POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Development of Complex Scenarios and Control Algorithms for Autonomous Driving Functions (ADFs) in a Driving Simulator

Supervisor

Prof. CARLO NOVARA

Co-Supervisor

Eng. MATTIA BOGGIO

Co-Supervisor - Centro Ricerche Fiat

Dr. FABIO TANGO

Candidate

MERT BATMAZ

April 2024

Abstract

Autonomous vehicles and Advanced Driver Assistance Systems (ADAS) applications are emerging fields of research and development that require specialized controllers to ensure safe and efficient functionality. However, testing these controllers in real-world scenarios can be costly and risky, making the use of simulators necessary for their design and validation. This thesis presents a framework for the control and simulation of fundamental ADAS applications for autonomous vehicles, using Simulink as the controller design environment and Carla as the simulation environment. The integration of these tools is first described, followed by the development and testing of two control methods: PID and Nonlinear Model Predictive Control (NMPC). These two methods are applied to both lateral and longitudinal control of the vehicle, using the single-track model for PID tuning and as internal model of the NMPC. To address the limitations of the single-track model, which lacked direct throttle and brake control capabilities, a new dispatching function is derived from the vehicle data collected in the Carla simulation environment. Finally, a comprehensive testing scenario is designed within the Carla simulator, consisting of a curve that requires deceleration and steering and a straight road for acceleration. Simulation results verified the effectiveness of the proposed control strategies.

Keywords: Autonomous Vehicles, ADAS (Advanced Driver Assistance Systems), Control Systems, Simulation, Simulink, Carla Simulator, Vehicle Modeling, Single-Track Model, PID (Proportional, Integral, Derivative) Control, NMPC (Nonlinear Model Predictive Control), Longitudinal Control, Lateral Control, Non-linear System Identification, Mathematical Modeling.

Acknowledgements

"This thesis acknowledgment is a tribute to all the individuals who have made my academic journey worthwhile. First and foremost, I extend my deepest gratitude to Professor Carlo Novara for granting me the opportunity to work on this thesis and generously sharing his vast knowledge and expertise throughout the process. I also wish to express my thanks to the professionals involved in the project, particularly Dr. Fabio Tango from Centro Ricerche Fiat. His participation and valuable input were pivotal to the success of this project. Heartfelt appreciation goes to Eng. Mattia Boggio, whose office door was always open whenever I encountered a challenge or had a question during the project. He consistently steered me in the right direction whenever he believed I needed it.

I cannot overlook the support system that made this journey less daunting; Thanks to my friends who have always been my pillars of strength. My deepest appreciation is for my wife, whose unwavering support and dedication were indispensable. This thesis would not have been possible without her. Finally, I owe a debt of gratitude to my parents for their continuous encouragement throughout my years of study. Their belief in me has been a constant source of strength."

Mert Batmaz

Table of Contents

List of Tables

List of Figures

Outline and Contributions

With the advancement of technology today, there have been innovations and developments in many sectors. The automotive sector is one of the most affected by these advancements. The most significant development in this sector has been adapting advanced safety systems to vehicles and making them fully autonomous in the long run. Autonomous vehicles are vehicles that drive themselves without the need for a driver. The Society of Automotive Engineers (SAE) has published a standard (J3016) for autonomous vehicles, ranging from level 0 to 5. Level "0" refers to simple model vehicles without automation, while level "5" indicates a true driverless vehicle [1]. Currently, level three vehicles, which have environmental sensing capabilities, have been commercialized, and it is anticipated that level five autonomous vehicles will be commercialized in the coming years.

