GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

SIMULACIÓN DE LAS DINÁMICAS DE UN COCHE EN UN ENTORNO MONTAÑOSO USANDO ROS Y GAZEBO

Autor
Jaime Jarauta Gastelu

Director
Dr. Richard B. Sowers

Madrid
Mayo 2024

BACHELOR'S DEGREE IN INDUSTRIAL ENGINEERING

BACHELOR'S THESIS

SIMULATION OF THE DYNAMICS OF A MOUNTAIN CAR IN A ROS/GAZEBO ENVIRONMENT

Author
Jaime Jarauta Gastelu

Director
Dr. Richard B. Sowers

Madrid
May 2024

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Simulation of the dynamics of a Mountain Car in a ROS/Gazebo Environment

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2023/24 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Jaime Jarauta Gastelu Fecha: 01/05/2024

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Richard B. Sowers Fecha: 01/05/2024

Acknowledgements y Agradecimientos

Ante todo, quiero expresar mi más profundo agradecimiento a mis padres, Rocío y Javier, por su amor incondicional y apoyo constante. A mi hermano y compañero de viajes, Javi, gran parte de lo que he logrado y de quien soy hoy, te lo debo a ti.

A mis amigos Galo, Jose y Víctor, gracias por todas las horas compartidas de trabajo y esfuerzo, que sin duda han valido la pena. Quién nos iba a decir hace años que llegaríamos hasta aquí. Espero que el futuro nos traiga muchos más años llenos de momentos juntos.

También debo un agradecimiento especial a mis amigos y compañeros de ICAI. En particular a personas como Ana, Isa y Sara entre otras, que han hecho que estos años sean inolvidables. Aún nos queda mucho por compartir.

To Professor Sowers and everyone at the University of Illinois, thank you for the unimaginable opportunities provided this year. This place has earned a special place in my heart. I am incredibly grateful to all the friends, teachers, and experiences that have shaped my journey there.

Finalmente, a la Universidad Pontificia Comillas, aunque nuestro camino ha tenido sus retos, reconozco el papel crucial que ha jugado en mi formación. Espero que las ecuaciones diferenciales no formen parte de mi futuro.

"We are not going in circles, we are going upwards.
The path is a spiral; we have already climbed many steps."

— Hermann Hesse, *Siddhartha*.

SIMULACIÓN DE LAS DINÁMICAS DE UN COCHE EN UN ENTORNO MONTAÑOSO USANDO ROS Y GAZEBO

Autor: Jarauta Gastelu, Jaime.

Supervisor: Sowers, Richard B., PhD.

Entidad Colaboradora: Universidad de Illinois en Urbana-Champaign

RESUMEN

Este Trabajo de Fin de Grado documenta el desarrollo de un simulador de las dinámicas de un coche en un entorno montañoso utilizando ROS (Robot Operating System) y Gazebo. El propósito es implementar y evaluar diversas estrategias de aprendizaje automático que controlan el movimiento del vehículo en función de su posición y velocidad instantánea, así como examinar cómo navega a través de un terreno con altitud variante.

La metodología incluye el uso de modelos simplificados de un coche y una montaña para simular las interacciones físicas como la gravedad, fricción y colisiones. El entorno de simulación fue creado en ROS y Gazebo por varias razones, incluyendo la capacidad de estos programas para desarrollar entornos basados completamente en código y la posibilidad de cambiar todos los parámetros de la simulación a través de la programación. Entre otros programas usados se incluyen Python o C++ para desarrollar las políticas de machine learning, así como XML para el desarrollo de los entornos de ROS y Autodesk y Fusion 360 para el diseño del entorno montañoso.

Se desarrolló un modelo del coche con diferentes parámetros como "links y joints" de ROS, inercias y caja de colisiones. Las estrategias de aprendizaje automático se aplican al modelo del coche, ajustando dinámicamente parámetros como velocidad y aceleración en respuesta a la retroalimentación en tiempo actual de la simulación.

La investigación demuestra que la integración de ROS con Gazebo proporciona una herramienta efectiva para desarrollar y probar políticas de aprendizaje automático aplicadas a entornos robóticos. Los resultados de la simulación sirvieron para entender cómo diferentes estrategias afectan el comportamiento del modelo bajo condiciones cambiantes, permitiendo la selección de la estrategia óptima basada en el tiempo necesario para el modelo en llegar a un objetivo.

Palabras Clave: Simulador, ROS, Gazebo, Coche, Machine Learning.

1 Introducción

El uso de simuladores para comprobar situaciones antes de llevarlas a la realidad es un ámbito tecnológico en auge desde hace tiempo. [1] Es fundamental el uso de estas técnicas para reducir el coste a la hora de desplegar una solución, así como mejorar y optimizar su rendimiento de manera rápida y eficiente. En muchos casos, plantear un entorno simulado presenta los problemas que pueden surgir antes de llevarlo a la realidad, por lo que también reduce significativamente el tiempo que se tarda en implementar una solución. [2]

Por esta razón es necesario plantear un simulador robusto, que permita una fiel simulación de los entornos y físicas planteadas. Esto es uno de los objetivos que presenta este trabajo, aunque no el único.

2 Definición del Proyecto

El proyecto se define con el objetivo de desarrollar un simulador que replique las condiciones físicas y mecánicas que pueden plantearse a un coche moviéndose a través de un terreno con altura variable. Se consideran factores como gravedad, fricción o colisiones mientras que se ignoran otros como viento. El objetivo no es desarrollar un coche que se asemeje completamente a la realidad, sino desarrollar un entorno que permita la simulación de diferentes técnicas de Machine Learning. Para ello se usan modelos y situaciones simplificadas como el coche o la montaña.

Para crear este entorno, se usan los programas de ROS (Robot Operating System) y Gazebo. Se han escogido éstos por las siguientes razones.

- **Desarrollo completo en código**: El uso de estas herramientas permite que el simulador sea desarrollado únicamente en código y que posibilite el ajuste de todos los parámetros a través de código programado. El único modelo que no se desarrolla completamente en código es la montaña, ya que, su perfil se ha creado en Autodesk Fusion 360, pero, todos los parámetros relacionados con su física, fricción o colisiones se definen a través de archivos .XML.

  Es importante considerar que hay otros programas que simulan mejor las condiciones deseadas. Sin embargo, al tener en cuenta otros factores, como el objetivo de estudiar políticas de aprendizaje automático, se ha decidido que este es el entorno óptimo.

- **Estándar en la Industria**: ROS y Gazebo son programas que se utilizan de manera común en la industria, por lo que desarrollar el entorno usando estos permite el aprendizaje de estas herramientas para un posible uso posterior, además de ser entornos con un futuro extenso y prometedor en el ámbito de la robótica.

- **Vanguardia de la simulación**: estas herramientas son usadas con mucha frecuencia debido a su fiel simulación de escenarios, así como por su continuo desarrollo que sigue mejorando sus capacidades.

- **Código abierto**: debido a que estos programas son de software libre para cualquiera que lo quiera usar, permite una transportabilidad y acceso a la herramienta que facilita su uso e implementación.

3 Descripción de las Herramientas

(a) **ROS (Robot Operating System)**: conjunto de herramientas y librerías principalmente usado para programar el modelado y comportamiento de robots. Utilizado para gestionar las interacciones y movimientos del coche dentro de la simulación.

(b) **Gazebo**: plataforma de simulación para renderizar el entorno 3D, permitiendo la visualización e interacción con el terreno montañoso creado. También se usa para simular las físicas e interacciones entre modelos.

(c) **RViz**: Herramienta de visualización integrada de ROS. Usado para simular el comportamiento del modelo sin que la simulación sea afectada por físicas. Principalmente necesario para comprobar el correcto funcionamiento de la arquitectura de Publicador-Suscriptor en el entorno de ROS.

(d) **Python**: lenguaje de programación usado para definir los comportamientos de la simulación e implementar los algoritmos de aprendizaje automático.

(e) **C++**: lenguaje usado junto con Python para el desarrollo de algunas partes críticas de la simulación como plugins de los modelos en ROS.

(f) **XML**: lenguaje de marcado para definir los componentes y links de un robot (en nuestro caso, del coche que se simula), así como otros elementos del entorno desarrollado.

Hay otras herramientas usadas para la creación y ejecución de la simulación los cuales se explican en detalle en el Capítulo 5.

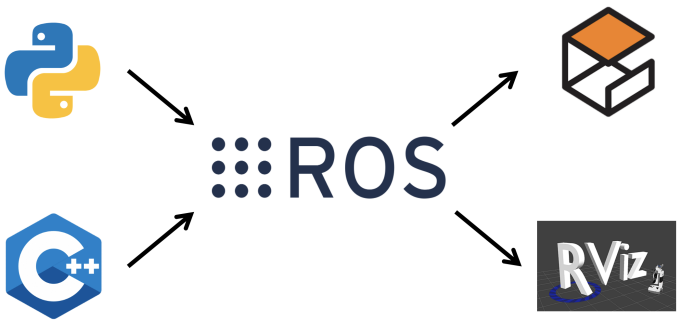

Figura 1: Arquitectura del Sistema

4 Resultados

Los resultados de la simulación muestran que el comportamiento del coche varía significativamente según las diferentes políticas de control aplicadas. Ajustando parámetros como la velocidad, aceleración y el control direccional basados en la retroalimentación en tiempo real del coche dentro de la simulación, se evaluaron tres diferentes estrategias por su eficacia en la navegación a través del terreno creado. El parámetro para medir la efectividad de cada protocolo es el tiempo en recorrer una distancia establecida.

Cuadro 1: Resultados

	Policy 1	**Policy 2**	**Policy 3**
Time	1m 38s	1m 18s	58s
Max Position	-12.064	-12.52	-11.94
Min Position	-23.448	-23.286	-23.06
Iterations	16	12	3

5 Conclusiones

El trabajo concluye que la integración de ROS con Gazebo proporciona una entorno capaz de desarrollar una herramienta de simulación para probar modelos de aprendizaje automático, y aplicarlos a entornos robóticos. La simulación resultó fundamental para comprender cómo diferentes estrategias se ejecutan bajo condiciones cambiantes, y cómo estas varían la respuesta de los modelos estudiados.

6 Referencias

[1] Edgardo J. Roldán-Villasana. "Importance of Simulators, Systematic Approach to Training, and Integral Instruction Centres in the Process Industry". In: 2015 IEEE European Modelling Symposium (EMS). 2015, pp. 157–162. doi: 10.1109/EMS.2015.33.

[2] Johannes Colditz et al. "Use of Driving Simulators within Car Development". In: DaimlerChrysler AG (2007).

SIMULATION OF THE DYNAMICS OF A MOUNTAIN CAR IN A ROS/GAZEBO ENVIRONMENT

Author: Jarauta Gastelu, Jaime

Supervisor: Sowers, Richard B., PhD.

Collaborating Institution: University of Illinois at Urbana-Champaign

ABSTRACT

This Bachelor's Thesis documents the development of a simulator for analyzing the dynamics of a car in a mountainous environment with changing altitude using ROS (Robot Operating System) and Gazebo. The aim is to implement and evaluate various machine learning strategies that control the vehicle's movement based on its instantaneous position and speed, and to examine how it navigates through terrain of varying height.

The methodology involves using simplified models of a car and a mountain to simulate physical interactions such as gravity, friction, and collisions. The simulation environment was created in ROS and Gazebo for several reasons, including the ability of these programs to develop environments that are entirely code-based and the capability to modify all simulation parameters through programming. Other programs used include Python and C++ for developing machine learning policies, XML for developing ROS environments, and Autodesk Fusion 360 for designing the mountainous setting.

A model was developed that includes various car features and parameters, such as ROS "links and joints," inertia, and collision boxes. Machine learning strategies are applied to the car model, dynamically adjusting parameters like speed and acceleration in response to real-time simulation feedback.

The research demonstrated that integrating ROS with Gazebo provides an effective tool for developing and testing machine learning policies applied to robotic environments. The simulation results were crucial for understanding how different strategies affect the model's behavior under changing conditions, allowing for the selection of the optimal strategy based on the time needed for the model to reach a target.

Keywords: Simulator, ROS, Gazebo, Car, Machine Learning.

1 Introduction

The use of simulators to test situations before trying them out in reality has been a field in growth for quite some time. [1] The use of these techniques is essential to reduce the cost of deploying a solution, as well as to improve and optimize its performance quickly and efficiently. In many cases, creating a simulated environment exposes the problems that can occur before having to take the situation to reality, thereby also significantly reducing the time it takes to implement a solution. [2]

For this reason, it is necessary to create a robust simulator, which allows for a realistic simulation of the environments and physics involved.

2 Project Definition

The project has the objective of developing a simulator that can accurately replicate the physical and mechanical conditions of a car moving through terrain of varying height. Factors such as gravity, friction, or collisions are considered, while others such as wind are ignored. The goal is not to develop a simulation that completely resembles reality, but to develop an environment that allows the testing of different Machine Learning policies applied to the car. For this reason, simplified models and situations such as the car or mountain are used.

For the project, ROS (Robot Operating System) and Gazebo programs are used. They have been chosen for the following reasons.

- **Complete code development**: The use of these tools allows for the simulator to be developed fully through code and enables the adjustment of all parameters through changing programming parameters. The only model that is not developed entirely in code is the mountain model, since its profile has been created using Autodesk Fusion 360, but regardless all parameters related to its physics, friction, or collisions are defined through .XML files.

  There are other programs and tools that simulate better the conditions being studied, but, considering other factors related to the project's focus such as the application of Machine Learning policies, it was decided that this is the optimal environment for this purpose.
- **Industry Standard**: ROS and Gazebo are programs commonly used in the industry, therefore developing the environment with them allows for learning on how these tools work and a possible later use. They are environments with an extensive and relevant future in the field of robotics and simulations.
- **Cutting-edge simulation**: these applications are used very frequently due to their realistic simulation of scenarios, as well as for their continuous development that keeps improving their capabilities.
- **Open source**: due to these platforms being open-source software, available to anyone who wants to use it, this allows for portability and access to them, facilitating the use and implementation of the environment created.

3 Description of Tools

(a) **ROS (Robot Operating System)**: used to program the models and behavior of the robots. Allows the user to manage the interactions and movements of the car within the simulation.

(b) **Gazebo**: platform for rendering the 3D environment, allowing for the visualization and interaction with the simulated mountainous terrain. Also used to compute the physics and interactions between models.

(c) **RViz**: Integrated visualization tool of ROS. Used to visually represent the behavior of the model without the models being affected by physical conditions in the simulation. Mainly used to debug and verify the correct functioning of the Publisher-Subscriber architecture within ROS.

(d) **Python**: Used to program the behavior of the car and to implement the machine learning algorithms.

(e) **C++**: Used along with Python for the development of some critical parts of the simulation such as the car model's ROS plugins.

(f) **XML**: Markup language used to define the components and links of a robot (in our case, the car model) and create other elements of the environment developed.

Other tools were used for the creation and execution of the simulation. They are explained in detail in chapter 5.

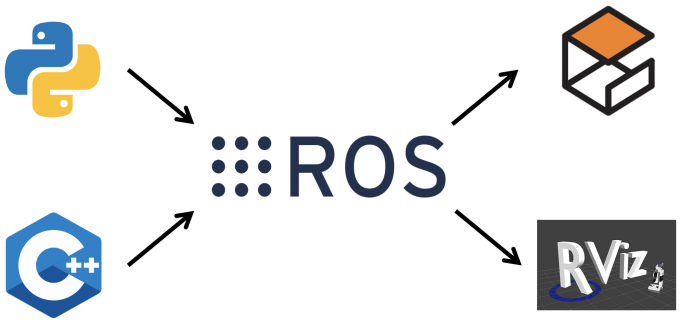

Figure 1: System Architecture Overview

4 Results

The results of the simulation show that the behavior of the car varies significantly depending on the different control policies applied. By adjusting parameters such as speed, acceleration, and directional control based on real-time feedback from the car within the simulation, three different strategies were evaluated for their efficacy in navigating the simulated mountainous terrain. Time is used as the parameter to measure the effectiveness of each policy.

Table 2: Results

	Policy 1	**Policy 2**	**Policy 3**
Time	1m 38s	1m 18s	58s
Max Position	-12.064	-12.52	-11.94
Min Position	-23.448	-23.286	-23.06
Iterations	16	12	3

5 Conclusions

The work concludes that the integration of ROS with Gazebo provides an environment capable of developing a simulation tool to test Machine Learning policies, and apply them to robotic environments. The simulation was fundamental in understanding how different strategies perform under changing conditions, and how these vary the response of the models studied.

6 References

[1] Edgardo J. Roldán-Villasana. "Importance of Simulators, Systematic Approach to Training, and Integral Instruction Centres in the Process Industry". In: 2015 IEEE European Modelling Symposium (EMS). 2015, pp. 157–162. doi: 10.1109/EMS.2015.33.

[2] Johannes Colditz et al. "Use of Driving Simulators within Car Development". In: DaimlerChrysler AG (2007).

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In recent years, the use of simulators has gained considerable importance across most scientific and engineering disciplines [1]. These virtual environments are fundamental for accurately recreating and studying complex real-world scenarios using advanced computational methods. By providing a controlled yet realistic setting, simulators allow researchers and engineers to analyze the potential outcomes and behaviors of systems under specific conditions before real physical implementation.[2]

This capability is crucial not only for enhancing the precision of theoretical models but also for validating and refining practical applications in a cost-effective manner. These tools allow for researchers to find problems before the system is implemented in real life, improving the time it takes to get final results. It is of great importance that these simulators offer a valid, accurate, and reliable representation of environments to ensure that the insights gained are both applicable and beneficial. The development of such tools, as demonstrated in this report, show the significant strides being made in simulation technology, paving the way for innovative solutions to engineering challenges.

This report on the "Development of a Mountain Car Dynamics Simulator in a ROS/Gazebo environment" outlines an approach to simulating vehicle dynamics on terrain of varying height. Its main goal is to create a setting that represents the movement of a car through a mountainous terrain with different physical parameters affecting its movement. Developed in partnership with the "University of Illinois at Urbana-Champaign" and "Universidad Pontificia Comillas" by Jaime Jarauta Gastelu under the guidance of Professor Richard B. Sowers.

The simulation environment is created to include diverse terrain elevations, friction, and kinetic energy components, with the main goal of implementing and testing various machine learning strategies to influence car behavior under differing conditions. Creating an environment like this, involves multiple engineering scopes [3]:

- **Mechanical engineering:** focuses on designing vehicles and specifically the mechanisms that enable movement.
- **Electrical engineering:** involves integrating systems, sensors, and communication into the simulation or its real counterpart. Handles visualization, simulation, and control through algorithms.

The project focuses on evaluating the efficacy of the different control policies on the mountain car's performance. Through the simulation, three distinct strategies are examined to observe the effects on the car's speed and trajectory. These methods apply velocities to the car of different value and direction depending on the current condition of the movement of the car. By analyzing the outcome when these situations are applied, the optimal policy can be found.

This project aims to refine the understanding of vehicle dynamics in a robot programming solution environment while serving as a valuable tool for testing different algorithms. The simulation results focus on the time it takes for the car to reach its target, providing a quantitative measure to understand the effectiveness of each policy.

Installation and configuration of the ROS and Gazebo on a virtual machine are critical components of the setup, ensuring a stable and reproducible environment for conducting the simulation. This report details the process of setting up the necessary software, constructing the simulation environment and code, and iterating through the different ML policies, as well as the steps it took to get there. It also discusses the challenges encountered during the development process and the solutions implemented to overcome them.

1.1 Motivation

The motivation of this project is to create a fully code-based simulation for the dynamics of the car. The reason for this is to have full control over all the parameters ruling over the simulation, therefore, being able to change all of them through code.

It is also important to mention the context of this project as a potential section of a future doctorate on reinforced learning and artificial intelligence capabilities by Ethan Torres at the University of Illinois.

Lastly, the motivation of this project can be seen as research related to the following paper published by Hossein Nick Zinat Matin and Dr. Richard B. Sowers, the director helping with the project.

[4] Hossein Nick Zinat Matin and Richard B. Sowers.(2020). "Nonlinear Optimal Velocity Car Following Dynamics (I): Approximation in Presence of Deterministic and

Stochastic Perturbations". In: vol. 2020-July. 2020.21
doi: 10.23919/ACC45564.2020.9147363.

This paper explores the complexities of car-following dynamics, focusing on the nonlinear optimal velocity (OV) model under both deterministic and stochastic perturbations. The research, investigates how the model behaves when subjected to these perturbations, and understanding how real-world factors like variable road conditions or driver behaviors can influence the model's predictions.

The project developed can be related to this one since it also centers on simulating car dynamics. Although it focuses on different parameters such as perturbations on a car rather than testing machine learning policies, the research share similarities.

1.2 Objectives

This section outlines the primary goals of the project, which are integral to the development and analysis of the simulation environment. Each objective is designed to build upon the last.

The following are the main objectives of the project.

1. **Develop a ROS environment.**

2. **Develop a ROS model** of a car with joints and links which are affected by physical parameters such as friction or gravity.

3. **Develop a Gazebo World** that includes a model with varying height in which the car can move through

4. **Tune the physics** of the simulation

5. **Apply forces and velocities** to the car model that are in line with the machine learning policies that are being studied

6. **Analyze the results** for time spent towards reaching the final goal and find out which is the optimal strategy

Upon the finalization of the research, all of these objectives were met with the exception of fully optimising the behaviour of the car and fixing the control that changes the yaw of the car while moving due to factors that will be explained in chapter 10.

1.3 Methodology

The development of the environment is based on a trial-error method. Firstly, an initial version is created making sure that all the parameters established are working properly, to afterwards add more functionalities as the project evolves. When it comes to the research, it is mainly code-based research, therefore the following are some of the programming languages used: python, C++, xacro, as well as the files used for the simulation, which include (but are not limited to): .launch, .world, .urdf, .rviz, .sdf, .yaml. These files and programming languages are explained in chapter 2.

Weekly meetings were held with Professor Sowers in order to update on the work done, propose new ideas and guide the project.

When it comes to the resources used for the research, guidance from academic articles, online repositories and research thesis were used, which are referenced in this report, as well as official guides from the ROS and Gazebo platforms. Assistant professors at the University of Illinois or Universidad Pontificia Comillas were consulted in the case that problems arose for which a solution was not found.

1.4 Plan

- **October 2023**: Project Definition. Set up virtual machine and familiarize with ROS basic concepts. Study the ML documentation provided to fully understand the scope of the problem. Research the packages and libraries needed for the project.
- **November 2023**: Develop initial version of car and mountain models. Set up a ROS environment with basic capabilities such as packages and nodes. Create a Gazebo world and import the models into it.
- **December 2023**: Input Physics Parameters into simulation such as Gravity and Friction. Consider creating other parameters such as wind.
- **January 2024**: Create initial movement through an interface. This is human inputted and does not work automatically.
- **February 2024**: Create first python script that controls car movement automatically.
- **March 2024**: Develop python script for the different Machine Learning policies to be studied.
- **April 2024**: Create videos of the policies tested and analyze the results obtained.
- **May 2024**: Write the final report and document the development process. Improve the simulation as much as possible.

