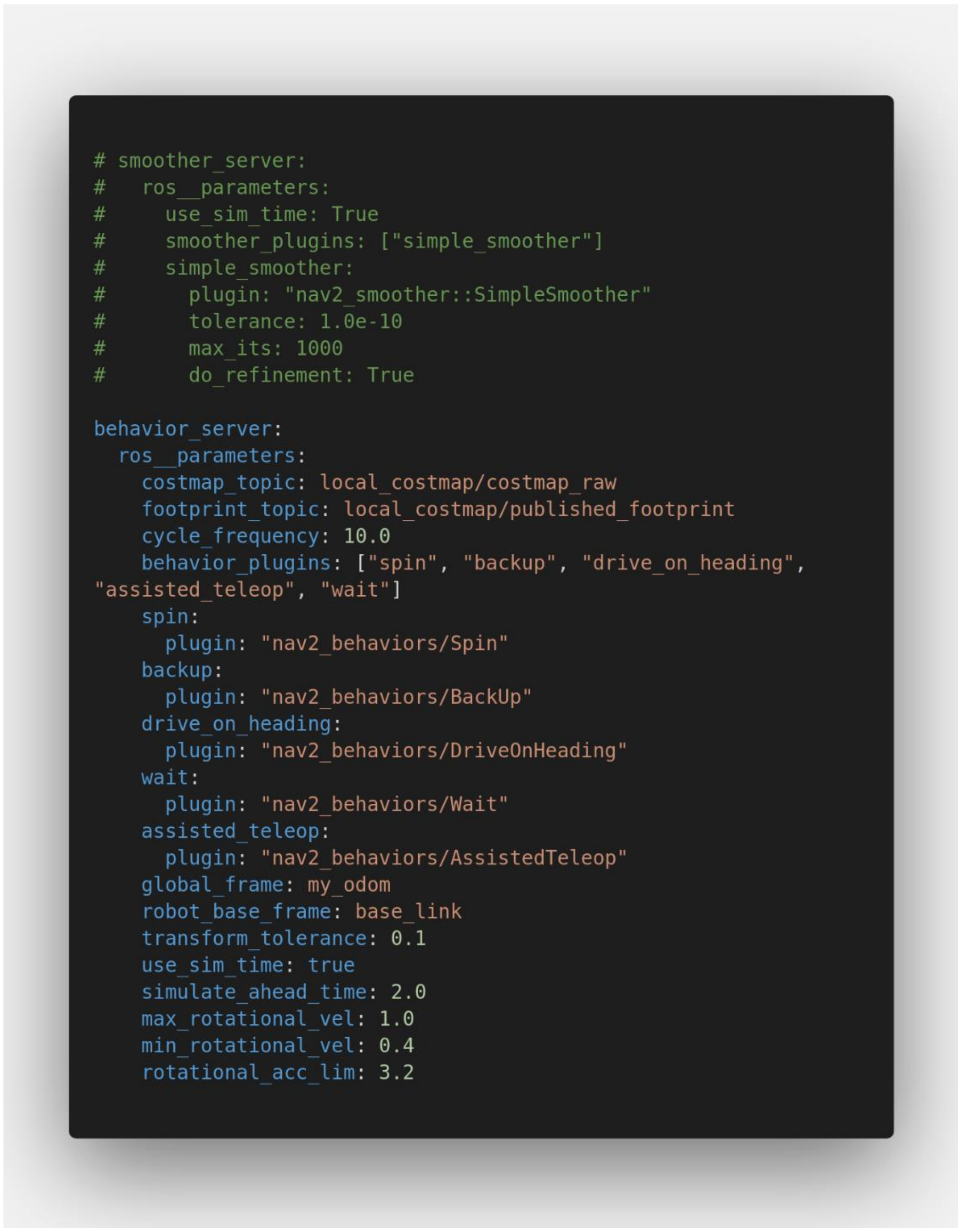
```
local_costmap:
  local_costmap:
    ros__parameters:
      update_frequency: 5.0
      publish_frequency: 2.0
      global_frame: my_odom
      robot_base_frame: base_link
      use_sim_time: True
      rolling_window: true
      width: 5
      height: 5
      resolution: 0.1
      #robot_radius: 0.775
      footprint:
"[[-0.775, -0.355], [-0.775, 0.355], [0.775, 0.355], [0.775, -0.3
55]]"
      plugins: ["voxel_layer", "inflation_layer"]
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      voxel_layer:
        plugin: "nav2_costmap_2d::VoxelLayer"
        enabled: True
        publish_voxel_map: True
        origin_z: 0.0
        z_resolution: 0.05
        z_voxels: 16
        max_obstacle_height: 2.0
        mark_threshold: 0
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"
          raytrace_max_range: 15.0
          raytrace_min_range: 0.1
          obstacle_max_range: 15.0
          obstacle_min_range: 0.1
      static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
        map_subscribe_transient_local: True
      always_send_full_costmap: True
```