However, there are some barriers to the realization of fully autonomous vehicles. The first of these barriers is people's lack of trust in fully autonomous vehicles. According to a recent study [2] in the United States, half of the adults believe that autonomous vehicles are less reliable than traditional vehicles. In this case, the sensors and microprocessors used in fully autonomous vehicles are of great importance for safety. The difficulty of the autonomous vehicle's driving scenario may increase due to variable weather conditions and changing geographies. With advancing technology, the sophistication of sensors increases each year, and their costs decrease. This situation provides an advantage for the transition to fully autonomous vehicles. Another important issue for safety is test simulators. Although the costs of sensors will decrease, the loss of many sensors during tests can lead to significant costs. Therefore, simulators that can simulate real vehicles one-to-one and can simulate different maps along with different weather conditions are of great importance in terms of cost and safety. In these simulators, we can test our controller in variable weather conditions and test the behavior of the vehicle without any additional cost.

Autonomous vehicles will undoubtedly bring many advantages. The most important advantage is their ability to prevent accidents caused by loss of attention in heavy traffic. Moreover, reducing errors to zero by an autonomous vehicle will minimize fuel consumption, thus achieving significant savings. Additionally, the elimination of driver costs, which are the biggest expense in the transportation sector, will positively affect prices in other sectors.

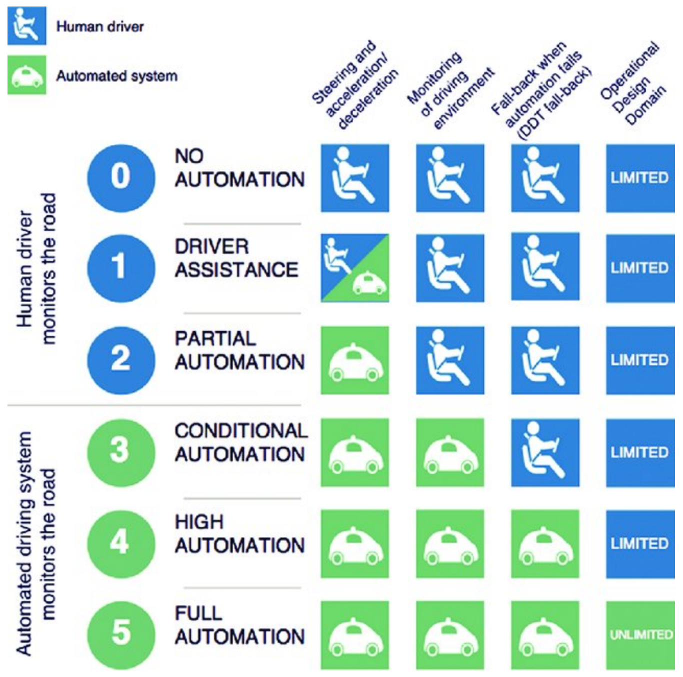
Human driver
Automated system
Steering and acceleration/ deceleration
Monitoring of driving environment
Fall-back when automation fails (DDT fall-back)
Operational Design Domain
Human driver monitors the road
Automated driving system monitors the road
0 NO AUTOMATION
1 DRIVER ASSISTANCE
2 PARTIAL AUTOMATION
3 CONDITIONAL AUTOMATION
4 HIGH AUTOMATION
5 FULL AUTOMATION
LIMITED
UNLIMITED

Figure 1: SAE J3016 levels of driving automation

Driving simulators are of great importance for projects in the field of Advanced Driver Assistance Systems (ADAS). They allow us to see the errors in the early stages of the project and to make developments accordingly. Today, there are many simulator options available. Undoubtedly, the most important feature of a simulator is the ability to test under real conditions.
In this regard, this thesis uses the Carla simulator to create environments closest to real conditions and perform simulations. By doing so, it circumvents one of the most substantial expenses associated with autonomous driving development: the cost of sensors. With Carla's multiple maps, it is possible to create different scenarios. Moreover, the ability to simulate diverse weather conditions and driving scenarios not only enhances the precision of these tests but also significantly reduces the financial burden associated with conducting such evaluations in the initial stages. Additionally, with the Carla-Matlab connection, a model created in Simulink can be simulated for a vehicle created in the Carla environment. This also allows us to develop controllers compared to the current real commercial vehicles. Finally, creating a trajectory through manual driving and comparing these manual driving data with other control methods is beneficial in terms of seeing how successful our

control is. This thesis, therefore, makes a valuable contribution to the advancement of autonomous driving technology.