Chapter 2

Tools

The following section explores the set of tools employed in this project, providing an overview of their history, common applications, and specific use within this research. This includes an in-depth look at ROS and Gazebo, alongside programming languages like Python and C++ which are used for script development and system operation. The different file types needed for the system are mentioned and explained as well.

2.1 ROS

ROS (Robot Operating System) is a collection of libraries and tools used to simulate environments involving robots. [5] It was developed out of the need for a unified language around projects that require robotic capabilities. ROS was created in 2007, by Eric Berger and Keenan Wyrobek, PhD students at Stanford University, with its first official release occurring in 2010. The initial version was developed at a Silicon Valley robotics lab named "Willow Garage". [6].

Before this environment existed, each robotics project used different systems and programs, which prevented the export and execution of projects developed by separate people or environments, thus requiring extensive code adaptation work. This environment was also created with accessibility in mind for anyone wanting a free and usable tool, similar to what Linux had become at the time. [7]

ROS includes numerous packages that allow for the development of any robotic field, as well as the ability to create custom packages specifically designed for any need. While ROS 1 only allows its use on Linux, the new version (ROS 2) also includes access to platforms like Windows or MacOS. This newer version of the OS also includes some of the following features: [8] [9]

- **DDS Integration:** uses Data Distribution Service.
- **Real-Time Support:** includes capabilities for real-time computing in robotics applications.
- **Improved Security:** built-in security features including encryption, authentication, and secure node handling.
- **Modularity:** improved modularity for core components.
- **Command Line Tools:** improved command line tools for an easier system management.
- **Multi-Robot Systems:** better support for managing and coordinating multiple robots in a system.

Even though the newer version of ROS 2 was available at the time of research, the older ROS 1 version was preferred for the following reasons: [10]

- **Maturity and Stability:** due to the system being available and improved since 2007, the environment is more stable than its newer counterpart.
- **Extensive Library and Tools:** includes a vast array of libraries that help with specific tasks related to the simulation.
- **Lower Computational Requirements:** due to running the simulation on conventional computers, a lower computational requirement was preferred in order to not overload the systems. [11]

ROS is completely open source and supported by numerous industries, which has allowed it to become the standard when it comes to robotics-related projects. This is the reason why ROS was chosen as the program for this research. [12]

Figure 2.1: ROS Logo
Source: Courtesy of ROS [13]

2.2 Gazebo

Gazebo is an open-source robotics simulator initially developed as part of the Player Project (open-source project for robotics and sensor research) in the early 2000s. It features advanced 3D graphics and support for multiple physics engines like ODE, Bullet or Simbody, allowing realistic simulations of robots with virtual sensors and actuators. [14] [15] [16]

The integration of Gazebo with the Robot Operating System (ROS) significantly increased its functionality, making it a fundamental tool for testing and validating robot software. This combination allowed developers to test robot designs in a controlled setting, reducing the cost and risk associated with physical prototyping. This integration occurred in "Willows Garage" robotics lab, where ROS underwent significant development. Therefore ROS was developed with Gazebo in mind as its simulation environment since its conception. [6]

Figure 2.2: Gazebo Logo

Source: Courtesy of Gazebo [17]

Nowadays, Gazebo is used to recreate models, commands, sensors and other elements related to a robotics environment in a computer simulation before running the real model. It is used to test the physicality and validity of these environments. Gazebo has a very wide array of purposes such as robotic arms, models or vehicles.[18]

While Gazebo is more frequently used for the simulation of indoor environments, it also supports outdoor situations like this project. For this reason and others, it has the capability of integrating different physical engines such as ODE or DART, each designed to simulate specific types of robotics dynamics, making it a fundamental complement to ROS. The architecture of Gazebo is structured around a client-server model, where 'gzserver' handles the physics, rendering, and sensor data processing, while 'gzclient' provides a graphical interface for interaction with the simulation, and a topic-based Publisher/Subscriber communication model. [3] [5]. For more information on how Gazebo works, refer to chapter 5.

2.3 Linux/Ubuntu

Linux is an open-source operating system kernel first released by Linus Torvalds in 1991, which has since evolved into various distributions that cater to the different needs of users. [19] Linux has become renowned for its robustness, security, and performance, making it a preferred choice for servers, desktops, and embedded systems alike.

Its open-source nature allows developers to modify and distribute their versions of the OS. Ubuntu, introduced in 2004 by Canonical Ltd., is the distribution selected and used for this project. [20] In order to run ROS 1, a Linux OS (specifically with the Ubuntu 20.04 distribution version) was created through a Virtual Machine.

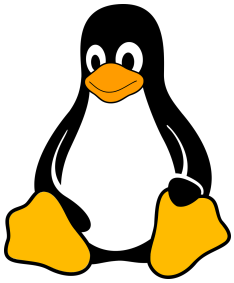

Figure 2.3: Linux Logo
Source: Courtesy of Linux [21]

Figure 2.4: Ubuntu Logo
Source: Courtesy of Ubuntu [22]

2.4 Git

Git is a tool for version control in software development. Developed by Linus Torvalds in 2005, the creator of Linux, Git has become essential for many modern software development processes. [23] Git is the source control used throughout this project to keep, maintain and update all code developed.

Specifically, a GitLab repository with a University of Illinois License was set up as the version control.

Figure 2.5: Git Logo
Source: Courtesy of Git [24]

Figure 2.6: GitLab Logo
Source: Courtesy of GitLab [25]

2.5 Visual Studio Code

Visual Studio Code (VS Code) is an open-source code editor developed by Microsoft. Released in April 2015, it is one of the most popular development environments among programmers [26]. Designed to be a lightweight yet powerful source code editor, VS Code can run on Windows, MacOS, and Linux platforms. It includes capabilities for debugging, syntax highlighting and others through the installation of plug-ins.

For this project, it was used to develop code in the languages and file types presented in section 2.8.

Figure 2.7: Visual Studio Code Logo
Source: Courtesy of VSCode [27]

2.6 CAD Tools

For the development of the CAD model of the hill, a set of tools were used to design and process the profile.

Firstly, "Autodesk Fusion 360" was used to create the model with the parameters required. Fusion 360 is an integrated, cloud-based CAD software developed by Autodesk, launched in 2013. It allows users to perform a range of functions including 3D modeling, computer-aided engineering, and manufacturing. [28]

Afterwards, the open source program "Blender" was used in order to edit minor things from the model, and to export it into a ".dae" Collada file that is able to be read by ROS. Blender is a free and open-source 3D creation suite, first released in 1998 by the Blender Foundation. It supports the entirety of the 3D necessary tools such as modeling, animation, simulation, rendering or motion tracking, as well as video editing and game creation. [29]

Figure 2.8: Autodesk Fusion 360 Logo
Source: Courtesy of Autodesk [28]

Figure 2.9: Blender Logo
Source: Courtesy of Blender [29]

2.7 Other Programming Languages and Tools

The following are some of the programming languages, files, libraries and other tools used in the simulation environment. All of these have been used extensively for the development of the environment.

2.7.1 Programming Languages

1. **'.py' (Python)**: Python scripts are widely used for writing ROS nodes, scripts, publisher-subscriber networks, services and others. Python is also used for applying the Machine Learning policies to the models that are being tested. [A.3.1, A.3.4].

2. **'.cpp' (C++)**: another primary programming language for ROS. It is used for writing ROS nodes and Gazebo plugins.

Figure 2.10: Python Logo
Source: Courtesy of Python [30]

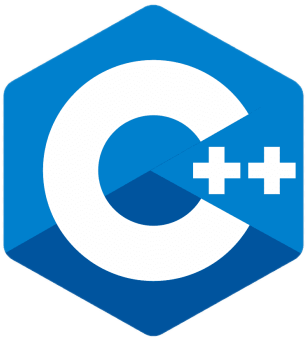

Figure 2.11: C++ Logo
Source: Courtesy of C++ [31]

2.7.2 Tools and Libraries

1. **RViz (ROS Visualization)**: ROS integrated tool used for 3D visualization of robots, sensors and other ROS model parameters. Even though it may seem similar to Gazebo as it presents an environment, RViz does not have the capability to simulate physics as Gazebo does. It was mainly used throughout this project for debugging purposes.

2. **rqt**: Qt-based framework for GUI development, which includes a wide range of plugins for visualizing and interacting with a ROS system. For the initial version of the system in which the car's linear and angular speed is controlled manually, a rqt interface is used.

3. **CMake**: build system that generates makefiles and software compilation processes within an environment.

4. **catkin**: build system that allows for workspace management and compilation. It simplifies the creation of ROS packages that create environments for simulation.

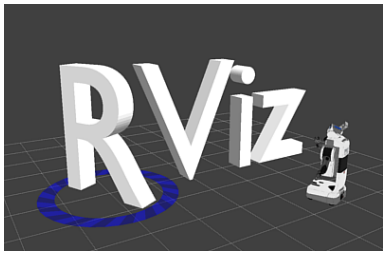

Figure 2.12: RViz Logo

Figure 2.13: CMake Logo
Source: Courtesy of CMake [32]

2.8 Files

1. **'.launch' (Launch File)**: XML files used by ROS to start up multiple nodes and set parameters. They can include other launch files, allowing complex systems to be started quickly and consistently without having to run an excessive number of commands. [A.1.1, A.1.2]

2. **'.urdf' (Unified Robot Description Format)**: XML files that describe the physical and visual aspects of a robot model, including its kinematics, dynamics, and geometry (such as links, joints or sensors). URDF are the types of files used to declare the car model in the simulation. [33]

3. **‘.xacro‘ (XML Macro)**: files used to define robot structures. They are preprocessed into URDF files, allowing for parametrization and modularity in the declarations. [A.4.4]

4. **‘.txt‘ (Text File)**: used for configuration, lists of parameters, or simple data storage in both ROS and Gazebo. Within this research, they are mainly used for package and environment general data. [A.4.1]

5. **‘.rviz‘ (RViz Configuration File)**: files which contain configuration settings for RViz, specifying which topics to subscribe to, how to display data, and the visual properties of the simulation. [A.4.5]

6. **‘.world‘ (Gazebo World Files)**: XML files that define the simulation environment in Gazebo. They specify and insert the models, lighting, physics, and other properties of the simulation world. [A.4.3]

7. **‘.sdf‘ (Simulation Description Format)**: XML format used to describe objects for robotics simulations in Gazebo. It's more an addition to URDF files that allow to declare position, friction, inertial elements and other properties for a model. The hill model as well as some of the initial versions of the car model in the simulation are declared using .sdf files. [A.2.3] [5]

8. **‘.yaml‘ (Yet Another Markup Language)**: used for defining parameters for ROS nodes, robot descriptions, and other settings. The final version of the car model is defined using this type of file. [A.2.1, A.2.2]

9. **‘.dae‘ (Collada Digital Asset Exchange)**: XML-based files used to describe digital interactive 3D objects. These files are used to define both visual and physical properties of models in simulations and rendering. They include details such as textures, materials, and mesh information. The hill model's profile is declared using collada files that were exported using Blender Software.

Chapter 3

State of the Art Analysis

This section examines a number of papers, articles, and projects related to the research presented, discussing their similarities and differences, as well as the new elements each one introduces to the field of robotics simulations. It aims to find key methodologies, frameworks, and application areas that have been used in previous robotic simulation technologies and either apply them to this research or discuss their differences and how this project can expand on these topics.

Furthermore, this overview looks to find gaps in the existing research, suggesting potential ways for future development. By studying these different sources and contributions, a reasonable snapshot of the state-of-the-art situation in robotics simulations can be found.

3.1 "Car dynamics model - Design for interactive driving simulation use"

[34] Bouchner, P., Novotný, S. (2011). Car dynamics model - Design for interactive driving simulation use. Recent Researches in Applied Informatics - Proceedings of the 2nd International Conference on Applied Informatics and Computing Theory, AICT'11.

This first paper, developed by Peter Bouchner and Stanislav Novotný at the Czech Technical University in Prague, describes the development, implementation, and testing of a mathematical software model designed to present car dynamics for interactive driving simulations. It is focused on human use therefore it includes virtual reality to provide a comprehensive system for driving simulators. Regardless, it is still relevant to this research given that the paper focuses on simulating a car environment.

The model consists of two main components: a virtual reality generator that creates

3D graphics and sounds, and a physical model of vehicle dynamics. It is more focused towards simulators used by humans to present a realistic version of a car, which can be used for training. When it comes to its relation with this project, the most important section comes from the "Physics Simulation Engine" developed, which controls elements such as the rate of depression of brake and throttle pedal or steering angle.

The paper presents a figure seen below in Figure 3.1 which explains the architecture of the physics engine, which is in some way similar to the project presented in this report due to the fact that in both simulations there is a controller that acts upon the car considering its instantaneous conditions.

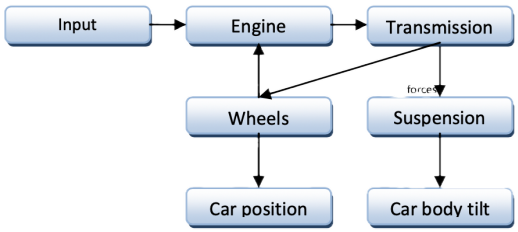

Figure 3.1: Arch. of the physics engine
Source: [34]

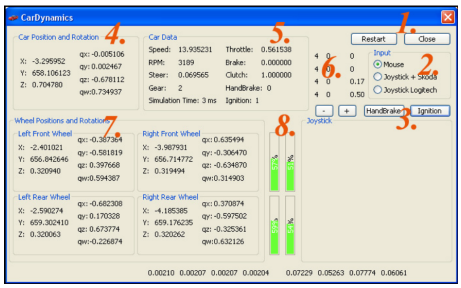

Figure 3.2: Front end to car model
Source: [34]

As seen in Figure 3.2, there are car simulation parameters that are updated as the simulation is run that are similar to the ones this research includes in Figure 7.1.

Lastly, the paper also discusses the simulator's layered structure which allows for flexibility, as well as its modularity to remove or add new components as needed. This modularity is similar to what ROS allows in our environment by smoothing the process of including new models into the simulation. It is mentioned the importance of balancing model accuracy and computational feasibility, which applies as well in our case due to the fact that when running the ROS simulation, computational capability limits the speed at which the simulation can run as conventional computers are being used for the simulation.

3.2 "Modeling and identification of passenger car dynamics using robotics formalism"

[35] Venture, G., Ripert, P. J., Khalil, W., Gautier, M., Bodson, P. (2006). Modeling and identification of passenger car dynamics using robotics formalism. IEEE Transactions on Intelligent Transportation Systems, 7(3). https://doi.org/10.1109/TITS.2006.880620

This second article presents a new method for modeling and identifying the dynamics of passenger cars using robotics techniques. They use a using the recursive Newton–Euler algorithm to develop a dynamic model of a Peugeot 406 passenger car that correlates with its dynamic parameters. Their approach is validated through both simulation and real-world experiments using a real car. The goal is to accurately predict car dynamics to help in the design and tuning of prototypes.

These are some of the following vehicle dynamics parameters that they use for the simulation. The ones in **bold** are considered in this project as well, showing the similarities between the paper and this project.

- **Chassis**
- **Steering system**
- Suspension bars
- Unsprung bodies
- **Wheels**
- **Track width**
- **Wheelbase**
- Suspension clearance
- Toe and steering angle
- Camber angle
- Kingpin angle
- **Wheel Rotation**
- Tire deflection
- **Longitudinal trans.**
- **Lateral translation**
- **Vertical translation**
- **Roll**
- **Pitch**
- **Yaw**

This model also considers parameters such as friction, inertia, mass, velocity, acceleration and orientation, all of which are studied in this research. The main difference is that it is more mathematically focused, whereas the physics simulation presented in this thesis is done by Gazebo using an "ODE" physics model. The dynamic model's effectiveness in the shown paper is tested through comparing the data obtained to an experimental vehicle equipped with sensors whereas ours is not implemented in real life.

3.3 "Unmanned ground vehicle modelling in Gazebo and ROS-based environments"

[3] Rivera, Z. B., de Simone, M. C., Guida, D. (2019). Unmanned ground vehicle modelling in Gazebo/ROS-based environments. Machines, 7(2).
https://doi.org/10.3390/machines7020042

The article explores the modeling of unmanned ground vehicles (UGVs) within Gazebo and ROS environments, showing the integration of these systems for effective simulation and design. The study focuses on waypoint navigation activities for a custom-designed wheeled mobile robots (WMR) in a simulated 3D indoor environment. The model developed is a three wheeled robot with sensors and actuators that move it throughout an environment, similar to the one presented in section 4.3.

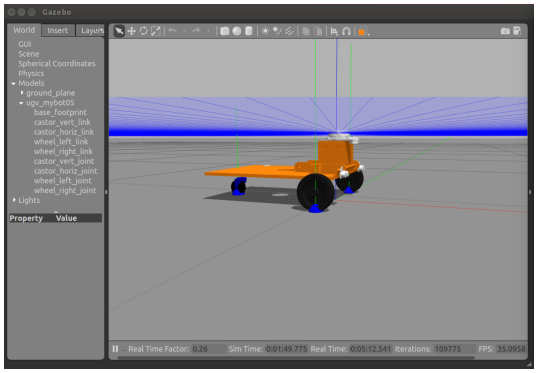

Figure 3.3: Robot Presented in [3]

This setup demonstrates the use of Gazebo's high-performance physics and sensor emulation to test and develop robotic prototypes effectively, simulating realistic operational scenarios. This is fundamental for the project being developed as it showcases the realistic nature of this environment. For this article, the robot simulated is also recreated in real life which allows for comparison between the simulations and the actual car.

The paper also discusses the technical aspects of Gazebo's configuration, including the creation of dynamic multibody models (like the car being simulated in our project which is similar to the one presented in this article) and the integration of control systems using MATLAB and Simulink.

The article presented resembles this project as it is based on a ROS/Gazebo simulation and has a similar robot in its simulation. It also utilizes tools such as RViz and files such as .sdf or .urdf to declare the model in Gazebo and includes a PI/PID control to vary the movement of the car. However, it differs as the environment simulated is flat, whereas in this project it is of varying height. The article also focuses on comparing a simulation with its real counterpart whereas our research focuses on testing Machine Learning Policies.

3.4 "Vehicle Dynamics Model for Simulation Use with Autoware.ai on ROS"

[36] Roger W. Zeits. 2023. Vehicle Dynamics Model for Simulation Use with Autoware.ai on ROS. Master's Thesis, The Ohio State University, Department of Mechanical and Aerospace Engineering.

This thesis investigates the development of a vehicle dynamics model designed for simulation applications integrating Autoware.ai with ROS. It is focused on improving the fidelity of simulations used to test autonomous driving technologies, considering the limitations of real testing.

The research focuses on taking a 2017 Lincoln MKZ Hybrid production vehicle through different acceleration and steering tests and comparing it with a simulation performed using ROS and Autoware.ai. This allows for a controlled and repeatable testing environments. The dynamics model aims to reproduce the real-world vehicle behavior in a virtual setting in order to compare the results through calibrating and validating the model against actual vehicle performance data.

The following are some of the tests performed and compared in the thesis in order to validate the capability of ROS and Autoware.ai to simulate faithful real environments.

- Positive and Negative Acceleration Test
- Straight Line Path Test
- Curved Path Test
- Constant Radius Turn Test

By using ROS and Autoware.ai, the research illustrates how the dynamics model can be incorporated into a simulation framework that accurately replicates real driving conditions. This thesis relates to the research in this report as it is ROS-based, and, while Autoware is used as the autonomous driving system instead of simulating the environment in a platform such as Gazebo, it still presents a car with similar characteristics as the one that is being used in this project. In our case, while the ML policies can be applied for autonomous driving, self-driving is not the main focus of the research.

This thesis highlights the implications of using advanced simulation tools to perform extensive testing without the high costs and risks associated with physical testing. This approach significantly accelerates the pace of development, pushing the boundaries of what can be tested in a virtual domain.

3.5 "ROS-based Simulation for Mixed Reality Test Systems for Autonomous Vehicles"

[37] Zofka, M. R., Tottel, L., Zipfl, M., Heinrich, M., Fleck, T., Schulz, P., Zollner, J. M. (2020). Pushing ROS towards the Dark Side: A ROS-based Co-Simulation Architecture for Mixed-Reality Test Systems for Autonomous Vehicles. IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, 2020-September. https://doi.org/10.1109/MFI49285.2020.9235238

This paper presents an approach to improving the validation and verification of autonomous vehicle systems using ROS. This architecture is designed to integrate mixed-reality environments, allowing for realistic testing scenarios that include real world elements with simulated models. It focuses on creating complex traffic scenarios involving both virtual and real components, which allows for a deeper understanding of how autonomous vehicles perform under varied and unpredictable conditions.

The main similarities with the project presented here are that the environment uses ROS as its main framework, using its capabilities to integrate various models in the world environment, as well as vehicle dynamics. Tools like RViz are also used to recreate the situation.

There is as well some previous research by the same authors that utilizes Gazebo as the main physics simulator.

[38] Zofka, M. R., Klemm, S., Kuhnt, F., Schamm, T., Zöllner, J. M. (2016). Testing and Validating High Level Components for Automated Driving: Simulation Framework for Traffic Scenarios. 2016 IEEE Intelligent Vehicles Symposium (IV), 144-150.

In this case, they make use of ROS, Gazebo and SUMO (Simulation of Urban Mobility) to create a simulation with cars, pedestrians, streets and other elements in order to evaluate automated driving systems under varied traffic conditions.

The main differences are that, even though these projects are based on a simulation environment for a car, they are more focused on a mixed-reality based system, while the research done in this report focuses more on the simulation of the dynamics of a car without considering daily real world factors such as those presented. It only considers physics factors such as friction, height or collisions. Regardless, the tools used for the projects are similar.

3.6 "Simulation environment for mobile robots testing using ROS and Gazebo"

[5] Takaya, K., Asai, T., Kroumov, V., Smarandache, F. (2016). Simulation environment for mobile robots testing using ROS and Gazebo. 2016 20th International Conference on System Theory, Control and Computing, ICSTCC 2016 - Joint Conference of SINTES 20, SACCS 16, SIMSIS 20 - Proceedings. https://doi.org/10.1109/ICSTCC.2016.7790647

The article discusses the development of a simulation environment for mobile robots using ROS and Gazebo. This environment allows for the testing of robot behaviors and control strategies in a virtual space before real physical deployment. The paper shows the ability to directly transfer code used in simulations to real world robots. This is relevant to the project presented because, even though the real aspect of the simulation developed is not being studied, it shows the capability of a platform such as ROS/Gazebo to present accurate simulation modelling.