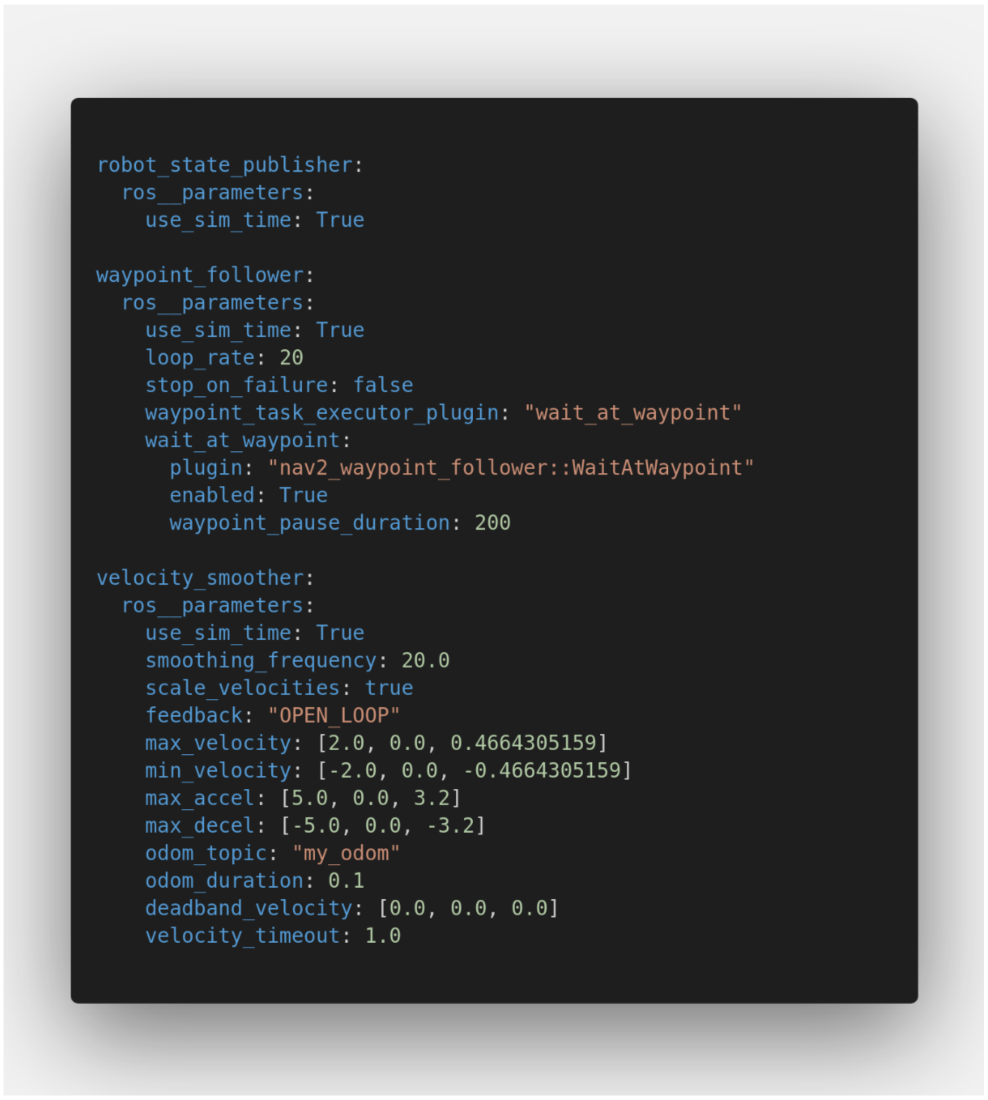
```
global_costmap:
  global_costmap:
    ros__parameters:
      update_frequency: 1.0
      publish_frequency: 1.0
      global_frame: map
      robot_base_frame: base_link
      use_sim_time: True
      #robot_radius: 0.775
      resolution: 0.05
      footprint:
"[[-0.775, -0.355], [-0.775, 0.355], [0.775, 0.355], [0.775, -0.3
55]]"
      track_unknown_space: true
      plugins: ["static_layer", "obstacle_layer",
"inflation_layer"]
      obstacle_layer:
        plugin: "nav2_costmap_2d::ObstacleLayer"
        enabled: True
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"
          raytrace_max_range: 15.0
          raytrace_min_range: 0.1
          obstacle_max_range: 15.0
          obstacle_min_range: 0.1
      static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
        map_subscribe_transient_local: True
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 2.0
        inflation_radius: 1.0
      always_send_full_costmap: True

map_server:
  ros__parameters:
    use_sim_time: True

# Overridden in launch by the "map" launch configuration or provi
ded default value.