This thesis focuses on the development of autonomous vehicles and driving simulations within the framework of ADAS applications. Chapter 2 addresses what autonomous driving is and the importance of simulation in this field, then it explains the most important simulators and makes a comparison among existing simulators, and introduces fundamental ADAS applications. Chapter 3 thoroughly explains the processes of selecting the most suitable simulation application and integrating Carla with Anaconda and Matlab interfaces. Chapter 4 describes the development and simulation of different control strategies for path tracking, and vehicle modeling in the Simulink environment, and PID and NMPC control methods are examined in depth. The lateral and longitudinal path-tracking performance of PID and NMPC is evaluated under two different scenarios, Chapter 5 gives the results of simulations and compares them with manual driving data, and the effectiveness of various control methods is questioned. Lastly, Chapter 6 summarizes the results obtained and offers suggestions for future work. This study overall, aims to contribute to the understanding and improvement of methodologies and control strategies developed for autonomous driving simulations.

Chapter 1

Introduction

1.1 What is Autonomous Driving?

The term autonomous vehicle has been used many times throughout history. When cars were first designed, the definition of "automobile," derived from the combination of the Greek word "autòs," meaning "self, individual, independent," and the Latin word "mobilis," meaning "movable," focused on the concept of "moving by itself" [3]. The main idea here was to achieve movement without the effort of the driver, similar to how horses could move on their own. However, this term failed to acknowledge that the absence of horses also meant the loss of a certain type of freedom. Through training and dressage, carriage horses had learned to stay within the limits of simple rules on their own (Greek autos, as mentioned above, means "by itself," nómos: "human order, laws made by humans").
In this sense, the horse and carriage thus achieved a certain autonomy. But this feature of independent movement, which was very important in the transition from horse-drawn carriages to cars, was lost. Initially, even if horses were not suitable for driver control, they could drive the vehicle and safely bring the driver home or wait while grazing somewhere, ensuring the driver's safety. The term autonomous vehicle today aims to regain this lost feature of independent movement and advance it much further. Towards the end of the 20th century, advancements in computer science and artificial intelligence began to make the concept of vehicles sensing their environment and making decisions a possibility. Initial tests in the 1980s and 1990s demonstrated that vehicles could move autonomously under certain conditions. During this time, developments in GPS navigation and sensor technology allowed vehicles to more accurately locate and perceive objects around them. Driverless vehicle technology gained momentum at the beginning of the 21st century. Initiatives like Google's driverless car project [4] captured public attention and created a general awareness that autonomous driving was a real possibility. The use of

cameras, radars, lidars, and artificial intelligence algorithms in vehicles endowed them with the capability to navigate complex traffic situations independently. Many car manufacturers and technology companies are now accelerating their efforts to develop fully autonomous vehicles. The goal of these vehicles is to reduce traffic accidents, increase transportation efficiency, and provide greater independence for people with mobility limitations. However, technical challenges, as well as ethical, legal, and safety issues, continue to shape the advancement in this field. The future of autonomous driving depends on overcoming these challenges and gaining societal acceptance for this new technology.