The work also serves as a guide for constructing 2D and 3D environmental simulation models within Gazebo, and for simulating robot models within these environments. The main contribution of this research is its detailed explanation on the integration of ROS and Gazebo for developing a reliable control system for mobile robots, which can be used as a reference for the project developed. The robots in question are programmed to have different sensors such as LIDAR and cameras in order to create a 2D and 3D representation of an environment and display it using Gazebo and RViz.

This is likely the most similar project to the one presented in this report of those analyzed, but it does not present a situation in which the car has to move through varying heights. It only must consider 3D physical objects and how to move around them.

3.7 "Conclusion"

Analyzing the difference between the projects presented and the one developed for this thesis, it is important to mention the following differences and advancements:

Unlike previous studies that primarily explore vehicle dynamics in typical flat driving or robotic navigation scenarios, this project employs a specific world environment where a car must navigate steep, variable terrains. This adds a layer of complexity by varying friction levels and landscapes, which are not commonly addressed in standard vehicle simulation environments.

Also, the project aims to evaluate the efficacy of different machine learning strategies in real-time control scenarios, analyzing the results of how varying testing policies affect performance, which is something that was not seen in the papers and articles presented. While most of the previous research focused mainly on simulating cars for an autonomous driving situation, this investigation looks to test out and analyze the optimality of different policies that affect the response of the car depending on its instant situation.

Chapter 4

System Development

The following chapter describes the creation and development of the environment developed for our research. This includes designing the hill model through which the car moves, the different versions of the car as well as the control systems that govern the car's movement.

This section also delves into the methodologies used to simulate the environment and discusses the parameters, software tools, and feedback mechanisms that ensure the car's response to these simulated conditions. Lastly, this chapter illustrates the integration of multiple software tools, including Autodesk Fusion 360, Blender, ROS, and Gazebo, to create a cohesive and interactive simulation environment.

4.1 Hill Development

The iterative design and development process of the terrain model on which the car moves is a critical component of our simulation environment. This section explains the different versions of the hill model, each modified through iterations to better meet the requirements of our simulations.

All of these models were designed using Autodesk Fusion 360, a 3D CAD tool as explained in section 2.6. Afterwards, these models were exported to a Collada (.dae) file format using Blender, another 3D graphics tool, to ensure readability within the ROS/Gazebo simulation software.

Each version of the hill is discussed in detail below, with graphical representations and descriptions to show the evolution of its design and the implication on the simulation's outcome.

4.1.1 Version 1

This first version of the hill presented a profile similar to the one studied in the machine learning presentation [39], but, which did not have exactly the same graph function as the one wanted. It was built to understand and test out the capability of inserting a model into the Gazebo world. Initially, in the simulation created for this design, which is explained in detail in section 4.2, the "ambulance" model was inserted at a position over the hill and through gravity, it moved downwards to slide down the hill and stop once the model lost all its energy through friction losses.

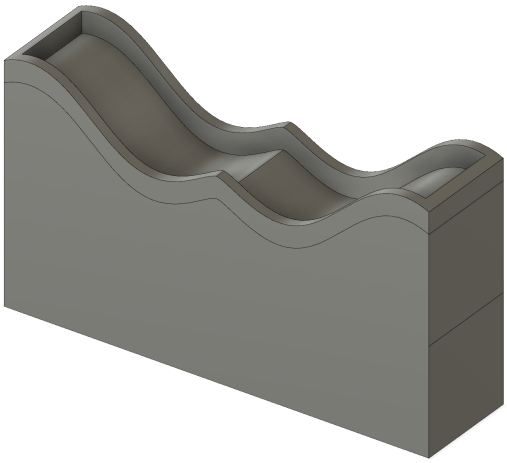

Figure 4.1: Hill Version 1

4.1.2 Version 2

After the initial testing and evaluation of the first hill model, a newer version was created to better match the wanted profile. The aim was to refine the hill's geometry to align more closely with the following mathematical formula:

$$U(x) = 0.45\sin(3x) + 0.55 \quad x \in \mathbb{R}$$

This formula represents a sinusoidal wave modified to fit the requirements of our simulation, creating a complex terrain for testing the vehicle's capabilities. This function presents the ability to introduce a model of varying slopes and height. In Figure 4.2, you the mathematical graph of the formula presented above can be seen.

The formula for the slope is the following, obtained by differentiating the original expression:

$$U'(x) = 1.35\cos(3x) \quad x \in \mathbb{R}$$

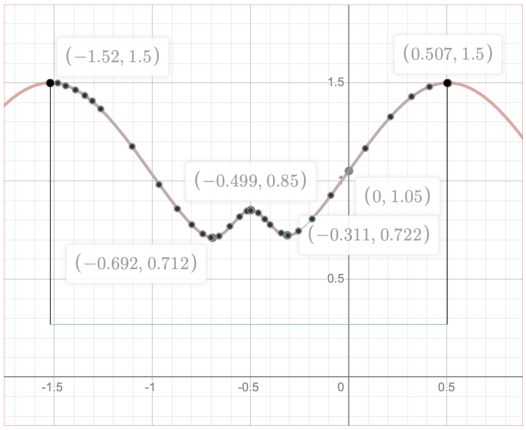

Figure 4.2: Hill Version 2 Profile

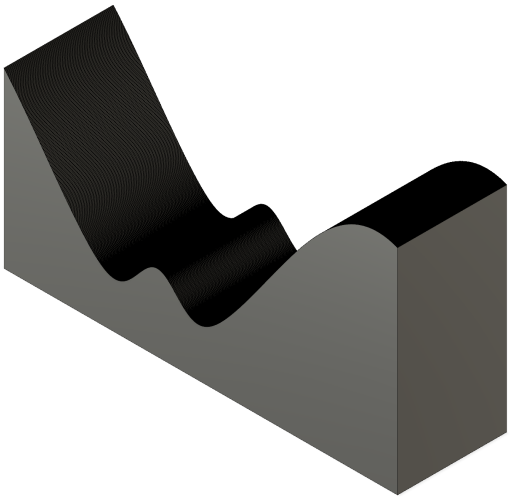

Figure 4.3: Hill Version 2

4.1.3 Versions 3 and 4

Finally, the last two versions present the section of the hill that is being studied rather than the whole profile which are from x = -1.2 to x = 0.45.

There is also as a newer version that is scaled in the vertical axis by a fourth of its size in order to allow the car to move more easily through the hill (as it was not able to do so if the profile was the original one). Its formula is the following:

$$U(x) = 0.1125\sin(3x) + 0.55 \quad x \in \mathbb{R}$$

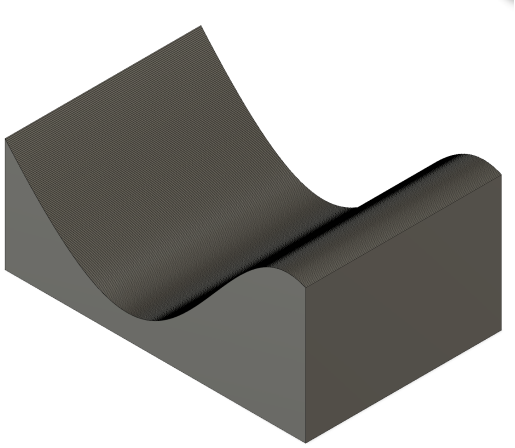

Figure 4.4: Hill Version 3

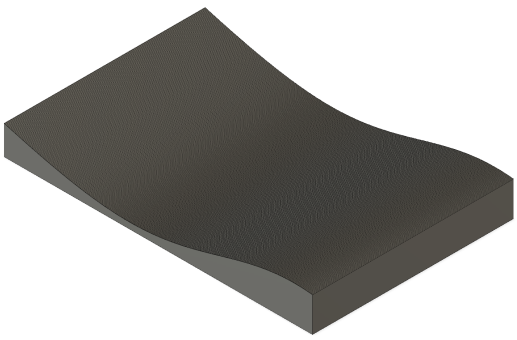

Figure 4.5: Hill Version 4 Scaled x0.25 in Vertical Axis

4.2 Initial Simulation Environment

The initial steps of the simulation consisted of the creation of a Gazebo world, with a model in it. This can be done both through inserting the car once the world was initialized, as well as doing so through the .launch file. The second option was preferred due to the ability to choose precisely the conditions in which the model is introduced (such as pose or angle) and create repeatability.

After some trial and error, a standard ambulance model included in Gazebo was input to the world but, at the time, it was not affected by gravity.

Afterwards, the initial hill model was included. This was harder as it had to be included through the my_mesh.world file instead of inserting it through the models window in Gazebo as is explained in chapter 11. [40]

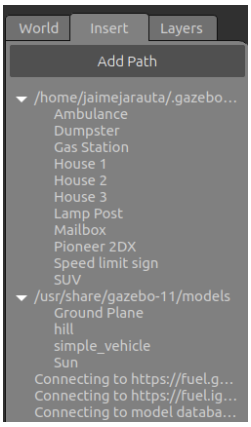

Figure 4.6: Gazebo Insert Model Screen

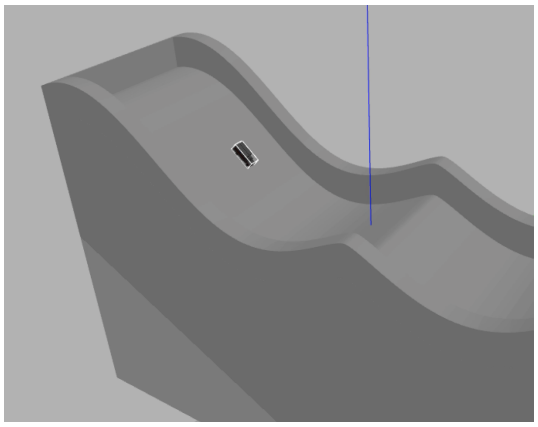

Figure 4.7: Ambulance Simulator

In Figure 4.6, the screen used to input models straight into the Gazebo environment is shown.

Initially, the models did not interact with each other as they were static and gravity was not included in the world. The first time gravity was included in the simulation was through the Gazebo World Parameters Screen, which provided a place to include the value for gravity and the selection of the physics engine. This screen can be seen in Figure 4.8. [41] As seen in that same figure, there is a screen within the Gazebo app that allows both for the visualization of the state of each model in the simulation, with the possibility of checking other parameters such as their position, properties etc.

The problem with this was that every time the world was initialized, the gravity had to be input manually in order for it to affect the models. That is the reason why the next development objective consisted on inserting the values for gravity, physics engines, collisions and others directly through the .world file located in subsection A.4.3. That

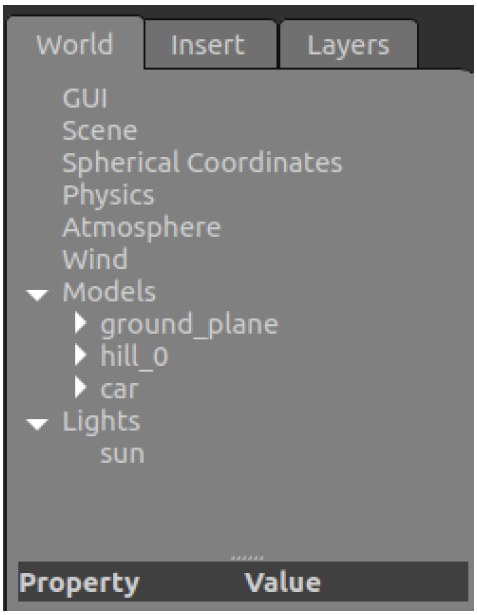

Figure 4.8: Gazebo World Parameters Screen

way, every time ROS and Gazebo were initialized, the models would already be affected by all the physics and collisions parameters desired.

After doing this, the world consisted of two models (ambulance and hill), that were initialized to a certain position (determined in the .world file) which were affected by gravity. This made the ambulance fall from its original position and move through the hill until it stopped due to friction, but, at this point, there was no tool to control the velocity or direction of travel of the model.

4.3 Simple Robot

The second version of the simulation included a world developed for creating and testing a publisher-subscriber network described in chapter 7. For this, a simple car model with two wheels was developed. [42]

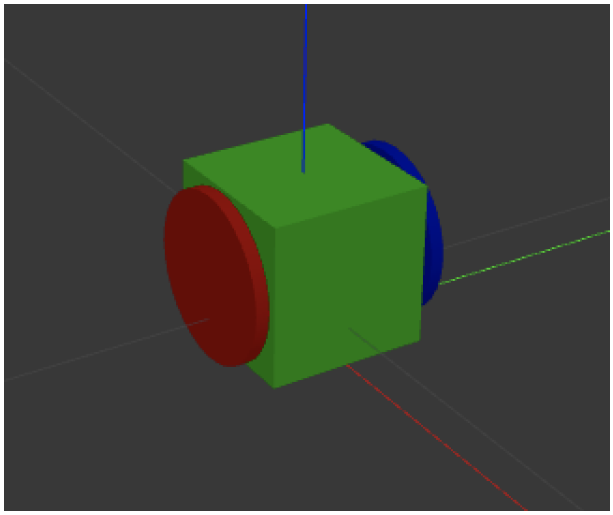

Figure 4.9: Simple Robot

Using an rqt plugin [43], the environment was able to control the car's speed. The constraint with this simulation was that there was a limit to how the velocity was applied to the model (both for its value and because the car had to be controlled through a user interface), therefore it was hard to replicate the ML policies wanted to be tested out. [44]

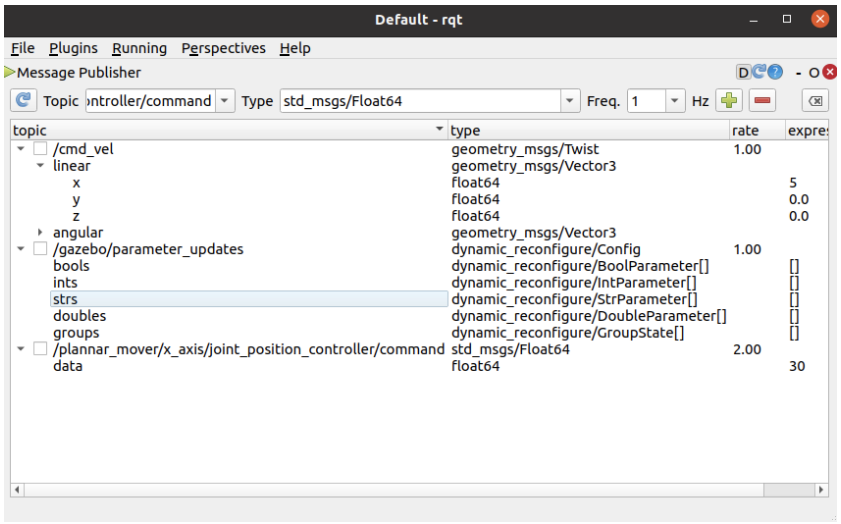

Figure 4.10: RQT Speed Interface

Through the GUI presented in Figure 4.10, the user was able to select the topic to which to publish the speed (seen in the top left of the image), the type of message sent

(in the case of velocity it was a Twist type, explained in chapter 7), as well as the value and rate of publishing. The problem appearing is obvious when considering that the change of speed when applying the Machine Learning policies has to be precise and at the exact moment that the car's real speed changes, therefore a newer version that applies the strategies automatically had to be developed.

4.4 RViz

One of the following developments of the system was the discovery and use of RViz. As seen in subsection 2.7.2, this is a tool to simulate visually the state of the environment but without considering physics. It only shows the actual value of the topics published and the state of the models.

This was very useful to debug the system and be able to check whether a topic was being published correctly or not. It was used in conjunction with Gazebo to check the state of the environment when running.

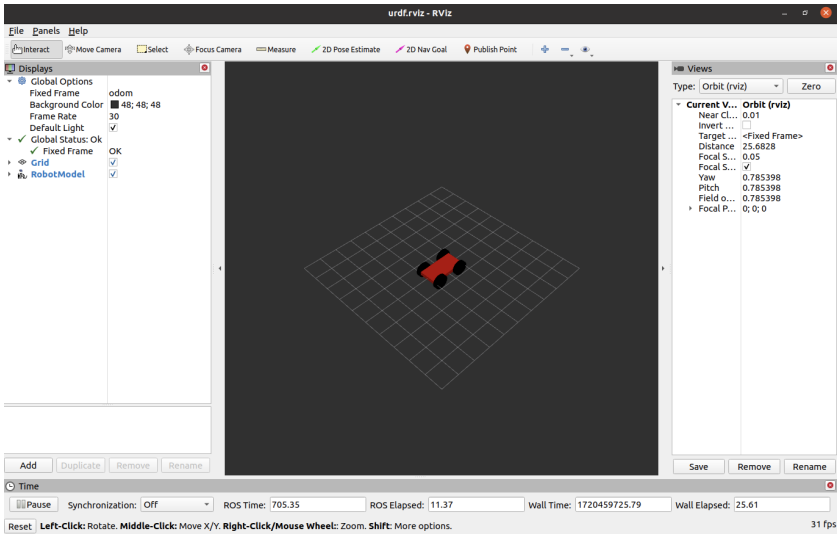

Figure 4.11: RViz Screen

4.5 Differential Drive Robot

The third version of the simulator included a differential drive robot that was able to be controlled through three methods.

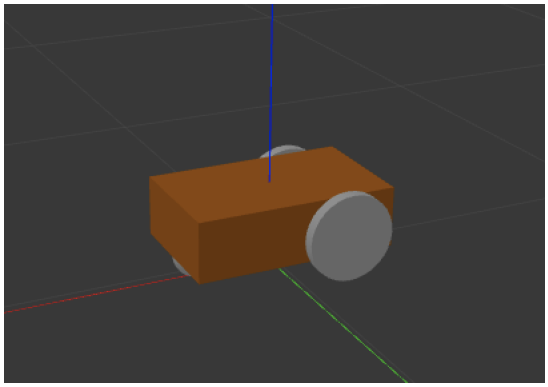

Figure 4.12: Differential Drive Robot

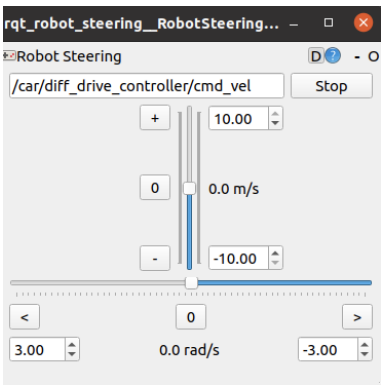

Figure 4.13: RQT GUI

The first was publishing the value for the speed to a ROS topic. This presented the problem that it was hard to change instantaneously the speed depending on the direction of the velocity. This was done through the terminal using a command like the following: [45]

```
rostopic pub /simple_robot/left_wheel_speed
std_msgs/Float32 "data: -0.5"
```

The second way of controlling the car was through the teleop_twist_keyboard package that allows for the control of the car using keyboard keys and a terminal window, as if it was a video game. This screen can be seen in Figure 4.14. [46]

Running the Twist Teleop shows an interface through the terminal window that allows for controlling the movement of the car using keys on the computer's keyboard, as well as increasing or decreasing the value of the topic published.

The main problem with this was that, even though it allowed for easy control of the car as if it was a video game, it was sometimes hard to control the precise moment in which the speed was changed. There could also be errors when controlling it due to the fact that hitting the wrong keys is easy and having clear knowledge of the direction of the car when looking at it from a third person point of view is hard.

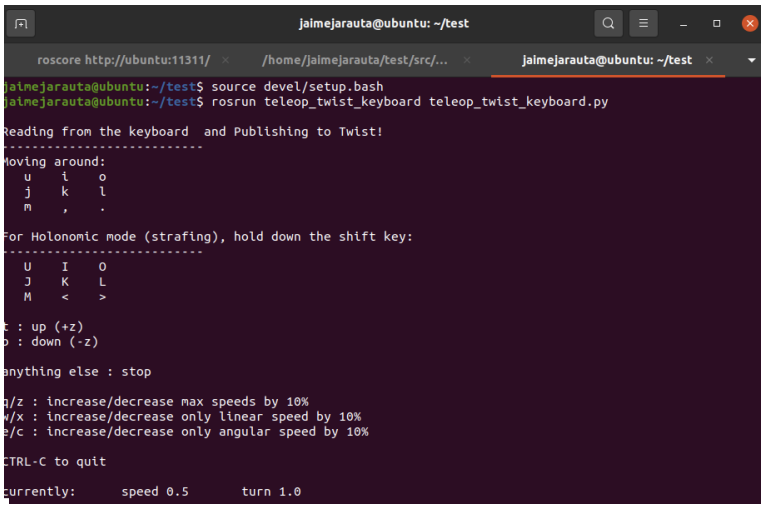

Figure 4.14: Twist Teleop Controller

To run the Twist Teleop option, use the following command in terminal.

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

The last option to control the car was through a GUI that published directly to the speed Topic. This screen can be seen in Figure 4.13

This version of the simulation did not include a hill model either, it was created to check different ways of creating the publisher-subscriber network and to do research on the options available for the final version.

The following are the ROS topics involved in the simulation and movement of this differential drive robot:

- /clock
- /cmd_vel
- /gazebo/link_states
- /gazebo/model_states
- /gazebo/parameter_descriptions
- /gazebo/parameter_updates
- /gazebo/performance_metrics
- /gazebo/set_link_state
- /gazebo/set_model_state
- /joint_states
- /odom
- /rosout
- /rosout_agg
- /tf
- /tf_static

The topics used for publishing the speed are: /cmd_vel and /odom

4.6 Final Version

The final version of the car included all of the previous lessons learned. A new car model explained in section 6.1 was declared and inserted into the Gazebo world with the newest version of the hill model. A publisher-subscriber network was used for changing the values for the speed and a new python script which can be seen in subsection A.3.3 was developed in order to control the speed values and direction depending on the movement of the car. [47] This allowed for the automation of the whole environment

As can be seen in Figure 4.15, there is a car model with four wheels, a chassis and all its correspondent links and joints, with the hill as the mountainous profile shown in Figure 4.5.

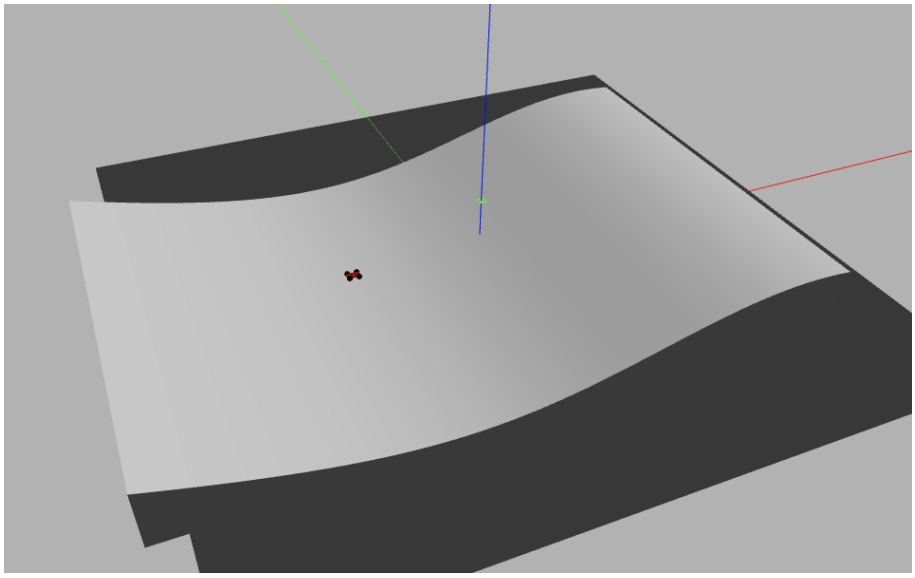

Figure 4.15: World Overview with Small Hill

This version allows for applying the controls necessary to run the car simulation with all of the physics involved (gravity, collisions, friction), and apply the different machine learning policies according to the movement of the car. The results of the simulation are explained and analyzed in chapter 9.