# To use in yaml, remove the default "map" value in the tb3_simul
ation_launch.py file & provide full path to map below.
    yaml_filename: ""

map_saver:
  ros__parameters:
    use_sim_time: True
    save_map_timeout: 5.0
    free_thresh_default: 0.25
    occupied_thresh_default: 0.65
    map_subscribe_transient_local: True
```



```yaml
planner_server:
  ros__parameters:
    planner_plugins: ["GridBased"]
    use_sim_time: True

    GridBased:
      plugin: "nav2_smac_planner/SmacPlannerHybrid"
      tolerance: 0.25
# tolerance for planning if unable to reach exact pose, in meters
      downsample_costmap: false
# whether or not to downsample the map
      downsampling_factor: 1
# multiplier for the resolution of the costmap layer (e.g. 2 on a
5cm costmap would be 10cm)
      allow_unknown: true
# allow traveling in unknown space
      max_iterations: 10000000
# maximum total iterations to search for before failing (in case
 unreachable), set to -1 to disable
      max_on_approach_iterations: 1000
# maximum number of iterations to attempt to reach goal once in t
olerance
      max_planning_time: 3.5
# max time in s for planner to plan, smooth, and upsample. Will s
cale maximum smoothing and upsampling times based on remaining ti
me after planning.
      motion_model_for_search: "DUBIN"
# For Hybrid Dubin, Redds-Shepp
      cost_travel_multiplier: 2.0
# For 2D: Cost multiplier to apply to search to steer away from h
igh cost areas. Larger values will place in the center of aisles
 more exactly (if non-`FREE` cost potential field exists) but tak
e slightly longer to compute. To optimize for speed, a value of
 1.0 is reasonable. A reasonable tradeoff value is 2.0. A value o
f 0.0 effective disables steering away from obstacles and acts li
ke a naive binary search A*.
      angle_quantization_bins: 72
# For Hybrid nodes: Number of angle bins for search, must be 1 fo
r 2D node (no angle search)
      analytic_expansion_ratio: 3.5
# For Hybrid/Lattice nodes: The ratio to attempt analytic expansi
ons during search for final approach.
      analytic_expansion_max_length: 15.0
# For Hybrid/Lattice nodes: The maximum length of the analytic ex
pansion to be considered valid to prevent unsafe shortcutting (in
meters). This should be scaled with minimum turning radius and be
no less than 4-5x the minimum radius
      minimum_turning_radius: 2.35
# For Hybrid/Lattice nodes: minimum turning radius in m of path /
vehicle
      reverse_penalty: 2.1
# For Reeds-Shepp model: penalty to apply if motion is reversing,
must be => 1
      change_penalty: 50.0
# For Hybrid nodes: penalty to apply if motion is changing direct
ions, must be >= 0
      non_straight_penalty: 1.20
# For Hybrid nodes: penalty to apply if motion is non-straight, m
ust be => 1
      cost_penalty: 5.0
# For Hybrid nodes: penalty to apply to higher cost areas when ad
ding into the obstacle map dynamic programming distance expansion
heuristic. This drives the robot more towards the center of passa
ges. A value between 1.3 - 3.5 is reasonable.
      retrospective_penalty: 0.025
# For Hybrid/Lattice nodes: penalty to prefer later maneuvers bef
ore earlier along the path. Saves search time since earlier nodes
are not expanded until it is necessary. Must be >= 0.0 and <= 1.0
      rotation_penalty: 5.0
# For Lattice node: Penalty to apply only to pure rotate in place
commands when using minimum control sets containing rotate in pla
ce primitives. This should always be set sufficiently high to wei
ght against this action unless strictly necessary for obstacle av
oidance or there may be frequent discontinuities in the plan wher
e it requests the robot to rotate in place to short-cut an otherw
ise smooth path for marginal path distance savings.