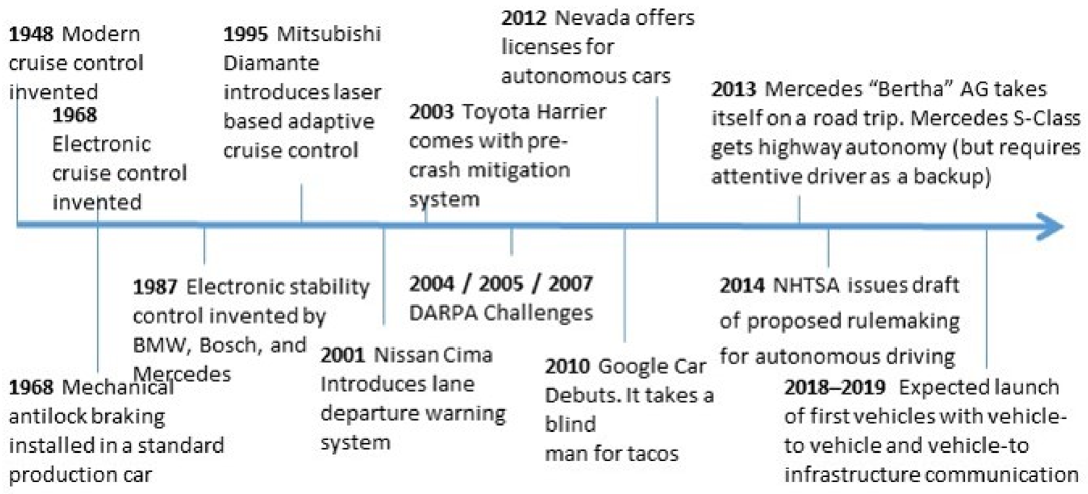
1948 Modern cruise control invented
1968 Electronic cruise control invented
1995 Mitsubishi Diamante introduces laser based adaptive cruise control
2003 Toyota Harrier comes with pre-crash mitigation system
2012 Nevada offers licenses for autonomous cars
2013 Mercedes "Bertha" AG takes itself on a road trip. Mercedes S-Class gets highway autonomy (but requires attentive driver as a backup)
1968 Mechanical antilock braking installed in a standard production car
1987 Electronic stability control invented by BMW, Bosch, and Mercedes
2001 Nissan Cima Introduces lane departure warning system
2004 / 2005 / 2007 DARPA Challenges
2010 Google Car Debuts. It takes a blind man for tacos
2014 NHTSA issues draft of proposed rulemaking for autonomous driving
2018–2019 Expected launch of first vehicles with vehicle-to vehicle and vehicle-to infrastructure communication

Figure 1.1: The historical development of autonomous driving.

1.2 The Importance of Simulation in Autonomous Driving

In today's world, developing and using software has become very important across many industries. While these software solutions provide us with numerous conveniences and advantages, they also introduce certain requirements. One of the most critical requirements for any software is testing. The testing phase can be used to determine whether an application fulfills its intended function or to assess its efficiency. Of course, in some sectors, this functionality takes on an even greater importance. One such area is safety. In industries like aviation and automotive, where a user's safety can be directly affected by an error in the application, testing and simulation are more important than ever. In model-based software applications, after a model is created, it moves to the verification stage. Before being loaded onto the hardware, the model undergoes several verification steps. Some of these verifications are named Model In the Loop (MIL) [5], Software In the Loop (SIL) [6], and Hardware In the Loop (HIL) [7].

- **Model in the Loop (MIL)**

  In the initial stages of system design, before real hardware or software components are implemented, the simulation and testing of system models are carried out by the Model in the Loop (MIL) testing method. This method uses mathematical models and simulations for system or component design. These models are utilized to understand how the designed system will function and behave. MIL testing is a critical tool for verifying the design and identifying potential problems at an early stage. With this method, engineers and designers can gain valuable insights into how the system will operate before actually manufacturing the hardware or fully developing the software.

- **Software in the Loop (SIL)**

  The Software in the Loop (SIL) testing method is the process of testing a system or component's software without using real hardware. This method simulates environmental factors or other parts interacting with the system while the software runs directly on a computer. This simulation evaluates the software's behavior and performance under real-world conditions. The main goal of SIL testing is to verify the software's functionality, detect errors at an early stage, and analyze interactions between the system and the software. This approach accelerates the software development and refinement processes and prevents costly errors.