The car is initialised at the bottom section of the hill and its objective is x=0 (where the blue coordinate line axis is located). Through applying a forwards or backwards force to the car's chassis, the model starts oscillating through the surface. Due to challenges that are discussed in chapter 10, the car is not able to reach the final objective. Regardless, by comparing the policies up to equivalent points in the hill, the analysis and comparison of how each of the situations affect the model can be performed.

Chapter 5

System Architecture

The following section presents the programs required for the functioning of the simulator and details the overall system architecture.

Figure 5.1 presents the system flow of the environment. Firstly, the code is created through Python, C++ and XML files using Visual Studio Code and other build tools such as Catkin or CMake. ROS takes these files and compiles them into a working environment which is later simulated visually by tools such as Gazebo or RViz. Ubuntu is used as the Linux distribution, which is overall contained within the VMWare Virtual Machine.

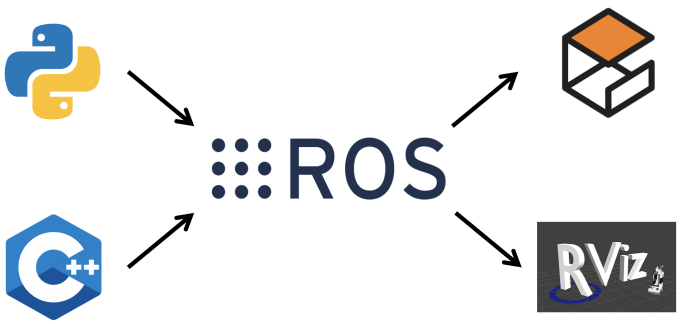

Figure 5.1: System Architecture Overview

All of the tools will be now presented and explained in detail, describing what each one brings to the system.

5.1 Virtual Machine

When it comes to the Virtual Machine used for the simulation, the program "VMWare" was used running an Ubuntu

In order to run ROS and Gazebo, a VM running Ubuntu was created. Initially, a different program (VMBox) was used to install it with approximately 50 GB of storage and 1024 MB of RAM. Although ROS and Gazebo were successfully installed, this setup crashed when some intense tasks were performed and it was not stable under certain conditions.

For this reason, a decision was made to start using VMware, which allowed for a similar install. This alternative proved to be much more stable, allowing for the download of ROS and Gazebo with no further problems.

Figure 5.2: VMWare Logo
Source: [48]

Originally, there was a problem ROS Noetic was not able to be installed. After doing some research, it was found that this happened because that version of ROS did not (at the moment) work with the latest Ubuntu version (22.04 “Jammy Jellyfish”). Therefore a previous version (20.04 “Focal Fossa”) was downloaded and used from the repository of earlier Ubuntu versions.

5.2 ROS

As seen previously, ROS is a framework that allows for writing software related to the modelling and simulation of robots. This section analyzes the main components that create a ROS simulation, what each one does and their importance within the environment. This is fundamental in order to have a complete understanding on how the system works.

As an overview on the ROS architecture, the basic executable process is referred to as a **Node**, and it uses a **Publisher/Subscriber** model for communication. The information exchanged during this communication is called a **Topic**. A Publisher Node can publish one or more Topics, and any Node subscribed to a specific Topic can receive its content. [5] [10] [12] [11]

1. **ROS Nodes**: process that performs computation and is the basic unit of software execution in ROS. Designed to communicate with other nodes over the network (ROS Master). Nodes can publish or subscribe to topics and provide or request services from other nodes. As an example, one node could be in charge of controlling motors whereas other could be responsible for sensor data.

2. **ROS Master**: component that acts as a centralized server. It maintains a registry of all the active nodes, topics, and services, enabling nodes to find and communicate with each other.

3. **ROS Topics**: bus over which nodes exchange messages. Topics allow for a decoupled design where publishers send messages without knowledge of who, if anyone, will receive them, and subscribers receive messages without knowledge of who is publishing, allowing for a many-to-many communication mechanism. Examples of ROS Topics can be a car's odometry or speed. Multiple nodes can both subscribe or publish to a single topic.

4. **ROS Messages**: data structure used for communication between nodes. It defines the type of data that can be sent over a topic (e.g., integers, floats, twist, Odom and others).

5. **ROS Package**: primary organizational unit of ROS code. It contains ROS nodes, libraries, datasets, configuration files, and more. Each package is designed to provide a specific functionality, such as publishing sensor data, performing complex calculations, etc. Packages can depend on other packages.

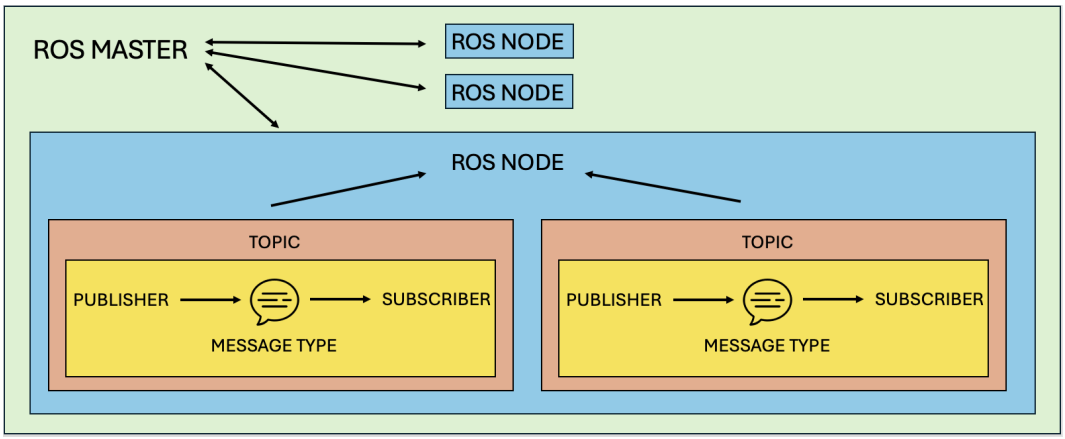

Figure 5.3: ROS Architecture

5.3 Gazebo

As with the previous section, this part introduces the main Gazebo elements and the importance of each one within the simulated environment. This is fundamental once again to understand the overall functioning of our system. [3]

1. **Models**: physical elements that conform a world's environment. In our simulation, both the hill and car are considered to be models.

   (a) **Links**: They represent the physical components of a robot or any other object within the simulation environment. Each link defines a separate rigid body with associated properties such as mass, inertia, and visual characteristics. Links are essential for building the structural aspects of models in Gazebo, creating a detailed specification of shapes, sizes, and materials.

   (b) **Joints**: Joints in Gazebo connect two link elements, enabling relative motion between them. They define the type of mechanical movement allowed and allow setting physical constraints like speed, torque limits, and axis of rotation. Joints are essential for creating realistic articulations in robotic mechanisms.

   - **Revolution Joints**: allow rotational movement around a single axis.
   - **Prismatic Joints**: allow translational movement along a single axis.
   - **Fixed Joints**: allows no relative motion, effectively welds two links together.
   - **Other Joints**: include ball, continuous or planar and other joints.

   (c) **Plugins**: modular codes that extend the capabilities of Gazebo models and the simulation environment. They can control robot behavior, simulate sensor output, or interact with external software libraries. Written in C++ or Python, these plugins can be attached to any robot model or world element.

2. **World**: The Gazebo World defines the simulation environment which includes lighting, ground, and other objects. It's specified in an SDF (Simulation Description Format) file, which details all physical elements and properties. [49]

3. **Physics Engine**: Gazebo supports multiple physics engines such as ODE, Bullet, Simbody, and DART. These engines compute the physical interactions and dynamics within the simulation environment, including collisions, friction, and joint constraints. Users can choose the engine that best fits their simulation needs in terms of accuracy and performance.

4. **Packages**: contain collections of models, scripts, and other simulation elements. These packages are used to organize and distribute simulation resources efficiently, allowing for easy sharing and reuse of simulation setups. Gazebo packages can include predefined models, environments, or fully configured worlds that are crucial for consistent and rapid deployment of simulation scenarios.

5. **Communication System:** Gazebo uses a publisher-subscriber and service-request communication system to manage the flow of information between the simulation elements and external interfaces like ROS as explained in chapter 7.

Figure 5.4, courtesy of Rivera et al., 2019 [3], presents the schematics of how a Gazebo environment runs and the connections between the items explained previously.

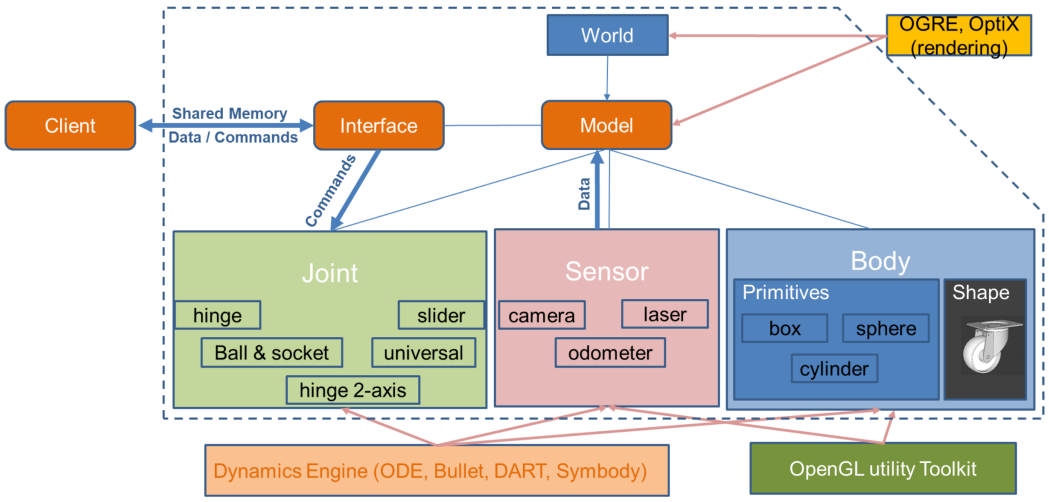

Figure 5.4: Gazebo Architecture Diagram

Source: Courtesy of Rivera et al., 2019 [3]

As a reference, the following table presents the different elements that create both a ROS package and a Gazebo package.

Table 5.1: Comparison of Elements in ROS and Gazebo Packages

Element	**ROS Packages**	**Gazebo Packages**
Package Manifest	`package.xml` contains metadata	`model.config` and `model.sdf` for model definition
Config Files	YAML files for parameter configuration	SDF/URDF files define simulation properties
Executable Files	Nodes and Plugins which can be C++ or Python scripts	Plugins (C++), that interact with the simulation engine
Build System	Uses `catkin` for building packages	Uses `CMake` and optionally `catkin`
Communication	Topics and services for node interaction	Through plugins and SDF files
User Interface	`rqt` and `rviz` for visualization	Gazebo GUI

Chapter 6

Simulation Parameters

This section presents the different components and parameters that declare the car and hill models in the final simulation version, as well as the specifications for the world environment in Gazebo.

6.1 Car

The car model is declared fully through code, creating its links and joints as well as the physics between them. The parameters of the car are presented in the following section:

The position of the car is defined by a three dimensional position (x,y,z) and a three dimensional angular rotation (ψ, ϕ, δ).

$$\vec{p} = (x, y, z, \psi, \phi, \delta)$$

The velocity of the car is declared equivalently, using a vector like the following:

$$\vec{v} = (\dot{x}, \dot{y}, \dot{z}, \dot{\psi}, \dot{\phi}, \dot{\delta})$$

The code where all of the values for the car's models are declared is included in subsection A.4.2.

1. **Links**:

   In this case, there are five links in the car model, composed of the four wheels and the chassis.

   - **Wheels**: four wheels in contact with the ground floor.
     - **Wheel Radius**: set to 0.4 meters.
     - **Wheel Width**: set to 0.2 meters.
     - **Wheel Height**: set to 0.8 meters.
     - **Wheel Mass**: set to 1 kilogram each wheel.

   - **Chassis**:

     Both the visual and collision properties of the chassis are defined with the same box geometry with the chassis' dimensions. This box determines the points of contact of the chassis with other models or elements in the environment. The values set for the chassis' parameters are defined as the following:

     - **Chassis Length**: set to 2 meters.
     - **Chassis Width**: set to 1 meter.
     - **Chassis Height**: set to 0.2 meters.
     - **Chassis Mass**: set to 1 kg

2. **Joints**: There are four joints in the car model, composed of the unions between the axis of the wheels and the chassis. These joints are of the type "continuous", which allow for unlimited rotational motion.

   The definition for the joints is contained in the following section of the code:

```
<joint name="${prefix}_${suffix}_w_j" type="continuous">
<parent link="chassis"/>
<child link="${prefix}_${suffix}_w"/>
<origin xyz="${X} ${Y} ${Z}" rpy="0 0 0"/>
<axis xyz="0 1 0"/>
<limit effort="5" velocity="10000"/>
<dynamics damping="0.0" friction="0.0"/>
</joint>
```

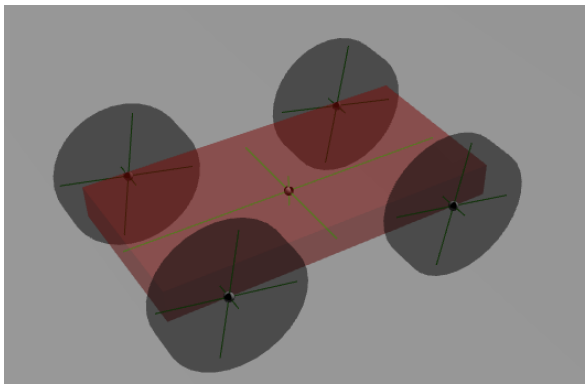

Figure 6.1: Car's Links Center of Mass

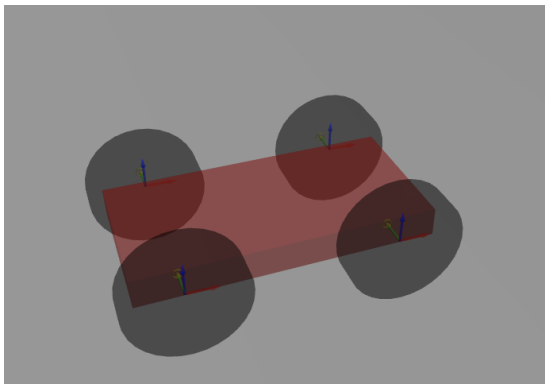

Figure 6.2: Car's Joints

3. **Inertia**:

   - **Cylinder Inertia**: declared within the model to control the response to an external action.
     - **Parameters**: mass (m), radius (r), height (h).

The inertia matrix for a cylinder is given by the following formula:

$$\begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}$$

For a cylinder with mass m, radius r, and height h, the elements of the inertia matrix are:

$$\begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

Where:

$$I_{xx} = I_{yy} = \frac{m(3r^2 + h^2)}{12}$$

$$I_{zz} = \frac{mr^2}{2}$$

The simplified inertia matrix for the cylinder thus becomes:

$$\begin{bmatrix} \frac{m(3r^2+h^2)}{12} & 0 & 0 \\ 0 & \frac{m(3r^2+h^2)}{12} & 0 \\ 0 & 0 & \frac{mr^2}{2} \end{bmatrix}$$

4. **Plugins**:

   There is one plugin declared for the car model named <gazebo_ros_control>. This plugin allows for the implementation of ROS controllers within the Gazebo system, integrating these algorithms into the simulated model. The library used for this plugin is "libgazebo_ros_control.so"

   The plugin handles two-way data from the Gazebo simulation to ROS (like joint states or velocities), and then receiving actuation commands from ROS to be applied within the simulation.

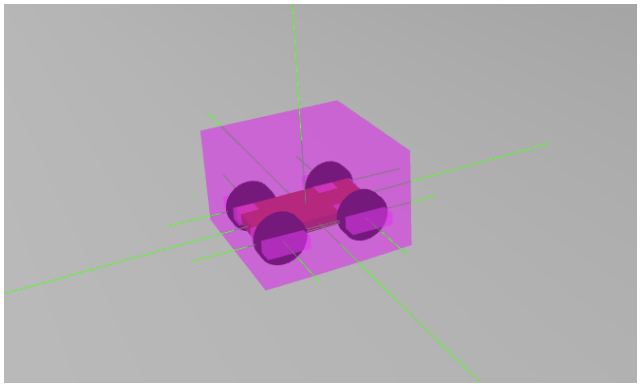

Figure 6.3: Car Model's Inertia

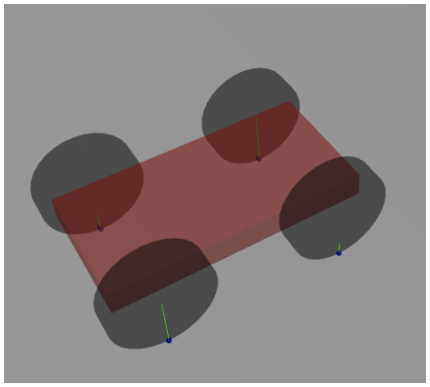

Figure 6.4: Car's Contact Points

5. **Friction Parameters**: (set to very high values in order to prevent slipping)
   - **mu1**: Coefficient of friction in the first direction, set to 1000000.0.
   - **mu2**: Coefficient of friction in the second direction, set to 1000000.0.
   - **kp**: Spring coefficient, set to 1000000.0.
   - **kd**: Damping coefficient, set to 1000000.0.
6. **Materials**: Declared to differentiate the car's wheels and chassis and allow for a clear view of the model within the simulation.
   - **Black**: Color specified as RGBA (0.0, 0.0, 0.0, 1.0) (for chassis)
   - **Red**: Color specified as RGBA (1.0, 0, 0, 1.0) (for wheels)

6.2 Hill

When it comes to the hill, the formula designating its profile comes from the following:

$$U(x) = 0.45\sin(3x) + 0.55 \quad x \in \mathbb{R}$$

Because the car was unable to climb the hill as steep as the one in the formula, it was decided to scale the hill in the z direction to a quarter of its height for the gazebo simulation, using this new formula instead.

$$U(x) = 0.1125\sin(3x) + 0.55 \quad x \in \mathbb{R}$$

For the full explanation on the development and design decisions of the hill model, refer to chapter 4. For the code where the model is declared, refer to subsection A.4.3.

The rest of the main properties of the hill are presented in this next section:

1. **Visual and Collision Properties**: all declared within the same box size, shape and position.
   - **Pose**: -22 0 0 0 -0 0.
   - **Geometry**:
     - **Mesh URI**: model://hill/meshes/hill.dae.
     - **Scale**: 1 1 1.
   - **Material and Texture**:
     - **Material**: model://hill/materials/scripts (pathname to file)
     - **Texture**: model://hill/materials/textures. (pathname to file)
2. **Friction** (set to very high values in order to prevent slipping):
   - **mu**: 1000.
   - **mu2**: 1000.
   - **Slip1**: 1000.
   - **Slip2**: 1000.
3. **Other Properties**:
   - **Maximum Contacts**: 10.
   - **Self Collide**: 0 (prevented)
   - **Static**: 1. (model does not move)

6.3 World

The environment in which the simulation runs, has certain parameters that need to be declared. The following are shown in this section. For the code where these parameters are declared, refer to subsection A.4.3. There are some values that have been omitted due to their relevance within the simulation.

1. **Ground Plane**:
    - **Static**: 1 (The ground plane is static).
    - **Collision Properties**:
        - **Size**: 100 100.
        - **Maximum Contacts**: 10.
    - **Surface Properties**:
        - **Friction**:
            - **mu**: 100 (Coefficient of friction in the first direction)
            - **mu2**: 100 (Coefficient of friction in the second direction,)
        - **Bounce**: Default values.
    - **Visual Properties**:
        - **Cast Shadows**: 0.
        - **Material**:
            - **Script URI**: file://media/materials/scripts/gazebo.material (pathname to file)
            - **Color**: Gazebo/Grey.
    - **Other Properties**:
        - **Self Collide**: 0.
        - **Enable Wind**: 0.
2. **Light (Sun)**:
    - **Type**: Directional.
    - **Pose**: 0 0 10 0 -0 0.
    - **Diffusion**: 0.8 0.8 0.8 1.
    - **Attenuation Range**: 1000
    - **Direction**: -0.5 0.1 -0.9.
3. **Physics**:
    - **Physics Engine Used**: ODE (Open Dynamics Engine).
    - **Max Step Size**: 0.001.

- **Real Time Factor**: 1.
- **Real Time Update Rate**: 1000.

4. **Scene**:
   - **Ambient Light**: 0.4 0.4 0.4 1.
   - **Background Color**: 0.7 0.7 0.7 1.
   - **Shadows**: 1.
5. **World Parameters**:
   - **Gravity**: 0 0 -9.81.
   - **Magnetic Field**: 6e-06 2.3e-05 -4.2e-05. (ignored in our simulation)
   - **Atmosphere**: Adiabatic.

Chapter 7

Publisher Subscriber Network

In robotic simulators, a reliable communication between the components is fundamental for operations. ROS uses a publisher-subscriber communication model to create such interactions. This model is especially effective in managing data flow between different nodes within an environment.

This section explores the fundamental components of the publisher-subscriber network in ROS/Gazebo simulations. This system is essential for creating the car's movement within the intended Machine Learning policies being tested. Each part details how these elements contribute to the functionality of robotic simulations, supporting features such as real-time control or state management.

7.1 Nodes

A node in ROS is an executable that uses ROS to communicate with other nodes as explained in chapter 5. Nodes can publish or subscribe to topics, provide or use services, or read and write parameters. In this simulation, the nodes are responsible for controlling the speed and direction of the car in the Gazebo environment.

The following are the ROS Nodes declared for our simulation. All of these are contained within the "Car Launch" file in subsection A.1.1.