      lookup_table_size: 20.0
# For Hybrid nodes: Size of the dubin/reeds-sheep distance window
to cache, in meters.
      cache_obstacle_heuristic: True
# For Hybrid nodes: Cache the obstacle map dynamic programming di
stance expansion heuristic between subsiquent replannings of the
 same goal location. Dramatically speeds up replanning performanc
e (40x) if costmap is largely static.
      allow_reverse_expansion: False
# For Lattice nodes: Whether to expand state lattice graph in for
ward primitives or reverse as well, will double the branching fac
tor at each step.
      smooth_path: True
# For Lattice/Hybrid nodes: Whether or not to smooth the path, al
ways true for 2D nodes.
      smoother:
        max_iterations: 1000
        w_smooth: 0.3
        w_data: 0.2
        tolerance: 1e-10
        do_refinement: true
# Whether to recursively run the smoother 3 times on the results
 from prior runs to refine the results further
```

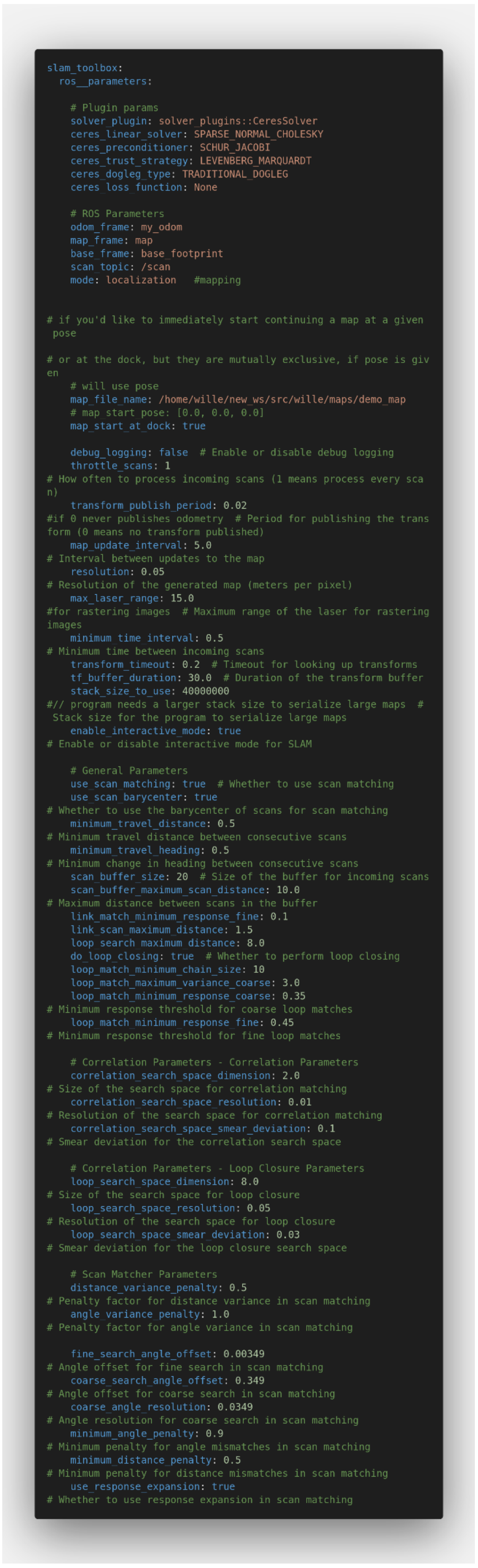


```yaml
# smoother_server:
#   ros__parameters:
#     use_sim_time: True
#     smoother_plugins: ["simple_smoother"]
#     simple_smoother:
#       plugin: "nav2_smoother::SimpleSmoother"
#       tolerance: 1.0e-10
#       max_its: 1000
#       do_refinement: True

behavior_server:
  ros__parameters:
    costmap_topic: local_costmap/costmap_raw
    footprint_topic: local_costmap/published_footprint
    cycle_frequency: 10.0
    behavior_plugins: ["spin", "backup", "drive_on_heading",
"assisted_teleop", "wait"]
    spin:
      plugin: "nav2_behaviors/Spin"
    backup:
      plugin: "nav2_behaviors/BackUp"
    drive_on_heading:
      plugin: "nav2_behaviors/DriveOnHeading"
    wait:
      plugin: "nav2_behaviors/Wait"
    assisted_teleop:
      plugin: "nav2_behaviors/AssistedTeleop"
    global_frame: my_odom
    robot_base_frame: base_link
    transform_tolerance: 0.1
    use_sim_time: true
    simulate_ahead_time: 2.0
    max_rotational_vel: 1.0
    min_rotational_vel: 0.4
    rotational_acc_lim: 3.2
```



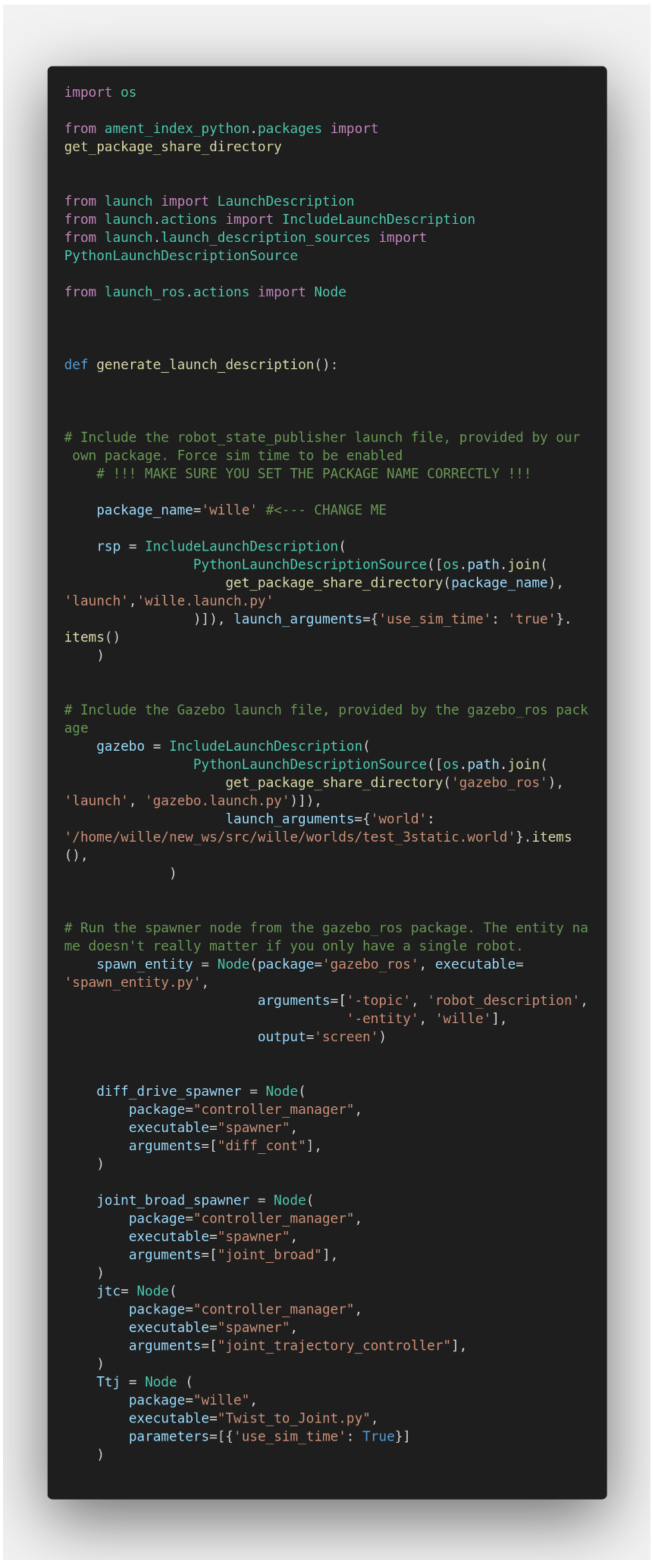
```yaml
robot_state_publisher:
  ros__parameters:
    use_sim_time: True

waypoint_follower:
  ros__parameters:
    use_sim_time: True
    loop_rate: 20
    stop_on_failure: false
    waypoint_task_executor_plugin: "wait_at_waypoint"
    wait_at_waypoint:
      plugin: "nav2_waypoint_follower::WaitAtWaypoint"
      enabled: True
      waypoint_pause_duration: 200

velocity_smoother:
  ros__parameters:
    use_sim_time: True
    smoothing_frequency: 20.0
    scale_velocities: true
    feedback: "OPEN_LOOP"
    max_velocity: [2.0, 0.0, 0.4664305159]
    min_velocity: [-2.0, 0.0, -0.4664305159]
    max_accel: [5.0, 0.0, 3.2]
    max_decel: [-5.0, 0.0, -3.2]
    odom_topic: "my_odom"
    odom_duration: 0.1
    deadband_velocity: [0.0, 0.0, 0.0]
    velocity_timeout: 1.0
```

Appendix C – SLAM Parameters

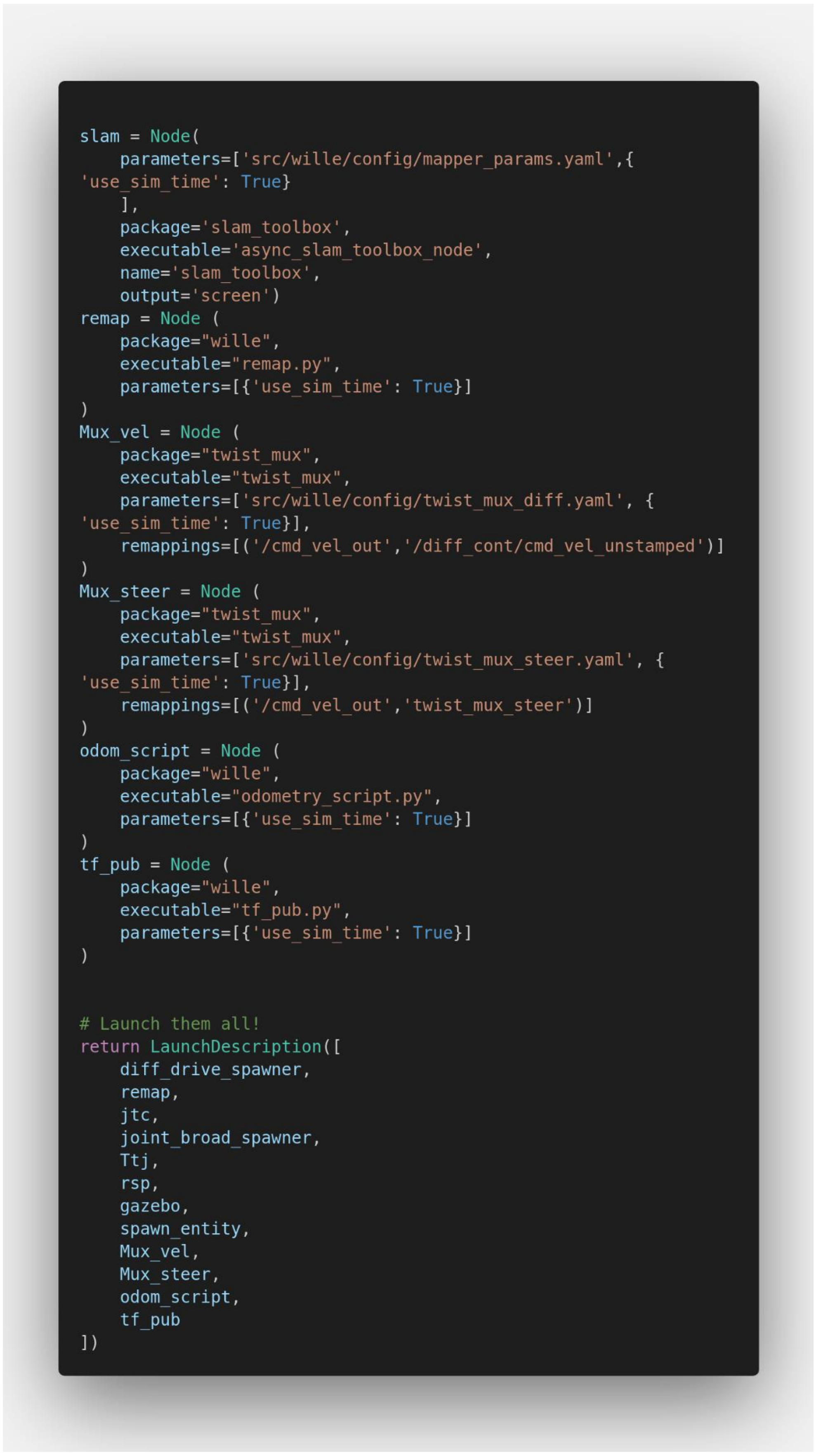


```yaml
slam_toolbox:
  ros__parameters:

    # Plugin params
    solver_plugin: solver_plugins::CeresSolver
    ceres_linear_solver: SPARSE_NORMAL_CHOLESKY
    ceres_preconditioner: SCHUR_JACOBI
    ceres_trust_strategy: LEVENBERG_MARQUARDT
    ceres_dogleg_type: TRADITIONAL_DOGLEG
    ceres_loss_function: None

    # ROS Parameters
    odom_frame: my_odom
    map_frame: map
    base_frame: base_footprint
    scan_topic: /scan
    mode: localization   #mapping


# if you'd like to immediately start continuing a map at a given
 pose

# or at the dock, but they are mutually exclusive, if pose is giv
en
    # will use pose
    map_file_name: /home/wille/new_ws/src/wille/maps/demo_map
    # map_start_pose: [0.0, 0.0, 0.0]
    map_start_at_dock: true

    debug_logging: false  # Enable or disable debug logging
    throttle_scans: 1
# How often to process incoming scans (1 means process every sca
n)
    transform_publish_period: 0.02
#if 0 never publishes odometry  # Period for publishing the trans