7.1.1 Nodes Used

Robot State Publisher Node:

This node is responsible for publishing the state of the robot, including its joint positions and transformations between various links. It helps other nodes to understand the robot's configuration in real-time

```
<node name="robot_state_publisher" pkg="robot_state_publisher" type="
    robot_state_publisher"
    respawn="false" output="screen">
</node>
```

Controller Spawner Node:

This node spawns and manages the controllers for the car, such as the joint state controller and differential drive controller. It ensures the proper controllers are loaded and running, allowing for the control of the car's movement and behavior.

```
<node name="robot_arm_controller_spawner" pkg="controller_manager" type="
    spawner"
  respawn="true" output="screen"
  args="/car/joint_state_controller
        /car/diff_drive_controller
        --shutdown-timeout 3"/>
```

RQT Robot Steering Node:

This node provides a graphical user interface for steering the robot. It allows users to send velocity commands to the car's differential drive controller via the specified topic.

```
<node name="rqt_robot_steering" pkg="rqt_robot_steering" type="
    rqt_robot_steering">
  <param name="default_topic" value="/car/diff_drive_controller/cmd_vel"/>
</node>
```

Lastly, there is an RViz node, whose use is optional in the case that this tool is used for debugging purposes which is declared in the following section:

RViz Node:

This optional node is used for visualization and debugging purposes. RViz provides a 3D visualization of the robot and its environment, helping to debug and understand the robot's behavior within the simulation.

```
<node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)"/>
```

7.1.2 Node Declaration and Elements

Now, the analysis will focus on how these nodes are managed while the simulation is running. In order to move the car using the python scripts that declare the different ML policies, the user has to publish a value to the node. This part will now analyse the different components of a node, what each one does within the publisher-subscriber network and how the script was used.

- **Node Initialization**: The ROS node is initialized with the name `speed_controller` using `rospy.init_node("speed_controller")`.

```
# Initialize the ROS node
rospy.init_node("speed_controller")
```

- **Subscribers**: The node subscribes to the `/gazebo/model_states` topic to receive updates on the car's state in the simulation. The `model_ states_callback` function is defined to handle incoming messages, updating global variables within the script for the car's position (`x`), speed (`gazebo_speed`), and orientation (`current_yaw`).

```
# Subscribe to the Gazebo model states topic
rospy.Subscriber("/gazebo/model_states", ModelStates,
    model_states_callback)
```

- **Publishers**: The node publishes velocity commands to the `/car/diff_drive_controller/cmd_vel` topic. The `pub` object of type `rospy.Publisher` is used for this purpose.

```
# Publisher for commanding the car's velocity
pub = rospy.Publisher("/car/diff_drive_controller/cmd_vel", Twist,
    queue_size=1)
```

7.2 Topics

As explained briefly in chapter 5 in ROS, topics are named buses over which nodes exchange messages. They facilitate the communication between nodes. This subsection presents the specific topics used within our simulation.

The main topics used for controlling the car's movement are the following. These are used within the python ML scripts in subsection A.3.1.

- `/gazebo/model_states`: This topic publishes the states of all models in the Gazebo simulation, including their positions, orientations, and velocities.

```
# Subscribe to the Gazebo model states topic
rospy.Subscriber("/gazebo/model_states", ModelStates,
    model_states_callback)
```

- `/gazebo/set_model_state`: This topic allows the user to set the state of a model in the simulation.

- `/car/diff_drive_controller/cmd_vel`: This topic is used to change the linear and angular velocity of the car. It expects messages of type `Twist`.

```
# Publisher for commanding the car's velocity
pub = rospy.Publisher("/car/diff_drive_controller/cmd_vel", Twist,
    queue_size=1)
```

- `/car/diff_drive_controller/odom`: This topic publishes odometry information, which includes data about the position and orientation of the car as well as its velocity (both linear and angular).

7.3 Messages

Messages in ROS are data structures used to communicate data between nodes. The following are some of the message types used within our simulation.

- **Point Message**: represents the x, y, z position of an object.
- **Pose Message**: represents the position and orientation of a robot or object. It includes two main fields:
  - `position`: `geometry_msgs/Point` type, specifying the position in x, y, and z coordinates.
  - `orientation`: `geometry_msgs/Quaternion` type, specifying the orientation as a quaternion.
- **Vector3 Message** The `Vector3` message, used within other messages like `Twist` and `Odom`, represents a vector in free space. It consists of three fields: `x`, `y`, and `z`, which specify the components of the vector in their respective directions.
- **Twist Message**: this message, sent to the `/car/diff_drive_controller/cmd_vel` topic, defines the desired linear and angular velocity for the car. It includes two main fields:
  - `linear`: `geometry_msgs/Vector3` type declaring linear velocity in the x, y, and z directions
  - `angular`: `geometry_msgs/Vector3` type declaring angular velocity around the x, y, and z axis.
- **Odom Message**: this message provides odometry information, including the position and velocity of the robot. It includes the following fields:
  - `pose`: `geometry_msgs/PoseWithCovariance` type, specifying the position and orientation.
  - `twist`: `geometry_msgs/TwistWithCovariance` type, specifying the velocity.
- **JointState Message** The `JointState` message contains information about the state of joints in a robot, including their positions, velocities, and efforts.
  - `name`: joint names.
  - `position`: joint positions.
  - `velocity`: joint velocities.
  - `effort`: joint efforts.

- **LinkStates Message**: this message is typically published to the `/gazebo/link_states` topic and contains information about the states of various links in a simulation environment managed by Gazebo. This message includes several arrays:
  - `name`: names of the links
  - `pose`: positions and orientations of each link
  - `twist`: velocities of each link

- **Clock Message** The `Clock` message, published on the `/clock` topic, is used for synchronizing events in a ROS simulation environment.

7.4 ROS-Gazebo Network Problem

In the context of operating our simulation within the Gazebo environment, it's crucial to understand the dynamics of the Publisher-Subscriber network and its difference from a pure ROS setup instead of a ROS/Gazebo setup.

In an only ROS environment, nodes have the capability to both publish to and subscribe from topics. For instance, publishing a velocity of 20 on the '/cmd_vel' topic would propagate that command to any listening subscribers, effectively setting the speed to 20 within the whole ROS network. However, topics originating from Gazebo are in most cases publish-only due to its role in simulating physical dynamics. Unlike ROS, which manages movement without accounting for physical forces, Gazebo incorporates real-world physics, making direct speed commands non-trivial as they must consider factors like gravity and friction.

Consequently, in our simulation, there can only be published commands directed to a ROS-specific velocity topic rather than to a Gazebo specific velocity topic. This approach is necessary since Gazebo simulates physics, which may lead to scenarios where increasing the velocity value in ROS terms does not correspond to movements in the Gazebo simulation. For example, despite increasing the value for the velocity topic in ROS, a Gazebo simulated car might move backward or stay static due to Gazebo's physical calculations.

ROS, in its pure form, simulates the robot navigating an ideal, obstacle-free plane free of environmental influences like gravity or friction. An understanding into the ROS-only simulated movements can be seen by launching RViz (refer to the 'car.launch' file in subsection A.1.1, where the RViz command is provided but commented out). This is because this tool only shows visually the values for the ROS topics, instead of simulating physics like Gazebo does.

The equivalent to this situation would be, when accelerating with a car, the ROS topic being published to would be the pressure applied to the accelerator, but the Gazebo

speed of the model would be the real speed, and direct publishing to the real speed is not possible. Therefore, in a steep hill, the accelerator may be at its highest value, yet the car may still be moving backwards.

This is the reason why the ROS and Gazebo velocity value are different as can be seen in 7.1 (in this case, the ROS published velocity has a value of -120 whereas the Gazebo real speed shown visually in the simulation has a value of -1.444).

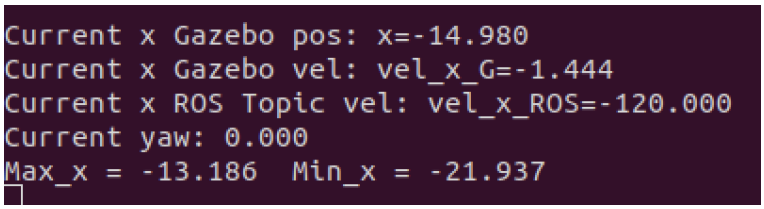

Figure 7.1: Display Info Python Script Terminal View

Chapter 8

Machine Learning Policies

The purpose of the project comes from testing different machine learning policies applied to the car and analyzing the results. The optimal policy will be the one that takes the car from point A to B in the shortest time, therefore, different hill profiles could have different optimal policies. The following mathematical approach comes from the original documents provided for this project.

[39] **Richard Sowers. 2023. Reinforcement Learning Presentation. University of Illinois at Urbana-Champaign.**

This problem resembles a person moving in a swing, in which depending on the position and speed of the swing, the force input has different direction and value with the purpose of increasing or decreasing the energy in the system.

The car has three control settings.

1. **Forward**: unit force is applied to the right (control +1)
2. **Backward**: unit force is applied to the left (control -1)
3. **Neutral**: no force is applied (control 0)

These values were ones for the original problem presented, but, due to the fact that after running the first versions of the simulations, if a constant force was applied to the car, then, there would be a point in which it would not ascend more up the hill due to the friction parameters. For this reason, as can be seen in the python scripts for each policy A.3.3, the absolute value for the force is increased for each iteration.

1. **Forward**: force is applied to the right with a value of abs(current speed) + speed increment
2. **Backward**: force is applied to the left with a value of -(current speed) - speed increment
3. **Neutral**: no force is applied

Optimal Policy Calculation

In order to calculate the optimal policy, the focus will be on the following key mathematical formulations that quantify both the performance and the effectiveness of different policies in reaching a predefined goal set, G. These formulations provide a guide to find out which policy gets the car to the goal in the shortest time.

1. **Definition of the Goal Set [G]**: The goal G is defined as $(x_+, \infty) \times \mathbb{R}$, where x_+ represents a threshold value beyond which the goal is considered to be achieved.
2. **Time to Reach G under Policy π ($T^\pi(z)$)**: The function $T^\pi(z)$ calculates the minimum number of steps (or time units) required to reach the goal set G from a starting state z under a policy π. Formally, it is defined as:

$$T^\pi(z) = \min\{n \geq 0 : z_n(z) \in G\}$$

where $z_n(z)$ denotes the state of the system at step n when starting from z and following the policy π. If no such n exists where $z_n(z)$ is within G, then $T^\pi(z)$ is defined to be infinity (∞), indicating that the goal is unachievable from z under π.

These formulations form the basis for developing a strategy that not only meets the operational requirements but does so in the most efficient manner possible. By determining analytically the optimal policy using these principles, strategies can implemented that are both effective and efficient. While the values for these formulas are not specifically calculated within our simulation, they are inherently present in studying the response of the system.

8.1 Python Script Overview

Even though this script was introduced in chapter 6 by showing how different nodes and topics were used within the simulation, this section will provide a more detailed explanation on what each script does.

These Python scripts serve as a ROS node to control a simulated car's velocity based on its position and orientation, using instantaneous data from the Gazebo simulation. The full script can be seen A.3.1

- **Imports and Global Variables:** The script imports necessary ROS packages and defines global variables for storing the car's position, speed, and orientation.

```
import rospy
from gazebo_msgs.msg import ModelStates
from geometry_msgs.msg import Twist
from tf.transformations import euler_from_quaternion
import display_info
import os

x = 0.0
gazebo_speed = 0.0
initial_heading = None
current_yaw = 0.0
```

- **Callback Function** (`model_states_callback`): Processes data from the `/gazebo/model_states` topic, updating the car's position (`x`), speed (`gazebo_speed`), and orientation (`current_yaw`). The car's initial heading is recorded the first time this callback is invoked.

```
def model_states_callback(msg):
    global x, gazebo_speed, initial_heading, current_yaw
    car_index = 2
    x = msg.pose[car_index].position.x
    gazebo_speed = msg.twist[car_index].linear.x
    orientation_q = msg.pose[car_index].orientation
    _, _, current_yaw = euler_from_quaternion([orientation_q.x,
    orientation_q.y, orientation_q.z, orientation_q.w])
    if initial_heading is None:
        initial_heading = current_yaw
```

- **Node Initialization:** Initializes the ROS node named "speed_controller", allowing the script to communicate with the ROS network.

```
rospy.init_node("speed_controller")
```

- **Subscriber:** the node subscribes to the `/gazebo/model_states` topic to receive updates about the model's state, particularly those of the car, within the Gazebo simulation.

```
rospy.Subscriber("/gazebo/model_states", ModelStates,
    model_states_callback)
```

- **Publisher Creation:** Establishes a publisher to the `/car/diff_drive_controller/cmd_vel` topic, which allows the script to command the car's velocity using a Twist type message.

```
pub = rospy.Publisher("/car/diff_drive_controller/cmd_vel", Twist,
    queue_size=1)
```

- **goTo Function:** Contains the logic to move the car towards a specified goal along the x-axis. It calculates the required velocity based on the car's current position and orientation, adjusting the car's speed and direction to reach the goal. The function makes decisions based on the car's current and previous states, applies corrections for orientation, and publishes velocity commands until the goal is reached.

```
def goTo(goal_x):
    global x, gazebo_speed, initial_heading, current_yaw
    ...
    if abs(goal_x - x) < 1:
        arrived = True
        speed.linear.x = 0
        speed.angular.z = 0
        print("Destination reached!")
    pub.publish(speed)
    rospy.sleep(0.1)
```

- **Main Execution:** The script's execution starts here, invoking the `goTo` function with a goal position.

```
if __name__ == "__main__":
    goTo(0)  # Set the x-coordinate goal here
```

8.2 Policy 1

Considering the profile of the hill, it can be seen that there is a low point. This is the defined position that divides the left and right part of the hill.

Considering this, in the first policy, a forward force is applied if the car is moving forward and if the car is in the left part of the hill, whereas no force is applied if it is either going backwards or at the right part of the hill.

$$\pi(x, v) = \begin{cases} 1 & \text{if } x < 0 \text{ and } v > 0 \\ 0 & \text{else} \end{cases}$$

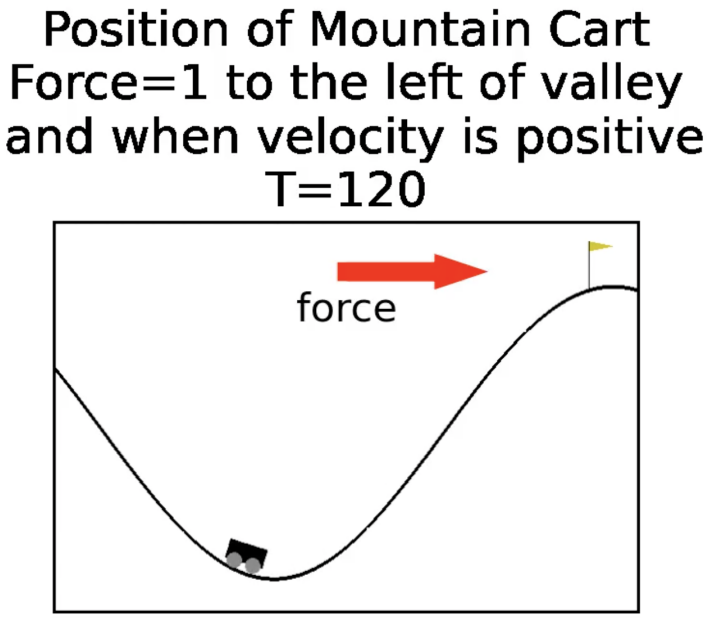

Figure 8.1: Visualization of Force applied for Policy 1

Source: [39]

Figure 8.2 represents the different states of the car depending on its position and speed. As can be seen, the car's objective is to reach position x=0.5 (dashed red line), while the vertical axis represents the instantaneous velocity of the car.

For this first policy, if the car is to the left of the hill (x<0) and with a positive velocity (v>0), then a forward force is applied (represented by the red shaded part of the graph).

The points are plotted on the graph at regular intervals, therefore, the less points on the graph means that the car took less time to reach its objective, showing which would be the optimal policy.

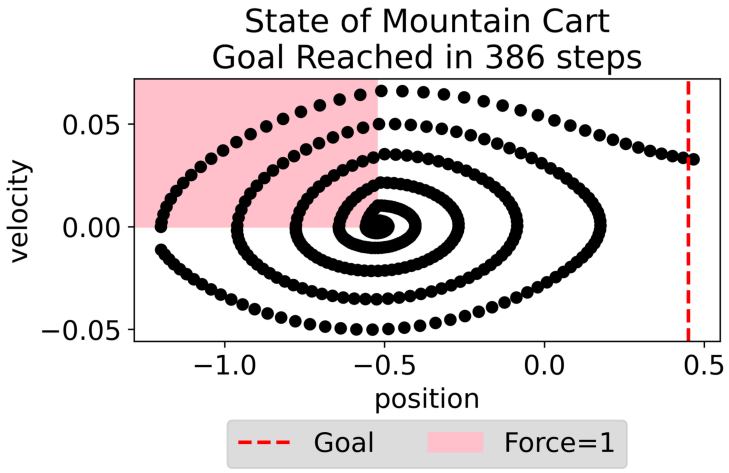

Figure 8.2: Position-Speed Graph for Policy 1

Source: [39]

8.3 Policy 2

In the second policy, a forward force is applied if the car is moving forwards, and no force is applied if it is going backwards, regardless of its position relative to the hill.

$$\pi(x, v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{else} \end{cases}$$

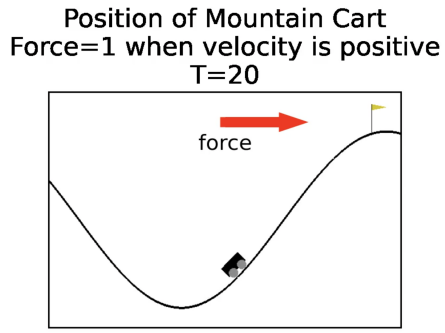

Figure 8.3: Force applied for P2

Source: [39]

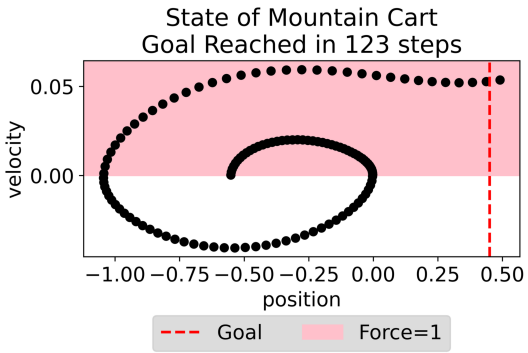

Figure 8.4: Graph for P2

Source: [39]

8.4 Policy 3

In this last policy studied, a forward force is applied if the car is moving forwards and a backward force if it is moving backwards.

$$\pi(x, v) = \begin{cases} 1 & \text{if } v > 0 \\ -1 & \text{if } v < 0 \end{cases}$$

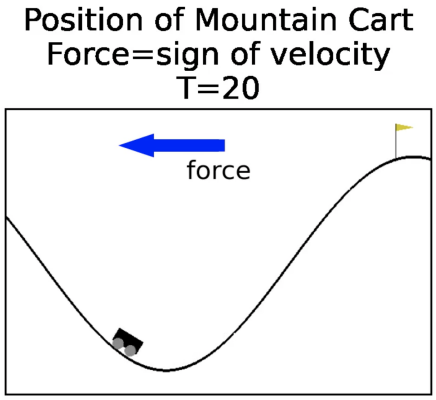

Figure 8.5: Force applied for P3

Source: [39]

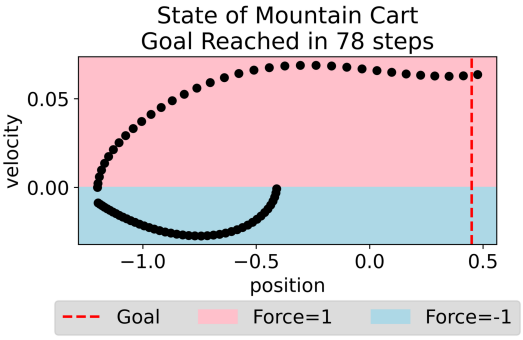

Figure 8.6: Graph for P3

Source: [39]

8.5 Yaw Control

The movement of the car in the simulator is focused on being forward and backwards. For that reason, it is important to make sure that the car stays straight at all times. This is especially difficult especially when the car is on the steep part of the hill and its velocity is close to zero. Even if a very high friction coefficient in the lateral direction is set, the car tends to turn.

For this reason, a control is applied that straightens the trajectory of the car in the case that it steers away from its original orientation. There have been attempts to establish different types of controllers, including a Proportional as well as a "PID" control. This can be seen in the A.3.3 file, in the section that controls both the forward and angular velocity.

Regardless, establishing the correct functioning of this control has been a challenging task and it sometimes does not function correctly. This happens when it does not have the ability to publish to the angular odometry of the car. All of this is explained in chapter 10 in detail.

Chapter 9

Results

The results discussed below are subject to various factors that need consideration. One significant issue is the malfunction of the Yaw controller. This fault affects the vehicle's ability to maintain course, particularly notable when the vehicle encounters large slopes with near-zero velocity. As a result, the vehicle fails to reach the intended final destination of $x = 0$. Therefore, the times and iterations reported here for the three policies only relate within the specified x range.

It is important to consider that changing the hill profile used will create different optimal policies. Regardless, it is obvious that the optimality of these policies will go generally in this order due to the fact that the higher the policy, the more time it spends applying a velocity to the model in the direction of travel.

$Policy\ 1 < Policy\ 2 < Policy\ 3$

As a reminder, these are the forces applied for each situation:

- **Policy 1:** only apply a positive force when the car is moving forwards and to the right of the hill
- **Policy 2:** apply positive force when the car is moving forwards regardless of position
- **Policy 3:** positive force when the car is moving forwards and vice-versa

The following part delves into the specific times and positions obtained when simulating the different protocols.

Small Hill Analysis

Table 9.1: Results for Small Hill Analysis

	Policy 1	**Policy 2**	**Policy 3**
Time	1m 38s	1m 18s	58s
Max Position	-12.064	-12.52	-11.94
Min Position	-23.448	-23.286	-23.06
Iterations	16	12	3

Even though there are some cases in which the car reaches higher positions in the hill, the comparison is based on as similar situations as possible, therefore the results are counting the time it takes for the car to reach equivalent positions in the three cases.

Looking at these results, it can be seen that the worst policy is the first one (if time is the parameter being optimised), and the best one is the third one.

It can also be seen that the number of iterations performed by the car when applying the first policy are significantly higher than the ones for the optimal case.

There have been simulations performed with the bigger model (without scaling in the z axis), but, due to the car not being able to reach far up the hill, those results are not relevant for our research.

9.1 Videos

As part of the results, videos of each policy were created which show the moments in which a force was applied. They also include the terminal window in which through the A.3.4 script, the ROS and Gazebo speeds can be seen as well as other significant parameters. The following figure presents a representation of the videos.