form (0 means no transform published)
    map_update_interval: 5.0
# Interval between updates to the map
    resolution: 0.05
# Resolution of the generated map (meters per pixel)
    max_laser_range: 15.0
#for rastering images  # Maximum range of the laser for rastering
images
    minimum_time_interval: 0.5
# Minimum time between incoming scans
    transform_timeout: 0.2  # Timeout for looking up transforms
    tf_buffer_duration: 30.0  # Duration of the transform buffer
    stack_size_to_use: 40000000
#// program needs a larger stack size to serialize large maps  #
 Stack size for the program to serialize large maps
    enable_interactive_mode: true
# Enable or disable interactive mode for SLAM

    # General Parameters
    use_scan_matching: true  # Whether to use scan matching
    use_scan_barycenter: true
# Whether to use the barycenter of scans for scan matching
    minimum_travel_distance: 0.5
# Minimum travel distance between consecutive scans
    minimum_travel_heading: 0.5
# Minimum change in heading between consecutive scans
    scan_buffer_size: 20  # Size of the buffer for incoming scans
    scan_buffer_maximum_scan_distance: 10.0
# Maximum distance between scans in the buffer
    link_match_minimum_response_fine: 0.1
    link_scan_maximum_distance: 1.5
    loop_search_maximum_distance: 8.0
    do_loop_closing: true  # Whether to perform loop closing
    loop_match_minimum_chain_size: 10
    loop_match_maximum_variance_coarse: 3.0
    loop_match_minimum_response_coarse: 0.35
# Minimum response threshold for coarse loop matches
    loop_match_minimum_response_fine: 0.45
# Minimum response threshold for fine loop matches

    # Correlation Parameters - Correlation Parameters
    correlation_search_space_dimension: 2.0
# Size of the search space for correlation matching
    correlation_search_space_resolution: 0.01
# Resolution of the search space for correlation matching
    correlation_search_space_smear_deviation: 0.1
# Smear deviation for the correlation search space

    # Correlation Parameters - Loop Closure Parameters
    loop_search_space_dimension: 8.0
# Size of the search space for loop closure
    loop_search_space_resolution: 0.05
# Resolution of the search space for loop closure
    loop_search_space_smear_deviation: 0.03
# Smear deviation for the loop closure search space

    # Scan Matcher Parameters
    distance_variance_penalty: 0.5
# Penalty factor for distance variance in scan matching
    angle_variance_penalty: 1.0
# Penalty factor for angle variance in scan matching

    fine_search_angle_offset: 0.00349
# Angle offset for fine search in scan matching
    coarse_search_angle_offset: 0.349
# Angle offset for coarse search in scan matching
    coarse_angle_resolution: 0.0349
# Angle resolution for coarse search in scan matching
    minimum_angle_penalty: 0.9
# Minimum penalty for angle mismatches in scan matching
    minimum_distance_penalty: 0.5
# Minimum penalty for distance mismatches in scan matching
    use_response_expansion: true
# Whether to use response expansion in scan matching
```

Appendix D – Launch file which launches all nodes except Nav2, SLAM and steering script



```
import os

from ament_index_python.packages import
get_package_share_directory


from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import
PythonLaunchDescriptionSource

from launch_ros.actions import Node



def generate_launch_description():



# Include the robot_state_publisher launch file, provided by our
 own package. Force sim time to be enabled
    # !!! MAKE SURE YOU SET THE PACKAGE NAME CORRECTLY !!!

    package_name='wille' #<--- CHANGE ME

    rsp = IncludeLaunchDescription(
                PythonLaunchDescriptionSource([os.path.join(
                    get_package_share_directory(package_name),
'launch','wille.launch.py'
                )]), launch_arguments={'use_sim_time': 'true'}.