9.2 Discussion

The results show that there is a clear optimal policy for the hill profile studied, and that the different protocols applied show changes in both the time it takes for the car to reach its goal, and the iterations. These results highlight as well the impact of the Yaw controller malfunction on the vehicle's performance, particularly in its ability to maintain a steady course on steep slopes at low velocities. This malfunction has a critical impact, preventing the vehicle from reaching the intended final destination of x = 0.

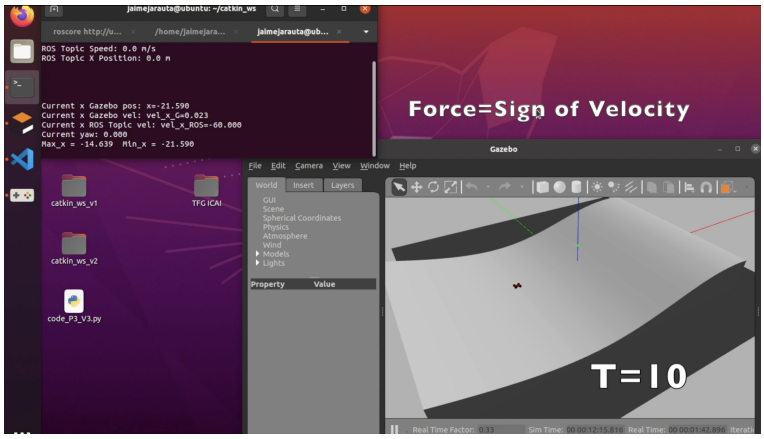

Figure 9.1: Frame of Video Created for Policy 3

Consequently, the recorded times and iterations for the three policies are constrained to a specific x range, affecting the overall assessment of their performance. The changing hill profile further complicates the evaluation as different profiles yield varying optimal policies. However, a clear trend is observed where the optimality of the policies improves with the amount of time the policy spends applying velocity in the direction of travel, resulting in the ranking: Policy 1 $<$ Policy 2 $<$ Policy 3.

In the Small Hill Analysis, the results illustrate that Policy 3, which applies a positive force when the car is moving forwards and viceversa, is the most efficient in terms of time taken, requiring only 58 seconds and 3 iterations. In contrast, Policy 1, which applies force only when the car is moving forwards and to the right of the hill, takes the longest time at 1 minute and 38 seconds and requires 16 iterations.

This significant difference in performance shows the effectiveness of a continuous and correctly directed force application in optimizing travel time. Despite the changes in maximum and minimum positions, the policies are compared under similar conditions, reinforcing the validity of the model through the observed simulation times.

Thes times presented are related to the simulated time, and not real time due to the fact that there is a Time Factor Scaling. This factor comes from the computer running the simulation performing slower than real time. As an example, the videos created are sped up to show a more realistic view on what the car is doing.

Overall, the results show a clear pattern on which policy performs the best under the conditions proposed. The focus of this project was to develop a fully code-based system that allows for the testing of different policies, and, even though there are problems that can be fixed in order to improve the validity of the simulation, the objectives were met.

Chapter 10

Challenges and Future Work

This section outlines the primary challenges encountered throughout the project as well as the work that could be included to the project in the future.

10.1 Challenges

The main challenges within this project have been the following. Most of them have been solved but they are presented here in order to recount how the development of the system was carried out:

- **Understand the ROS/Gazebo environment:** includes the process of learning how to create the simulation using code, learn how to introduce models, understand the file structure of ROS and the different build systems necessary and the interaction between files.
- **Learn the required programming languages:** while there was previous knowledge about python, and some of the tools such as Git or Fusion 360, there have been new programming languages such as C++ or XML that needed to be learnt before actually being able to create the environment.
- **Understand the use and functioning of the files needed for ROS:** these include .world, .launch or .xacro files and the process of investigating the purpose of each one within the simulation.
- **Create the publisher-subscriber network:** probably one of the toughest challenges. Creating the network that moves the car was a hard and iterative process with the help of external forces which ultimately led to being able to apply the ML policies.

- **Yaw Control**: the biggest unresolved problem was creating a system that does not allow the car to turn. An extensive amount of time was spent for this problem and the following are some of the solutions that have been tried and did not work:
  - Edit Friction Coefficients (high, low, medium)
  - Introduce "P" and "PID" controls
  - Speed Controls
  - Force Control
  - No Control
  - Apply linear force more gradually
  - Change mass of elements
  - Change interaction between elements

10.2 Future Work

The following are some ideas for future work in the case that this project keeps being developed:

- Fix Yaw Control
- Add varying friction throughout the hill
- Add varying friction for each wheel
- Introduce random gusts of wind

Chapter 11

How to run the simulation

The following section gives an overview on how to run the latest version of the simulation developed. For running previous versions refer to the README file in the Git repository. To access this, refer to section E.2.

11.1 Running the simulation

To run the ROS/Gazebo simulation for the `car_test` package, follow these steps in a Linux terminal. This procedure assumes there is a ROS environment set up with the `car_test` package located in a Catkin workspace (`~/catkin_ws`) in the home folder of the Linux OS. Refer to Appendix B if ROS or Gazebo is not installed in the computer.

These steps will launch the simulation using a launch file, and then run the Python scripts that interact with the simulation.

These commands are adapted for the latest version of the simulation, but if there are any different Catkin workspaces or packages, follow the same steps editing only the directory such as `catkin_ws` or `car_test` in the case of the package.

1. Start a new terminal session and navigate the Catkin workspace:

   ```
   cd ~/catkin_ws
   ```

   This command changes the directory to the Catkin workspace, where the ROS packages are located. Regardless, if the location or name of the catkin package is different, then the command should be:

```
cd [path to directory]
```

2. In the same terminal, initialize ROS by starting `roscore`:

   ```
   roscore
   ```

   This command starts the ROS master along with the ROS Parameter Server and rosout logging node, which are essential for nodes to communicate.

3. Open a new terminal tab or window (since `roscore` will continue running) and source the ROS environment. Make sure that the terminal is located at the same directory as set in the previous step.

   ```
   source devel/setup.bash
   ```

   This command configures the shell to include ROS environment variables and paths, making sure the ROS packages and nodes can be found and executed.

4. Launch the Gazebo simulation with the .launch file:

   ```
   roslaunch car_test car.launch
   ```

   This command uses `roslaunch` to start the simulation defined in `car.launch`, which initializes Gazebo, loads models, and sets up the simulation environment. After running this command, the Gazebo environment should begin starting up.

5. Once Gazebo is running, in a new terminal tab, source the ROS environment again and run the first Python script desired:

   ```
   source devel/setup.bash
   rosrun car_test code_P3.py
   ```

   Here, `rosrun` is used to execute `code_P3.py`, a Python script within the `car_test` package. This script interacts with the simulation by controlling the car's movement and processing simulation data.

6. The latest version of the python scripts show the current state of the car in the terminal windows. Optionally, if there is additional information that wants to be studied, a different python script named "Display Info" (subsection A.3.4) can be run. To do this, in a new terminal tab or window, source the ROS environment once more and run the script:

```
source devel/setup.bash
rosrun car_test display_info.py
```

Similar to the previous step, this uses `rosrun` to execute `display_info.py`, which displays simulation state information and other relevant outputs.

Make sure to keep `roscore` running throughout this process. Each step after starting `roscore` should be performed in a new terminal window or tab and sourcing the environment, ensuring that all components communicate properly within the ROS framework.

Bibliography

[1] Edgardo J. Roldán-Villasana. "Importance of Simulators, Systematic Approach to Training, and Integral Instruction Centres in the Process Industry". In: *2015 IEEE European Modelling Symposium (EMS)*. 2015, pp. 157–162. DOI: 10.1109/EMS.2015.33.

[2] Johannes Colditz et al. "Use of Driving Simulators within Car Development". In: *DaimlerChrysler AG* (2007).

[3] Zandra B. Rivera, Marco C. De Simone, and Domenico Guida. "Unmanned ground vehicle modelling in Gazebo/ROS-based environments". In: *Machines* 7 (2 2019). ISSN: 20751702. DOI: 10.3390/machines7020042.

[4] Hossein Nick Zinat Matin and Richard B. Sowers. "Nonlinear Optimal Velocity Car Following Dynamics (I): Approximation in Presence of Deterministic and Stochastic Perturbations". In: vol. 2020-July. 2020. DOI: 10.23919/ACC45564.2020.9147363.

[5] Kenta Takaya et al. "Simulation environment for mobile robots testing using ROS and Gazebo". In: 2016. DOI: 10.1109/ICSTCC.2016.7790647.

[6] Evan Ackerman and Erico Guizzo. "Wizards of ROS: Willow Garage and the Making of the Robot Operating System". In: *IEEE Spectrum Automaton* (2017).

[7] Keenan Wyrobek. "The Origin Story of ROS, the Linux of Robotics - IEEE Spectrum". In: *IEEE Spectrum Automaton* (2017).

[8] Vanessa Mazzari. *ROS vs. ROS2*. https://www.generationrobots.com/blog/en/ros-vs-ros2/. Accessed: June 23, 2024. Dec. 2019.

[9] Dirk Thomas. *ROS 2 Design Changes*. http://design.ros2.org/articles/changes.html. Last Modified: 2017-06, Accessed: June 23, 2024. Sept. 2015.

[10] Lentin Joseph and Jonathan Cacace. *Mastering ROS for robotics programming : design, build, and simulate complex robots using the Robot Operating System*. eng. Second edition. Birmingham, [England] ; Packt, 2018. ISBN: 1-78847-452-X.

[11] Carol Fairchild and Thomas L. Harman. *ROS robotics by example : bring life to your robot using ROS robotic applications*. eng. 1st edition. Community experience distilled. Birmingham: Packt Publishing, 2016. ISBN: 1-78528-670-6.

[12] Kumar Bipin. *Robot Operating System cookbook : over 70 recipes to help you master advanced ROS concepts*. eng. 1st edition. Birmingham, UK: Packt Publishing, 2018. ISBN: 1-78398-745-6.

[13] Open Robotics. *Robot Operating System (ROS) Official Website*. Accessed: June 11, 2024. 2024. URL: https://www.ros.org.

[14] Brian P. Gerkey et al. "Most valuable player: A robot device server for distributed control". In: *IEEE International Conference on Intelligent Robots and Systems* 3 (2001). DOI: 10.1109/IROS.2001.977150.

[15] B Gerkey, R Vaughan, and A Howard. "The player/stage project: Tools for multi-robot and distributed sensor systems". In: 2003.

[16] Nathan Koenig and Andrew Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: vol. 3. 2004. DOI: 10.1109/iros.2004.1389727.

[17] Open Robotics. *Gazebo Simulator Official Website*. Accessed: June 11, 2024. 2024. URL: https://gazebosim.org/home.

[18] Open Robotics. *About Gazebo*. Accessed: 2024-05-28. 2024. URL: https://gazebosim.org/about.

[19] Linux Foundation. *The Linux Foundation: It's Not Just the Linux Operating System*. Accessed: June 19, 2024. 2024. URL: https://www.linuxfoundation.org/blog/blog/the-linux-foundation-its-not-just-the-linux-operating-system.

[20] Canonical Ltd. *About the Ubuntu Project*. https://ubuntu.com/about. Accessed: June 19, 2024. 2024.

[21] Linux Foundation. *Linux: Open Source Operating System*. https://www.linux.org. Accessed: June 19, 2024. 2023.

[22] Canonical Ltd. *Ubuntu: The leading operating system for PCs, IoT devices, servers and the cloud*. https://www.ubuntu.com. Accessed: June 19, 2024. 2023.

[23] Scott Chacon and Ben Straub. *Getting Started: A Short History of Git*. https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git. Accessed: June 19, 2024. 2024.

[24] Git. *Git Control System*. https://git-scm.com. Accessed: June 19, 2024. 2023.

[25] GitLab Inc. *About GitLab*. https://about.gitlab.com. Accessed: June 19, 2024. 2023.

[26] Stack Overflow. *2023 Stack Overflow Developer Survey*. Online Survey. Accessed: June 13, 2024. 2023. URL: https://survey.stackoverflow.co/2023/.

[27] Microsoft Corporation. *Visual Studio Code*. https://code.visualstudio.com. Accessed: June 19, 2024. 2023.

[28] Autodesk. *The Next Generation of Design and Engineering Software Has Arrived*. Autodesk Blog. Accessed: June 28, 2024. 2023. URL: https://www.autodesk.com/products/fusion-360/blog/the-next-generation-of-design-and-engineering-software-has-arrived/.

[29] Blender Foundation. *History of Blender*. Official Blender Website. Accessed: June 28, 2024. 2023. URL: https://www.blender.org/about/history/.

[30] Python Software Foundation. *Python Community Logos*. Accessed: June 11, 2024. 2024. URL: https://www.python.org/community/logos/.

[31] International Organization for Standardization C++. *Current Status of the C++ Standard*. Accessed: June 11, 2024. 2024. URL: https://isocpp.org/std/status.

[32] Kitware, Inc. *CMake*. Accessed: 2024-07-09. 2024. URL: https://cmake.org.

[33] Open Robotics. *Using URDF with ROS*. Gazebo Tutorials. Accessed: June 28, 2024. 2023. URL: https://classic.gazebosim.org/tutorials/?tut=ros_urdf.

[34] Petr Bouchner and Stanislav Novotný. "Car dynamics model - Design for interactive driving simulation use". In: 2011.

[35] Gentiane Venture et al. "Modeling and identification of passenger car dynamics using robotics formalism". In: *IEEE Transactions on Intelligent Transportation Systems* 7 (3 2006). ISSN: 15249050. DOI: 10.1109/TITS.2006.880620.

[36] Roger W. Zeits. "Vehicle Dynamics Model for Simulation Use with Autoware.ai on ROS". Master's thesis. Columbus, Ohio: The Ohio State University, 2023.

[37] Marc Rene Zofka et al. "Pushing ROS towards the Dark Side: A ROS-based Co-Simulation Architecture for Mixed-Reality Test Systems for Autonomous Vehicles". In: vol. 2020-September. 2020. DOI: 10.1109/MFI49285.2020.9235238.

[38] Marc René Zofka et al. "Testing and Validating High Level Components for Automated Driving: Simulation Framework for Traffic Scenarios". In: *2016 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2016, pp. 144–150.

[39] Richard Sowers. *Reinforcement Learning Presentation*. University of Illinois at Urbana-Champaign. 2023.

[40] ROS Wiki. *ROS Tutorials*. http://wiki.ros.org/ROS/Tutorials. Accessed: June 13, 2024. 2024.

[41] Gazebo Simulation Environment. *Gazebo Tutorials*. http://classic.gazebosim.org/tutorials. Accessed: June 13, 2024. 2024.

[42] classic.gazebosim.org. *Gazebo : Tutorial : Applying Force/Torque*. Accessed April 3, 2024. 2024. URL: http://classic.gazebosim.org/tutorials?tut=apply_force_torque#Theinterfaceexplained.

[43] ROS Wiki. *rqt Plugins*. http://wiki.ros.org/rqt/Plugins. Accessed: July 1, 2024. 2024.

[44] ROS Wiki. *ROS rqt Plugins*. http://wiki.ros.org/rqt/Plugins. Accessed: June 13, 2024. 2024.

[45] Nils Rottmann. *ROS Gazebo Tutorial - YouTube*. Apr. 2020. URL: https://youtube.com/playlist?list=PLlcq6PMeufrApLSWZR73ivGDjTxayA_Ss.

[46] Mike Purvis. *teleop_twist_keyboard*. Accessed: July 2, 2024. 2015. URL: http://wiki.ros.org/teleop_twist_keyboard.

[47] WhiteBatCodes. *Anass-ABEA/ROS-Python-Robot-Command*. Sept. 2020. URL: https://github.com/Anass-ABEA/ROS-Python-robot-command.

[48] VMware Inc. *VMware Official Website*. https://www.vmware.com. Accessed: June 13, 2024. 2024.

[49] classic.gazebosim.org. *Gazebo : Tutorial : Physics Parameters*. Accessed: April 3, 2024. 2024. URL: https://classic.gazebosim.org/tutorials?tut=physics_params&cat=physics.

[50] ROS Community. *Noetic Installation on Ubuntu*. Accessed: May 30, 2024. 2024. URL: http://wiki.ros.org/noetic/Installation/Ubuntu.

[51] Open Source Robotics Foundation. *Gazebo Tutorial: Install Gazebo using Ubuntu packages*. https://classic.gazebosim.org/tutorials?tut=install_ubuntu. Accessed: May 31, 2024. 2024.

Appendix A

Code

A.1 Launch Files

A.1.1 car.launch File

```
1
2 <?xml version="1.0" encoding="UTF-8"?>
3 <launch>
4
5 <!-- Define arguments that can be modified at runtime -->
6 <arg name="model" default="$(find car_test)/urdf/car.xacro"/>
7 <arg name="hill" default="$(find car_test)/models/hill"/>
8 <arg name="rvizconfig" default="$(find car_test)/urdf.rviz"/>
9
10 <!-- Include Gazebo simulation environment setup -->
11 <include file="$(find car_test)/launch/gazebo.launch">
12   <arg name="model" value="$(arg model)"/>
13 </include>
14
15 <!-- Commented out RViz node for optional use -->
16 <!-- <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)"/>
     -->
17
18 <!-- Load controller configurations onto the ROS parameter server -->
19 <rosparam file="$(find car_test)/config/joint_states.yaml" command="load"/>
20 <rosparam file="$(find car_test)/config/diff_drive.yaml" command="load"/>
21
22 <!-- Spawn controllers for the robot -->
23 <node name="robot_arm_controller_spawner" pkg="controller_manager" type="
     spawner"
24   respawn="true" output="screen"
25   args="/car/joint_state_controller
26         /car/diff_drive_controller
27         --shutdown-timeout 3"/>
28
```

```
<!-- Publish joint states as TF transforms for visualization and other uses -->
<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"
    respawn="false" output="screen">
</node>

<!-- Provide a GUI for manual steering of the robot -->
<node name="rqt_robot_steering" pkg="rqt_robot_steering" type="rqt_robot_steering">
  <param name="default_topic" value="/car/diff_drive_controller/cmd_vel"/>
</node>

</launch>
```

Listing A.1: car.launch File.

A.1.2 gazebo.launch File

```

<?xml version="1.0" encoding="UTF-8"?>
<launch>

<!-- Configuration of simulation parameters and robot model -->
<arg name="paused" default="false"/>
<arg name="use_sim_time" default="true"/>
<arg name="gui" default="true"/>
<arg name="headless" default="false"/>
<arg name="debug" default="false"/>
<arg name="model" default="$(find car_test)/urdf/car.xacro"/>
<arg name="hill" default="$(find car_test)/models/hill"/>
<arg name="world_file" default="/home/jaimejarauta/catkin_ws/src/car_test/worlds/my_mesh.world"/>
<!-- Edit with path to world file  -->

<!-- Include the Gazebo simulation environment with specific configurations -->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="world_name" value="$(arg world_file)"/>
  <arg name="debug" value="$(arg debug)" />
  <arg name="gui" value="$(arg gui)" />
  <arg name="paused" value="$(arg paused)"/>
  <arg name="use_sim_time" value="$(arg use_sim_time)"/>
  <arg name="headless" value="$(arg headless)"/>
</include>

<!-- Load the URDF description of the robot from the specified model file -->
<param name="robot_description" command="$(find xacro)/xacro --inorder $(arg model)"/>

<!-- Spawn the robot model in Gazebo using the provided parameters -->
```

```
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
  args="-x -15 -y 0 -z 4 -unpause -urdf -model car -param robot_description"/>

</launch>
```

Listing A.2: gazebo.launch File.

A.2 Car Model Files

A.2.1 diff_drive.yaml File

```
# Configuration for the differential drive controller of the robot.
car:
  diff_drive_controller:
    # Specifies the type of the controller used.
    type: "diff_drive_controller/DiffDriveController"
    publish_rate: 50 # Rate at which the controller publishes its messages (in Hz).

    # Names of the joints for the left and right wheels.
    left_wheel: ['l_f_w_j','l_r_w_j'] # left front wheel joint, left rear wheel joint
    right_wheel: ['r_f_w_j','r_r_w_j'] # right front wheel joint, right rear wheel joint

    # Physical properties of the wheel assembly.
    wheel_separation: 1.2
    wheel_radius: 0.4

    # Odometry covariances for the encoder output of the robot.
    pose_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03] # covariance matrix for pose (position and orientation)
    twist_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03] # covariance matrix for twist (velocity and angular velocity)

    # The base frame identifier for the robot, used in transformations and localization.
    base_frame_id: chassis

    # Configurations for linear and angular velocity and acceleration limits.
    linear:
      x:
        has_velocity_limits    : false
        max_velocity           : 1000
        has_acceleration_limits: false
        max_acceleration       : 1000
    angular:
```

```
      z:
        has_velocity_limits     : true
        max_velocity            : 0
        has_acceleration_limits: true
        max_acceleration        : 0
```

Listing A.3: diffdrive.yaml File.

A.2.2 joint_states.yaml File

```

car:
  joint_state_controller:
    type: "joint_state_controller/JointStateController"
    publish_rate: 50
```

Listing A.4: jointstates.yaml File.

A.2.3 model.sdf File

```
<?xml version='1.0'?>
<sdf version='1.6'>
  <!-- Definition of the model named 'hill' -->
  <model name='hill'>
    <!-- Definition of the first link in the model -->
    <link name='link_0'>
      <!-- Visual properties of the link -->
      <visual name='visual'>
        <pose frame=''>0 0 0 0 -0 0</pose> <!-- The orientation and position of the visual component -->
        <geometry>
          <!-- Mesh used for the visual shape -->
          <mesh>
            <uri>model://hill/meshes/hill.dae</uri> <!-- Reference to the mesh file -->
            <scale>1 1 1</scale> <!-- Scale of the mesh -->
          </mesh>
        </geometry>
        <!-- Material properties for rendering -->
        <material>
          <ambient>0.5 0.5 0.5 1</ambient>  <!-- Grey color -->
          <diffuse>0.5 0.5 0.5 1</diffuse>
          <specular>0.1 0.1 0.1 1</specular>
          <emissive>0 0 0 0</emissive>
        </material>
      </visual>
      <!-- Collision properties of the link -->
      <collision name='collision'>
        <laser_retro>0</laser_retro> <!-- Reflectivity for laser sensors -->
```

```
28          <max_contacts>10</max_contacts> <!-- Maximum number of contact
    points to be processed during collision -->
29          <pose frame=''>0 0 0 0 -0 0</pose> <!-- The orientation and
    position of the collision component -->
30          <geometry>
31            <!-- Mesh used for collision detection -->
32            <mesh>
33              <uri>model://hill/meshes/hill.dae</uri>
34              <scale>1 1 1</scale>
35            </mesh>
36          </geometry>
37        </collision>
38      </link>
39      <static>1</static> <!-- Model is static within the simulation
    environment -->
40      <allow_auto_disable>1</allow_auto_disable> <!-- Allows the simulation
    to optimize by disabling inactive elements -->
41    </model>
42  </sdf>
```

Listing A.5: car.xacro File.