items()
    )


# Include the Gazebo launch file, provided by the gazebo_ros pack
age
    gazebo = IncludeLaunchDescription(
                PythonLaunchDescriptionSource([os.path.join(
                    get_package_share_directory('gazebo_ros'),
'launch', 'gazebo.launch.py')]),
                    launch_arguments={'world':
'/home/wille/new_ws/src/wille/worlds/test_3static.world'}.items
(),
             )


# Run the spawner node from the gazebo_ros package. The entity na
me doesn't really matter if you only have a single robot.
    spawn_entity = Node(package='gazebo_ros', executable=
'spawn_entity.py',
                        arguments=['-topic', 'robot_description',
                                   '-entity', 'wille'],
                        output='screen')


    diff_drive_spawner = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["diff_cont"],
    )

    joint_broad_spawner = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["joint_broad"],
    )
    jtc= Node(
        package="controller_manager",
        executable="spawner",
        arguments=["joint_trajectory_controller"],
    )
    Ttj = Node (
        package="wille",
        executable="Twist_to_Joint.py",
        parameters=[{'use_sim_time': True}]
    )
```



```
slam = Node(
    parameters=['src/wille/config/mapper_params.yaml',{
'use_sim_time': True}
    ],
    package='slam_toolbox',
    executable='async_slam_toolbox_node',
    name='slam_toolbox',
    output='screen')
remap = Node (
    package="wille",
    executable="remap.py",
    parameters=[{'use_sim_time': True}]
)
Mux_vel = Node (
    package="twist_mux",
    executable="twist_mux",
    parameters=['src/wille/config/twist_mux_diff.yaml', {
'use_sim_time': True}],
    remappings=[('/cmd_vel_out','/diff_cont/cmd_vel_unstamped')]
)
Mux_steer = Node (
    package="twist_mux",
    executable="twist_mux",
    parameters=['src/wille/config/twist_mux_steer.yaml', {
'use_sim_time': True}],
    remappings=[('/cmd_vel_out','twist_mux_steer')]
)
odom_script = Node (
    package="wille",
    executable="odometry_script.py",
    parameters=[{'use_sim_time': True}]
)
tf_pub = Node (
    package="wille",
    executable="tf_pub.py",
    parameters=[{'use_sim_time': True}]
)


# Launch them all!
return LaunchDescription([
    diff_drive_spawner,
    remap,
    jtc,
    joint_broad_spawner,
    Ttj,
    rsp,
    gazebo,
    spawn_entity,
    Mux_vel,
    Mux_steer,
    odom_script,
    tf_pub
])
```

Appendix E – Measurements from trajectory matching test

	Vehicle position turn (m)	Vehicle position obstacle (m)	Vehicle position destination (m)	Time (s)
X1	16,05	15,7	17,2	31
Y1	1,81	8,19	12,86	
Distance to obstacle/goal	1,26	0,7	0,11	

	Vehicle position turn (m)	Vehicle position obstacle (m)	Vehicle position destination (m)	Time (s)
X2	16,15	15,54	17,21	30
Y2	1,75	8,26	11,27	
Distance to obstacle/goal	1,37	0,54	0,27	

	Vehicle position turn (m)	Vehicle position obstacle (m)	Vehicle position destination (m)	Time (s)
X3	15,77	15,66	17,19	31
Y3	1,52	8,48	13,45	
Distance to obstacle/goal	1,25	0,66	0,31	

	Vehicle position turn (m)	Vehicle position obstacle (m)	Vehicle position destination (m)	Time (s)
X4	15,5	15,59	17,35	34
Y4	1,78	8,28	13,74	
Distance to obstacle/goal	0,88	0,59	0,34	

	Vehicle position turn (m)	Vehicle position obstacle (m)	Vehicle position destination (m)	Time (s)
X5	15,47	15,68	17,14	34
Y5	1,93	8,05	13,9	
Distance to obstacle/goal	0,74	0,68	0,38	