A.3 Script Files

A.3.1 Policy 1.py File

```
1
2  #!/usr/bin/env python3
3  import rospy
4  from gazebo_msgs.msg import ModelStates
5  from geometry_msgs.msg import Twist
6  from tf.transformations import euler_from_quaternion
7  import display_info
8  import os
9
10 # Global variables for the car's position, speed, and orientation
11 x = 0.0
12 gazebo_speed = 0.0
13 initial_heading = None
14 current_yaw = 0.0
15
16 def model_states_callback(msg):
17     global x, gazebo_speed, initial_heading, current_yaw
18     # Assuming the car's index is 2 in the ModelStates message
19     car_index = 2
20
21     # Update the real Gazebo x position and speed
22     x = msg.pose[car_index].position.x
23     gazebo_speed = msg.twist[car_index].linear.x
24
25     orientation_q = msg.pose[car_index].orientation
```

```
    _, _, current_yaw = euler_from_quaternion([orientation_q.x,
    orientation_q.y, orientation_q.z, orientation_q.w])

    if initial_heading is None:
        initial_heading = current_yaw

def clear_screen():
    os.system('clear')  # Use 'cls' for Windows

# Initialize the ROS node
rospy.init_node("speed_controller")

# Subscribe to the Gazebo model states topic
rospy.Subscriber("/gazebo/model_states", ModelStates, model_states_callback
    )

# Publisher for commanding the car's velocity
pub = rospy.Publisher("/car/diff_drive_controller/cmd_vel", Twist,
    queue_size=1)

# Function to move the car to the goal along the x-axis
def goTo(goal_x):
    global x, gazebo_speed, initial_heading, current_yaw

    current_speed = 20

    max_x = -10000
    min_x = 10000

    k = 0
    change_k = False

    speed = Twist()
    arrived = False

    while not arrived and not rospy.is_shutdown():

        clear_screen()

        display_info.display_info(0)
        print("\n")
        print(f"Current x Gazebo pos: x={x:.3f}")
        print(f"Current x Gazebo vel: vel_x_G={gazebo_speed:.3f}")
        print(f"Current x ROS Topic vel: vel_x_ROS={speed.linear.x:.3f}")
        print(f"Current yaw: {current_yaw:.3f}")
        print(f"Max_x = {max_x:.3f}  Min_x = {min_x:.3f}")

        if x > max_x:
            max_x = x

        if x < min_x:
            min_x = x

        # Determine current direction based on Gazebo speed
        current_direction = 'forward' if gazebo_speed >= 0 else 'backward'
```

```
78
79          # Check if direction changed
80
81          if x < -17.75:
82              if gazebo_speed >= 0:
83                  if not change_k:  # Check if we haven't already incremented
     k for this entry
84                      k += 1
85                      change_k = True  # Mark that k has been incremented for
     this positive velocity entry
86                  current_speed = k * 20
87              else:
88                  change_k = False  # Reset the flag when car is moving
    backward or not in positive velocity zone
89                  current_speed = 0  # Optional: Depends on whether you want
    to stop the car when it moves backward
90          else:
91              current_speed = 0  # If car is behind x = -21, speed should be
    0
92              change_k = False  # Also reset the change_k since we're not in
    the positive velocity, x > -21 zone
93
94          # Set the calculated speed
95          speed.linear.x = current_speed
96
97          # Angular control to correct the orientation
98          if initial_heading is not None:
99              angular_correction = -3 * (current_yaw - initial_heading)
100             speed.angular.z = angular_correction if abs(angular_correction)
     > 0.1 else 0
101
102         # Check if the goal is reached
103         if abs(goal_x - x) < 1:  # Assuming a small threshold for reaching
    the goal
104             arrived = True
105             speed.linear.x = 0  # Stop the car
106             speed.angular.z = 0  # Stop turning
107             print("Destination reached!")
108
109         # Publish the speed command
110         pub.publish(speed)
111         rospy.sleep(0.1)  # Sleep to allow for message processing
112
113 if __name__ == "__main__":
114     goTo(0)  # Set the x-coordinate goal here
```

Listing A.6: Policy 1 File.

A.3.2 Policy 2.py File

```
1
2 #!/usr/bin/env python3
3 import rospy
```

```
from gazebo_msgs.msg import ModelStates
from geometry_msgs.msg import Twist
from tf.transformations import euler_from_quaternion
import display_info
import os

# Global variables for the car's position, speed, and orientation
x = 0.0
gazebo_speed = 0.0
initial_heading = None
current_yaw = 0.0

def model_states_callback(msg):
    global x, gazebo_speed, initial_heading, current_yaw
    # Assuming the car's index is 2 in the ModelStates message
    car_index = 2

    # Update the real Gazebo x position and speed
    x = msg.pose[car_index].position.x
    gazebo_speed = msg.twist[car_index].linear.x

    orientation_q = msg.pose[car_index].orientation
    _, _, current_yaw = euler_from_quaternion([orientation_q.x,
    orientation_q.y, orientation_q.z, orientation_q.w])

    if initial_heading is None:
        initial_heading = current_yaw

def clear_screen():
    os.system('clear')  # Use 'cls' for Windows

# Initialize the ROS node
rospy.init_node("speed_controller")

# Subscribe to the Gazebo model states topic
rospy.Subscriber("/gazebo/model_states", ModelStates, model_states_callback
    )

# Publisher for commanding the car's velocity
pub = rospy.Publisher("/car/diff_drive_controller/cmd_vel", Twist,
    queue_size=1)

# Function to move the car to the goal along the x-axis
def goTo(goal_x):
    global x, gazebo_speed, initial_heading, current_yaw

    current_speed = 20

    max_x = -10000
    min_x = 10000

    k = 1
    current_speed = 0
    previous_velocity_sign = 0

```

```
    speed = Twist()
    arrived = False

    while not arrived and not rospy.is_shutdown():

        clear_screen()

        display_info.display_info(0)
        print("\n")
        print(f"Current x Gazebo pos: x={x:.3f}")
        print(f"Current x Gazebo vel: vel_x_G={gazebo_speed:.3f}")
        print(f"Current x ROS Topic vel: vel_x_ROS={speed.linear.x:.3f}")
        print(f"Current yaw: {current_yaw:.3f}")
        print(f"Max_x = {max_x:.3f}  Min_x = {min_x:.3f}")

        if x > max_x:
            max_x = x

        if x < min_x:
            min_x = x

        # Determine the sign of the current velocity
        velocity_sign = 1 if gazebo_speed >= 0 else -1

        # Check if the direction has changed based on velocity sign
        if velocity_sign != previous_velocity_sign:
            # If moving forward, calculate the new speed
            if gazebo_speed >= 0:
                current_speed = 20 * k
                k += 1  # Increment k for the next speed increase
            else:
                current_speed = 0  # Set speed to 0 if moving backward
            previous_velocity_sign = velocity_sign  # Update the previous
velocity sign

        speed.linear.x = current_speed

        # Angular control to correct the orientation
        if initial_heading is not None:
            angular_correction = -3 * (current_yaw - initial_heading)
            speed.angular.z = angular_correction if abs(angular_correction)
> 0.1 else 0

        # Check if the goal is reached
        if abs(goal_x - x) < 1:  # Assuming a small threshold for reaching
the goal
            arrived = True
            speed.linear.x = 0  # Stop the car
            speed.angular.z = 0  # Stop turning
            print("Destination reached!")

        # Publish the speed command
        pub.publish(speed)
        rospy.sleep(0.1)  # Sleep to allow for message processing

```

```
if __name__ == "__main__":
    goTo(0)  # Set the x-coordinate goal here
```

Listing A.7: Policy 2 File.

A.3.3 Policy 3.py File

```

#!/usr/bin/env python3
import rospy
from gazebo_msgs.msg import ModelStates
from geometry_msgs.msg import Twist
from tf.transformations import euler_from_quaternion
import display_info
import os

# Global variables for the car's position, speed, and orientation
x = 0.0
gazebo_speed = 0.0
initial_heading = None
current_yaw = 0.0

def model_states_callback(msg):
    # Extracting the car's state from the ModelStates message provided by Gazebo
    global x, gazebo_speed, initial_heading, current_yaw

    # Assuming the car's index is 2 in the ModelStates message
    car_index = 2

    # Update the real Gazebo x position and speed
    x = msg.pose[car_index].position.x
    gazebo_speed = msg.twist[car_index].linear.x

    # Convert quaternion orientation to Euler angles for yaw calculation
    orientation_q = msg.pose[car_index].orientation
    _, _, current_yaw = euler_from_quaternion([orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w])

    # Set the initial heading the first time a message is received
    if initial_heading is None:
        initial_heading = current_yaw

def clear_screen():
    # Set the initial heading the first time a message is received
    os.system('clear')  # Use 'cls' for Windows

# Initialize the ROS node
rospy.init_node("speed_controller")

# Subscribe to the Gazebo model states topic
rospy.Subscriber("/gazebo/model_states", ModelStates, model_states_callback)
```

```
44
45 # Publisher for commanding the car's velocity
46 pub = rospy.Publisher("/car/diff_drive_controller/cmd_vel", Twist,
       queue_size=1)
47
48 # Function to move the car to the goal along the x-axis
49 def goTo(goal_x):
50     global x, gazebo_speed, initial_heading, current_yaw
51
52     prev_direction = None
53     speed_increment = 20
54     current_speed = 20
55     changing_direction = False
56
57     max_x = -10000
58     min_x = 10000
59
60     speed = Twist()
61     arrived = False
62
63     while not arrived and not rospy.is_shutdown():
64
65         clear_screen()
66
67         display_info.display_info(0)
68         print("\n")
69         print(f"Current x Gazebo pos: x={x:.3f}")
70         print(f"Current x Gazebo vel: vel_x_G={gazebo_speed:.3f}")
71         print(f"Current x ROS Topic vel: vel_x_ROS={speed.linear.x:.3f}")
72         print(f"Current yaw: {current_yaw:.3f}")
73         print(f"Max_x = {max_x:.3f}  Min_x = {min_x:.3f}")
74
75         if x > max_x:
76             max_x = x
77
78         if x < min_x:
79             min_x = x
80
81         # Determine current direction based on Gazebo speed
82         current_direction = 'forward' if gazebo_speed >= 0 else 'backward'
83
84         # Check if direction changed
85         if current_direction != prev_direction:
86
87             if prev_direction is not None:  # This ensures we skip the
       initial condition where prev_direction is None
88                 changing_direction = True
89                 current_speed = abs(current_speed) + speed_increment  #
       Increase the speed
90
91                 if current_direction == 'backward':
92                     current_speed = -current_speed  # Change direction
93
94             prev_direction = current_direction  # Update the prev_direction
        for the next iteration
```

```
95
96          if changing_direction:
97              speed.linear.x = current_speed
98              changing_direction = False  # Reset the flag after applying the
        speed change
99          else:
100             # Maintain the current speed and direction if there is no
        change in direction
101             speed.linear.x = current_speed
102
103         # Set speed based on current direction
104         speed.linear.x = current_speed
105
106         # Update prev_direction for the next iteration
107         prev_direction = current_direction
108
109         # Angular control to correct the orientation
110         if initial_heading is not None:
111             angular_correction = -3 * (current_yaw - initial_heading)
112             speed.angular.z = angular_correction if abs(angular_correction)
        > 0.1 else 0
113
114         # Check if the goal is reached
115         if abs(goal_x - x) < 1:  # Assuming a small threshold for reaching
        the goal
116             arrived = True
117             speed.linear.x = 0  # Stop the car
118             speed.angular.z = 0  # Stop turning
119             print("Destination reached!")
120
121         # Publish the speed command
122         pub.publish(speed)
123         rospy.sleep(0.1)  # Sleep to allow for message processing
124
125 if __name__ == "__main__":
126     goTo(0)  # Set the x-coordinate goal here
```

Listing A.8: Policy 3 File.

A.3.4 display_info.py File

```
1
2 #!/usr/bin/env python3
3 import rospy
4 from gazebo_msgs.msg import ModelStates
5 from nav_msgs.msg import Odometry
6
7 # Global variables to store the latest data from Gazebo and ROS for the car
      's speed and position
8 gazebo_speed = 0.0
9 gazebo_x_pos = 0.0
10 ros_topic_speed = 0.0
11 ros_topic_x_pos = 0.0
```

```
12
13 def gazebo_state_callback(msg):
14     global gazebo_speed, gazebo_x_pos
15     # Extract speed and position data for the car from the ModelStates
       message provided by Gazebo
16     car_index = 2  # The index of the car in the ModelStates array
17     gazebo_speed = round(msg.twist[car_index].linear.x, 3)  # Car's speed
       from Gazebo
18     gazebo_x_pos = round(msg.pose[car_index].position.x, 3)  # Car's
       position from Gazebo
19
20 def odom_callback(msg):
21     global ros_topic_speed, ros_topic_x_pos
22     # Extract speed and position data from the Odometry message provided by
        a ROS topic
23     ros_topic_speed = round(msg.twist.twist.linear.x, 3)  # Car's speed
       from ROS topic
24     ros_topic_x_pos = round(msg.pose.pose.position.x, 3)  # Car's position
       from ROS topic
25
26 def display_info(event):
27     # Display the car's speed and position from both Gazebo (physics
       adjusted) and ROS (no physics applied)
28     print(f"Real Gazebo Speed: {gazebo_speed} m/s")
29     print(f"Real Gazebo X Position: {gazebo_x_pos} m")
30     print(f"ROS Topic Speed: {ros_topic_speed} m/s")
31     print(f"ROS Topic X Position: {ros_topic_x_pos} m")
32     print("\n")
33
34 def main():
35     rospy.init_node('info_display')
36
37     # Subscribe to Gazebo's model states and ROS's odometry data
38     rospy.Subscriber("/gazebo/model_states", ModelStates,
       gazebo_state_callback)
39     rospy.Subscriber("/car/diff_drive_controller/odom", Odometry,
       odom_callback)
40
41     # Timer to call display_info every 0.5 seconds to refresh the displayed
        data
42     rospy.Timer(rospy.Duration(0.5), display_info)
43
44     rospy.spin()  # Keep the script running and responding to data
45
46 if __name__ == '__main__':
47     main()
```

Listing A.9: displayinfo.py File.

A.4 Other Files

A.4.1 CMakeLists.txt File

```
1
2 cmake_minimum_required(VERSION 2.8.3)
3 project(car_test)
4
5 ## Compile as C++11, supported in ROS Kinetic and newer
6 # add_compile_options(-std=c++11)
7
8 ## Find catkin macros and libraries
9 ## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
10 ## is used, also find other catkin packages
11 find_package(catkin REQUIRED COMPONENTS
12   roscpp
13   rosmsg
14   rospy
15 )
16
17 ## System dependencies are found with CMake's conventions
18 # find_package(Boost REQUIRED COMPONENTS system)
19
20
21 ## Uncomment this if the package has a setup.py. This macro ensures
22 ## modules and global scripts declared therein get installed
23 ## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
24 # catkin_python_setup()
25
26 ################################################
27 ## Declare ROS messages, services and actions ##
28 ################################################
29
30 ## To declare and build messages, services or actions from within this
31 ## package, follow these steps:
32 ## * Let MSG_DEP_SET be the set of packages whose message types you use in
33 ##   your messages/services/actions (e.g. std_msgs, actionlib_msgs, ...).
34 ## * In the file package.xml:
35 ##   * add a build_depend tag for "message_generation"
36 ##   * add a build_depend and a exec_depend tag for each package in
   MSG_DEP_SET
37 ##   * If MSG_DEP_SET isn't empty the following dependency has been pulled
   in
38 ##     but can be declared for certainty nonetheless:
39 ##     * add a exec_depend tag for "message_runtime"
40 ## * In this file (CMakeLists.txt):
41 ##   * add "message_generation" and every package in MSG_DEP_SET to
42 ##     find_package(catkin REQUIRED COMPONENTS ...)
43 ##   * add "message_runtime" and every package in MSG_DEP_SET to
44 ##     catkin_package(CATKIN_DEPENDS ...)
45 ##   * uncomment the add_*_files sections below as needed
46 ##     and list every .msg/.srv/.action file to be processed
47 ##   * uncomment the generate_messages entry below
```

```
48 ##   * add every package in MSG_DEP_SET to generate_messages(DEPENDENCIES
       ...)
49
50 ## Generate messages in the 'msg' folder
51 # add_message_files(
52 #   FILES
53 #   Message1.msg
54 #   Message2.msg
55 # )
56
57 ## Generate services in the 'srv' folder
58 # add_service_files(
59 #   FILES
60 #   Service1.srv
61 #   Service2.srv
62 # )
63
64 ## Generate actions in the 'action' folder
65 # add_action_files(
66 #   FILES
67 #   Action1.action
68 #   Action2.action
69 # )
70
71 ## Generate added messages and services with any dependencies listed here
72 # generate_messages(
73 #   DEPENDENCIES
74 #   std_msgs  # Or other packages containing msgs
75 # )
76
77 ################################################
78 ## Declare ROS dynamic reconfigure parameters ##
79 ################################################
80
81 ## To declare and build dynamic reconfigure parameters within this
82 ## package, follow these steps:
83 ## * In the file package.xml:
84 ##   * add a build_depend and a exec_depend tag for "dynamic_reconfigure"
85 ## * In this file (CMakeLists.txt):
86 ##   * add "dynamic_reconfigure" to
87 ##     find_package(catkin REQUIRED COMPONENTS ...)
88 ##   * uncomment the "generate_dynamic_reconfigure_options" section below
89 ##     and list every .cfg file to be processed
90
91 ## Generate dynamic reconfigure parameters in the 'cfg' folder
92 # generate_dynamic_reconfigure_options(
93 #   cfg/DynReconf1.cfg
94 #   cfg/DynReconf2.cfg
95 # )
96
97 ##################################################
98 ## catkin specific configuration ##
99 ##################################################
100 ## The catkin_package macro generates cmake config files for your package
101 ## Declare things to be passed to dependent projects
```

```
## INCLUDE_DIRS: uncomment this if your package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES car_test
#  CATKIN_DEPENDS roscpp rosmsg rospy
#  DEPENDS system_lib
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)

## Declare a C++ library
# add_library(${PROJECT_NAME}
#   src/${PROJECT_NAME}/car_test.cpp
# )

## Add cmake target dependencies of the library
## as an example, code may need to be generated before libraries
## either from message generation or dynamic reconfigure
# add_dependencies(${PROJECT_NAME} ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
# add_executable(${PROJECT_NAME}_node src/car_test_node.cpp)

## Rename C++ executable without prefix
## The above recommended prefix causes long target names, the following renames the
## target back to the shorter version for ease of user use
## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg someones_pkg_node"
# set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

```

```
149 ## Specify libraries to link a library or executable target against
150 # target_link_libraries(${PROJECT_NAME}_node
151 #   ${catkin_LIBRARIES}
152 # )
153
154 #############
155 ## Install ##
156 #############
157
158 # all install targets should use catkin DESTINATION variables
159 # See http://ros.org/doc/api/catkin/html/adv_user_guide/variables.html
160
161 ## Mark executable scripts (Python etc.) for installation
162 ## in contrast to setup.py, you can choose the destination
163 # install(PROGRAMS
164 #   scripts/my_python_script
165 #   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
166 # )
167
168 ## Mark executables for installation
169 ## See http://docs.ros.org/melodic/api/catkin/html/howto/format1/
    building_executables.html
170 # install(TARGETS ${PROJECT_NAME}_node
171 #   RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
172 # )
173
174 ## Mark libraries for installation
175 ## See http://docs.ros.org/melodic/api/catkin/html/howto/format1/
    building_libraries.html
176 # install(TARGETS ${PROJECT_NAME}
177 #   ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
178 #   LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
179 #   RUNTIME DESTINATION ${CATKIN_GLOBAL_BIN_DESTINATION}
180 # )
181
182 ## Mark cpp header files for installation
183 # install(DIRECTORY include/${PROJECT_NAME}/
184 #   DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
185 #   FILES_MATCHING PATTERN "*.h"
186 #   PATTERN ".svn" EXCLUDE
187 # )
188
189 ## Mark other files for installation (e.g. launch and bag files, etc.)
190 # install(FILES
191 #   # myfile1
192 #   # myfile2
193 #   DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
194 # )
195
196 #############
197 ## Testing ##
198 #############
199
200 ## Add gtest based cpp test target and link libraries
201 # catkin_add_gtest(${PROJECT_NAME}-test test/test_car_test.cpp)
```

```
# if(TARGET ${PROJECT_NAME}-test)
#   target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})
# endif()

## Add folders to be run by python nosetests
# catkin_add_nosetests(test)
```

Listing A.10: CMakeLists Code File.

A.4.2 car.xacro File

```

<?xml version="1.0"?>
<robot name="car" xmlns:xacro="http://www.ros.org/wiki/xacro">

<!-- ////////////////////////////////////////////////////////////////////-->

<!-- Include materials-->

<material name="Black">
   <color rgba="0.0 0.0 0.0 1.0"/>
</material>

<material name="Red">
   <color rgba="1.0 0 0 1.0"/>
</material>

<!-- ////////////////////////////////////////////////////-->

<!-- constants -->
<xacro:property name="deg_to_rad" value="0.017453"/>
<xacro:property name="PI" value="3.14"/>

<!-- chassis property -->
<xacro:property name="chassis_len" value="2"/>
<xacro:property name="chassis_width" value="1"/>
<xacro:property name="chassis_height" value="0.2"/>

<!-- wheel property -->
<xacro:property name="wheel_radius" value="0.4"/>
<xacro:property name="wheel_width" value="0.2"/>
<xacro:property name="wheel_height" value="0.8"/>
<xacro:property name="wheel_mass" value="1"/>

<!-- ////////////////////////////////////////////////////-->

<!-- Macro for computing cylinder inertia, useful for dynamic simulations
   -->
<xacro:macro name="cylinder_inertia" params ="m r h">
  <inertial>
    <mass value="${m}"/>
    <inertia ixx="${m*(3*r*r+h*h)/12}"  ixy="0.0" ixz="0.0"
             iyy="${m*(3*r*r+h*h)/12}"  iyz= "0.0"
```

```
42              izz="${m*r*r/2}"/>
43    </inertial>
44 </xacro:macro>
45
46 <!-- ////////////////////////////////////////////////////-->
47
48 <!-- Macro to provide a simple inertial matrix for parts -->
49 <xacro:macro name= "inertial_matrix" params="mass">
50   <inertial>
51     <mass value="${mass}"/>
52       <inertia ixx="1.0" ixy="0.0"
53                iyy="1.0" iyz="0.0"
54                izz="1.0" ixz="0.0"/>
55    </inertial>
56  </xacro:macro>
57
58 <!-- ////////////////////////////////////////////////////-->
59
60 <!--transmission block-->
61
62 <xacro:macro name="Transmission_block" params="joint_name">
63    <transmission name="${joint_name}_t">
64     <type>transmission_interface/SimpleTransmission</type>
65      <joint name="${joint_name}">
66        <hardwareInterface>PositionJointInterface</hardwareInterface>
67      </joint>
68     <actuator name="${joint_name}_m">
69       <mechanicalReduction>1</mechanicalReduction>
70      </actuator>
71    </transmission>
72  </xacro:macro>
73
74 <!-- ////////////////////////////////////////////////// -->
75
76 <!-- chassis-->
77
78  <link name="chassis">
79     <visual>
80       <origin rpy="0 0 0" xyz="0 0 0"/>
81       <geometry>
82          <box size="${chassis_len} ${chassis_width} ${chassis_height}"/>
83        </geometry>
84      </visual>
85     <collision>
86       <origin rpy="0 0 0" xyz="0 0 0"/>
87       <geometry>
88           <box size="${chassis_len} ${chassis_width} ${chassis_height}"/>
89       </geometry>
90     </collision>
91    <xacro:inertial_matrix mass="1"/>
92  </link>
93
94  <gazebo reference="chassis">
95     <turnGravityOff>false</turnGravityOff>
96     <material>Gazebo/Red</material>
```

```
97   </gazebo>
98
99 <!-- ////////////////////////////////////////    -->
100
101 <!-- wheels-->
102
103 <xacro:macro name="wheel" params= "prefix suffix X Y Z">
104   <link name= "${prefix}_${suffix}_w">
105     <visual>
106       <origin rpy= "${PI/2} 0 0" xyz= "0 0 0"/>
107       <geometry><cylinder length="${wheel_width}" radius= "${wheel_radius}"/
      ></geometry>
108       <material name= "Black"/>
109     </visual>
110     <collision>
111       <origin rpy= "${PI/2} 0 0" xyz= "0 0 0"/>
112       <geometry><cylinder length="${wheel_width}" radius= "${wheel_radius}"/
      ></geometry>
113     </collision>
114   <xacro:cylinder_inertia m="${wheel_mass}" r="${wheel_radius}" h="${
      wheel_width}"/>
115   </link>
116
117 <!-- friction parameters-->
118 <gazebo reference ="${prefix}_${suffix}_w">
119     <mu1 value="1000000.0"/>
120     <mu2 value="1000000.0"/>
121     <kp value="1000000.0"/>
122     <kd value= "1000000.0"/>
123     <material>Gazebo/Black</material>
124 </gazebo>
125
126 <joint name="${prefix}_${suffix}_w_j" type="continuous">
127   <parent link= "chassis"/>
128   <child link= "${prefix}_${suffix}_w"/>
129   <origin xyz= "${X} ${Y} ${Z}" rpy="0 0 0"/>
130   <axis xyz="0 1 0"/>
131   <limit effort= "5" velocity="10000"/>
132   <dynamics damping="0.0" friction="0.0"/>
133 </joint>
134
135  <transmission name="${prefix}_${suffix}_w_t">
136       <type>transmission_interface/SimpleTransmission</type>
137       <actuator name="${prefix}_${suffix}_w_m">
138         <hardwareInterface>EffortJointInterface</hardwareInterface>
139         <mechanicalReduction>1</mechanicalReduction>
140       </actuator>
141       <joint name="${prefix}_${suffix}_w_j">
142         <hardwareInterface>hardware_interface/VelocityJointInterface</
      hardwareInterface>
143       </joint>
144  </transmission>
145 </xacro:macro>
146
147 <xacro:wheel prefix="r" suffix="f" X="0.7" Y="-0.6" Z="0"/>
```

```
148 <xacro:wheel prefix="r" suffix="r" X="-0.7" Y="-0.6" Z="0"/>
149 <xacro:wheel prefix="l" suffix="f" X="0.7" Y="0.6" Z="0"/>
150 <xacro:wheel prefix="l" suffix="r" X="-0.7" Y="0.6" Z="0"/>
151
152
153 <!-- ////////////////////////////////////////////////////////////// -->
154
155 <!-- ros_control plugin-->
156
157 <gazebo>
158   <plugin name="gazebo_ros_control" filename= "libgazebo_ros_control.so">
159    <robotNamespace></robotNamespace>
160   </plugin>
161 </gazebo>
162
163 </robot>
```

Listing A.11: car.xacro File.

A.4.3 my_mesh.world File

```
1
2 <sdf version='1.7'>
3   <world name='default'>
4
5     <!-- Plugin for the car model, can have parameters for customization --
    >
6     <gazebo>
7       <plugin name="car_color_plugin" filename="libcar_test_plugin.so">
8         <!-- Plugin parameters can go here -->
9       </plugin>
10    </gazebo>
11
12    <!-- Ground plane model, defined as static for stability -->
13    <model name='ground_plane'>
14      <static>1</static>
15      <link name='link'>
16        <!-- Collision definition for the plane -->
17        <collision name='collision'>
18          <geometry>
19            <plane>
20              <normal>0 0 1</normal>
21              <size>100 100</size>
22            </plane>
23          </geometry>
24          <!-- Surface properties for the ground -->
25          <surface>
26            <contact>
27              <collide_bitmask>65535</collide_bitmask>
28              <ode/>
29            </contact>
30            <friction>
31              <ode>
```

```
32                  <mu>100</mu>
33                  <mu2>100</mu2>
34                </ode>
35                <torsional>
36                  <ode/>
37                </torsional>
38              </friction>
39              <bounce/>
40            </surface>
41            <max_contacts>10</max_contacts>
42          </collision>
43
44          <!-- Visual representation of the ground plane -->
45          <visual name='visual'>
46            <cast_shadows>0</cast_shadows>
47            <geometry>
48              <plane>
49                <normal>0 0 1</normal>
50                <size>100 100</size>
51              </plane>
52            </geometry>
53            <material>
54              <script>
55                <uri>file://media/materials/scripts/gazebo.material</uri>
56                <name>Gazebo/Grey</name>
57              </script>
58            </material>
59          </visual>
60          <self_collide>0</self_collide>
61          <enable_wind>0</enable_wind>
62          <kinematic>0</kinematic>
63        </link>
64      </model>
65
66      <!-- Sun light source -->
67      <light name='sun' type='directional'>
68        <cast_shadows>1</cast_shadows>
69        <pose>0 0 10 0 -0 0</pose>
70        <diffuse>0.8 0.8 0.8 1</diffuse>
71        <specular>0.2 0.2 0.2 1</specular>
72        <attenuation>
73          <range>1000</range>
74          <constant>0.9</constant>
75          <linear>0.01</linear>
76          <quadratic>0.001</quadratic>
77        </attenuation>
78        <direction>-0.5 0.1 -0.9</direction>
79        <spot>
80          <inner_angle>0</inner_angle>
81          <outer_angle>0</outer_angle>
82          <falloff>0</falloff>
83        </spot>
84      </light>
85      <gravity>0 0 -9.81</gravity>
86      <magnetic_field>6e-06 2.3e-05 -4.2e-05</magnetic_field>
```

```
87      <atmosphere type='adiabatic'/>
88
89      <!-- Physics engine settings -->
90      <physics type='ode'>
91        <max_step_size>0.001</max_step_size>
92        <real_time_factor>1</real_time_factor>
93        <real_time_update_rate>1000</real_time_update_rate>
94        <ode>
95          <constraints>
96            <contact_max_correcting_vel>100000</contact_max_correcting_vel>
97          </constraints>
98        </ode>
99      </physics>
100
101     <!-- Scene ambient and background settings -->
102     <scene>
103       <ambient>0.4 0.4 0.4 1</ambient>
104       <background>0.7 0.7 0.7 1</background>
105       <shadows>1</shadows>
106     </scene>
107     <wind/>
108     <spherical_coordinates>
109       <surface_model>EARTH_WGS84</surface_model>
110       <latitude_deg>0</latitude_deg>
111       <longitude_deg>0</longitude_deg>
112       <elevation>0</elevation>
113       <heading_deg>0</heading_deg>
114     </spherical_coordinates>
115
116     <!-- Model for a hill, placed in the simulation -->
117     <model name='hill_0'>
118       <link name='link_0'>
119
120         <!-- Visual representation of the hill -->
121         <visual name='visual'>
122           <pose>-22 0 0 0 -0 0</pose>
123           <geometry>
124             <mesh>
125               <uri>model://hill/meshes/hill.dae</uri>
126               <scale>1 1 1</scale>
127             </mesh>
128           </geometry>
129           <material>
130             <script>
131               <uri>model://hill/materials/scripts</uri>
132               <uri>model://hill/materials/textures</uri>
133               <name>hill</name>
134             </script>
135           </material>
136         </visual>
137
138         <!-- Collision properties of the hill -->
139         <collision name='collision'>
140           <laser_retro>0</laser_retro>
141           <max_contacts>10</max_contacts>
```

```
142           <pose>-22 0 0 0 -0 0</pose>
143           <geometry>
144             <mesh>
145               <uri>model://hill/meshes/hill.dae</uri>
146               <scale>1 1 1</scale>
147             </mesh>
148           </geometry>
149           <surface>
150             <contact>
151               <ode/>
152             </contact>
153             <bounce/>
154             <friction>
155               <ode>
156                 <mu>1000</mu>
157                 <mu2>1000</mu2>
158                 <slip1>1000</slip1>
159                 <slip2>1000</slip2>
160               </ode>
161             </friction>
162           </surface>
163         </collision>
164         <self_collide>0</self_collide>
165         <enable_wind>0</enable_wind>
166         <kinematic>0</kinematic>
167       </link>
168
169       <static>1</static>
170       <allow_auto_disable>1</allow_auto_disable>
171       <pose>-22 0 0 0 -0 0</pose>
172     </model>
173
174     <state world_name='default'>
175       <sim_time>693 973000000</sim_time>
176       <real_time>315 602961457</real_time>
177       <wall_time>1699551541 145472743</wall_time>
178       <iterations>301591</iterations>
179       <model name='ground_plane'>
180         <pose>0 0 0 0 -0 0</pose>
181         <scale>1 1 1</scale>
182         <link name='link'>
183           <pose>0 0 0 0 -0 0</pose>
184           <velocity>0 0 0 0 -0 0</velocity>
185           <acceleration>0 0 0 0 -0 0</acceleration>
186           <wrench>0 0 0 0 -0 0</wrench>
187         </link>
188       </model>
189       <model name='hill_0'>
190         <pose>0 0 0 0 -0 0</pose>
191         <scale>1 1 1</scale>
192         <link name='link_0'>
193           <pose>0 0 0 0 -0 0</pose>
194           <velocity>0 0 0 0 -0 0</velocity>
195           <acceleration>0 0 0 0 -0 0</acceleration>
196           <wrench>0 0 0 0 -0 0</wrench>
```

```
197         </link>
198       </model>
199       <light name='sun'>
200         <pose>0 0 10 0 -0 0</pose>
201       </light>
202     </state>
203
204     <!-- Camera settings in the GUI -->
205     <gui fullscreen='0'>
206       <camera name='user_camera'>
207         <pose>-37 0 150 0 1.57 0</pose>
208         <view_controller>orbit</view_controller>
209         <projection_type>perspective</projection_type>
210       </camera>
211     </gui>
212   </world>
213 </sdf>
```

Listing A.12: mymesh.world File.

A.4.4 package.xml File

```
1
2 <?xml version="1.0"?>
3 <package format="2">
4   <name>car_test</name>
5   <version>0.0.0</version>
6   <description>Differential Mountain Car Dynamics</description>
7
8   <!-- One maintainer tag required, multiple allowed, one person per tag -->
9   <!-- Example: -->
10  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
11  <maintainer email="jaime.jarauta@gmail.com">Jaime Jarauta</maintainer>
12
13
14  <!-- One license tag required, multiple allowed, one license per tag -->
15  <!-- Commonly used license strings: -->
16  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
17  <license>TODO</license>
18
19
20  <!-- Url tags are optional, but multiple are allowed, one per tag -->
21  <!-- Optional attribute type can be: website, bugtracker, or repository -->
22  <!-- Example: -->
23  <!-- <url type="website">http://wiki.ros.org/car_test</url> -->
24
25
26  <!-- Author tags are optional, multiple are allowed, one per tag -->
27  <!-- Authors do not have to be maintainers, but could be -->
28  <!-- Example: -->
```

```
29  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->
30  <author email="jaime.jarauta@gmail.com">Jaime Jarauta</author> -->
31
32  <!-- The *depend tags are used to specify dependencies -->
33  <!-- Dependencies can be catkin packages or system dependencies -->
34  <!-- Examples: -->
35  <!-- Use depend as a shortcut for packages that are both build and exec
      dependencies -->
36  <!--   <depend>roscpp</depend> -->
37  <!--   Note that this is equivalent to the following: -->
38  <!--   <build_depend>roscpp</build_depend> -->
39  <!--   <exec_depend>roscpp</exec_depend> -->
40  <!-- Use build_depend for packages you need at compile time: -->
41  <!--   <build_depend>message_generation</build_depend> -->
42  <!-- Use build_export_depend for packages you need in order to build
      against this package: -->
43  <!--   <build_export_depend>message_generation</build_export_depend> -->
44  <!-- Use buildtool_depend for build tool packages: -->
45  <!--   <buildtool_depend>catkin</buildtool_depend> -->
46  <!-- Use exec_depend for packages you need at runtime: -->
47  <!--   <exec_depend>message_runtime</exec_depend> -->
48  <!-- Use test_depend for packages you need only for testing: -->
49  <!--   <test_depend>gtest</test_depend> -->
50  <!-- Use doc_depend for packages you need only for building
      documentation: -->
51  <!--   <doc_depend>doxygen</doc_depend> -->
52  <buildtool_depend>catkin</buildtool_depend>
53  <build_depend>roscpp</build_depend>
54  <build_depend>rosmsg</build_depend>
55  <build_depend>rospy</build_depend>
56  <build_depend>gazebo_ros</build_depend>
57  <build_export_depend>roscpp</build_export_depend>
58  <build_export_depend>rosmsg</build_export_depend>
59  <build_export_depend>rospy</build_export_depend>
60  <build_export_depend>gazebo_ros</build_export_depend>
61  <exec_depend>roscpp</exec_depend>
62  <exec_depend>rosmsg</exec_depend>
63  <exec_depend>rospy</exec_depend>
64  <exec_depend>gazebo_ros</exec_depend>
65
66
67
68  <!-- The export tag contains other, unspecified, tags -->
69  <export>
70    <!-- Other tools can request additional information be placed here -->
71
72  </export>
73 </package>
```

Listing A.13: package.xml File.

A.4.5 urdf.rviz File

```
1
2 # File to open RViz simulator
3 Panels:
4   - Class: rviz/Displays
5     Help Height: 78
6     Name: Displays
7     Property Tree Widget:
8       Expanded:
9         - /Global Options1
10        - /Status1
11      Splitter Ratio: 0.5
12    Tree Height: 549
13  - Class: rviz/Selection
14    Name: Selection
15  - Class: rviz/Tool Properties
16    Expanded:
17      - /2D Pose Estimate1
18      - /2D Nav Goal1
19      - /Publish Point1
20    Name: Tool Properties
21    Splitter Ratio: 0.5886790156364441
22  - Class: rviz/Views
23    Expanded:
24      - /Current View1
25    Name: Views
26    Splitter Ratio: 0.5
27  - Class: rviz/Time
28    Experimental: false
29    Name: Time
30    SyncMode: 0
31    SyncSource: ""
32 Preferences:
33   PromptSaveOnExit: true
34 Toolbars:
35   toolButtonStyle: 2
36 Visualization Manager:
37   Class: ""
38   Displays:
39     - Alpha: 0.5
40       Cell Size: 1
41       Class: rviz/Grid
42       Color: 160; 160; 164
43       Enabled: true
44       Line Style:
45         Line Width: 0.029999999329447746
46         Value: Lines
47       Name: Grid
48       Normal Cell Count: 0
49       Offset:
50         X: 0
51         Y: 0
52         Z: 0
53       Plane: XY
54       Plane Cell Count: 10
55       Reference Frame: <Fixed Frame>
```

```
      Value: true
    - Alpha: 1
      Class: rviz/RobotModel
      Collision Enabled: false
      Enabled: true
      Links:
        All Links Enabled: true
        Expand Joint Details: false
        Expand Link Details: false
        Expand Tree: false
        Link Tree Style: Links in Alphabetic Order
        chassis:
          Alpha: 1
          Show Axes: false
          Show Trail: false
          Value: true
        l_f_w:
          Alpha: 1
          Show Axes: false
          Show Trail: false
          Value: true
        l_r_w:
          Alpha: 1
          Show Axes: false
          Show Trail: false
          Value: true
        r_f_w:
          Alpha: 1
          Show Axes: false
          Show Trail: false
          Value: true
        r_r_w:
          Alpha: 1
          Show Axes: false
          Show Trail: false
          Value: true
      Name: RobotModel
      Robot Description: robot_description
      TF Prefix: ""
      Update Interval: 0
      Value: true
      Visual Enabled: true
  Enabled: true
  Global Options:
    Background Color: 48; 48; 48
    Default Light: true
    Fixed Frame: odom
    Frame Rate: 30
  Name: root
  Tools:
    - Class: rviz/Interact
      Hide Inactive Objects: true
    - Class: rviz/MoveCamera
    - Class: rviz/Select
    - Class: rviz/FocusCamera
```

```
111     - Class: rviz/Measure
112     - Class: rviz/SetInitialPose
113       Theta std deviation: 0.2617993950843811
114       Topic: /initialpose
115       X std deviation: 0.5
116       Y std deviation: 0.5
117     - Class: rviz/SetGoal
118       Topic: /move_base_simple/goal
119     - Class: rviz/PublishPoint
120       Single click: true
121       Topic: /clicked_point
122   Value: true
123   Views:
124     Current:
125       Class: rviz/Orbit
126       Distance: 25.682771682739258
127       Enable Stereo Rendering:
128         Stereo Eye Separation: 0.05999999865889549
129         Stereo Focal Distance: 1
130         Swap Stereo Eyes: false
131         Value: false
132       Focal Point:
133         X: 0
134         Y: 0
135         Z: 0
136       Focal Shape Fixed Size: true
137       Focal Shape Size: 0.05000000074505806
138       Invert Z Axis: false
139       Name: Current View
140       Near Clip Distance: 0.009999999776482582
141       Pitch: 0.785398006439209
142       Target Frame: <Fixed Frame>
143       Value: Orbit (rviz)
144       Yaw: 0.785398006439209
145     Saved: ~
146 Window Geometry:
147   Displays:
148     collapsed: false
149   Height: 846
150   Hide Left Dock: false
151   Hide Right Dock: false
152   QMainWindow State: 000000ff00000000
153   Selection:
154     collapsed: false
155   Time:
156     collapsed: false
157   Tool Properties:
158     collapsed: false
159   Views:
160     collapsed: false
161   Width: 1200
162   X: 67
163   Y: 27
```

Listing A.14: urdf.rviz File.

Appendix B

Installation Guide

The following section presents an overview on how to install the versions of ROS and Gazebo chosen for this simulation. Refer to the official corresponding websites for updates in case that these installation processes have become outdated.

B.1 ROS Installation

For this simulation, ROS 1 "Noetic Ninjemys" was used. The following section provides a guide on how to download it for Ubuntu 20.04 "Focal Fossa".[50]

B.1.1 Updating System Packages

Begin by updating your package list and upgrading the existing packages:

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

B.1.2 Setting up ROS Repositories

Add the ROS repository to your system:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" \
> /etc/apt/sources.list.d/ros-latest.list'
```

Install `curl` to facilitate downloading necessary packages:

```
sudo apt install curl
```

Add the ROS packages:

```
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc
| sudo apt-key add -
```

B.1.3 Installing ROS Noetic

Update the package list again after adding new repositories:

```
sudo apt update
```

Install ROS Noetic full desktop version:

```
sudo apt install ros-noetic-desktop-full
```

B.1.4 Initializing rosdep

Install and initialize `rosdep`:

```
sudo apt install rosdep
sudo rosdep init
rosdep update
```

B.1.5 Environment Setup

Update the bash environment to source ROS setups:

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

Verify the ROS installation:

```
rosversion -d
```

B.1.6 Additional ROS Packages

Install additional ROS packages necessary for various functionalities:

```
sudo apt install rospack-tools
sudo apt install ros-noetic-xacro
sudo apt install ros-noetic-robot-state-publisher
sudo apt install ros-noetic-rqt-robot-steering
sudo apt install ros-noetic-joint-state-controller
sudo apt install ros-noetic-diff-drive-controller
```

B.2 Gazebo Installation

The following is a tutorial on how to install Gazebo for Ubuntu 20.04 "Focal Fossa". This tutorial is based on the official Gazebo installation guide [51]. It is pretty straightforwards as it only requires (for most cases) one line executed through terminal.

```
curl -sSL http://get.gazebosim.org | sh
```

In order to run gazebo, execute this command in terminal:

```
gazebo
```

The only problem when doing this is that it initialises solely Gazebo, and, while it can still simulate and visualize environments, the ROS part of the simulation is not initiated.

Refer to chapter 11 to learn how to initialise the project running both ROS and Gazebo.

Additionally, remember that the files for the models used in the Gazebo worlds should be in the following directory:

```
cd /usr/share/gazebo-11/models
```

Appendix C

Sustainable Goals

When it comes to the Sustainable Development Goals by the United Nations that this Project focuses on, the following ones are being considered.

- **Quality Education**: provided by both Universidad Pontificia Comillas and University of Illinois at Urbana-Champaign, as well as any other professors, mentors, peers and helpers involved in the development of this project.
- **Industry, Innovation and Infrastructure**: with this project being based on research, it aims to advance the knowledge in simulation, robotics, mechanical engineering and transportation.

Appendix D

Abbreviations

- **ML**: Machine Learning.
- **ROS**: Robot Operating System.
- **UIUC**: University of Illinois at Urbana-Champaign.
- **OV**: Optimal Velocity.
- **VM**: Virtual Machine.
- **OS**: Operating System.
- **DDS**: Data Distribution Service.
- **UGVs**: Unmanned ground vehicles.
- **WMR**: Wheeled mobile robots.
- **SUMO**: Simulation of Urban Mobility.
- **GUI**: Graphical User Interface.
- **URDF**: Universal Robotic Description Format.
- **SDF**: Simulation Description Format.
- **yaml**: Yet Another Markup Language.
- **ODE**: Open Dynamics Engine.
- **AGV**: Autonomous Ground Vehicles.
- **IDE**: Integrated Development Environment.

Appendix E

Notes

E.1 Contacts

Author: Jaime Jarauta Gastelu (jaime.jarauta@gmail.com)
Supervisor: Professor Richard B. Sowers (r-sowers@illinois.edu)

Universidad Pontificia Comillas - ICAI: comillas.edu
University of Illinois at Urbana-Champaign: illinois.edu

E.2 Git Repository

Link to Git repository with the code and source control of the project:

https://gitlab.com/jaimejarauta/ros