- **Hardware in the Loop (HIL)**

  The Hardware in the Loop (HIL) testing method creates an environment to control and test real hardware. In this method, the hardware being tested, such as a vehicle control unit, operates in real-time, but the software simulates the physical systems (e.g., engines, sensors, actuators, etc.) interacting with the hardware in real-time. This simulation allows for the evaluation of how the hardware performs under real-world conditions.

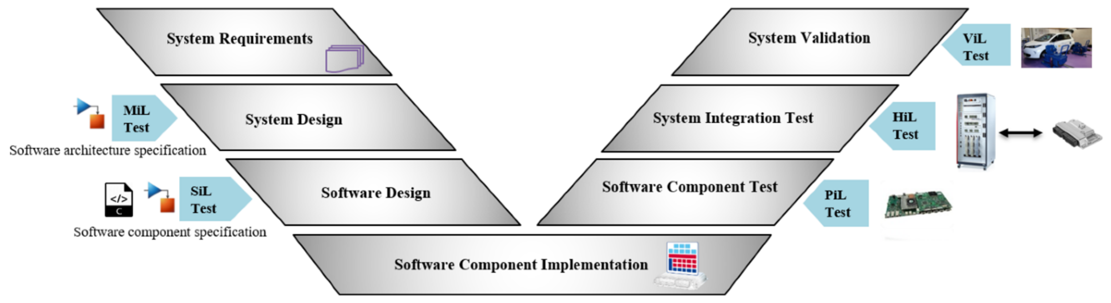

Figure 1.2: MIL, SIL , PIL, HIL and VIL tests in V-cycle development process.

Test simulators typically fall under the SIL category. The Software in the Loop (SIL) testing method has numerous advantages. These benefits explain why SIL tests are an important part of the software development process and demonstrate their value to engineers and developers across various industries. The main advantages can be listed as follows:

- **Cost Efficiency:** SIL tests are conducted in a simulated environment that does not require real hardware, thus reducing the costs associated with purchasing, maintaining, and repairing hardware. Additionally, early diagnosis of potential errors helps prevent more costly problems later on.

- **Rapid Feedback Loop:** Testing software in a model allows for quick iterations in the development process. This enables developers to instantly see the effects of their code and make fast modifications if necessary.

- **Risk Reduction:** Tests conducted without real hardware use a simulated environment to safely examine situations that could be potentially dangerous or harmful to the hardware. This is especially important when expensive hardware is involved.

- **Broad Test Scenarios:** SIL tests are capable of quickly and easily testing a wide range of scenarios that mimic real-world conditions. Developers can experiment with system parameters, error states, and various operational conditions.

- **Acceleration of the Development Process:** SIL tests can speed up the development process instead of waiting for the hardware to be ready. Testing the software alongside hardware development shortens the time to market for the product.

- **Preparation for Integration and System Tests:** When software is successfully tested in a SIL environment, the transition to more complex testing stages where hardware and software are run together becomes easier. This helps reduce problems that may arise during system tests and integration.

- **Flexibility in the Development Process:** Software can be tested and developed with various hardware platforms and configurations. This flexibility allows for the evaluation of how the software will perform on different systems.

Figure 1.3: Driving Test Simulator

1.3 State of the Art in Autonomous Driving Simulators

Advancements in the field of autonomous driving have led to the development of autonomous driving simulators as well. Nowadays, a wide range of simulators produced by various companies are in use. These include open-source projects (e.g., CARLA [8], AirSim [9]), commercial software, and customized simulation solutions used in academic research. At this point, it is crucial to clearly define our expectations from a test simulator and make a choice accordingly. Therefore, we can start by examining the popular autonomous driving simulation tools and platforms available in the market. The features I will focus on while reviewing these platforms include graphic quality, accuracy of the physics engine, sensor simulation, simulation of traffic and pedestrians, weather conditions, and the ability to simulate at different times of the day.

1.3.1 Waymo Simulator

Figure 1.4: Waymo Simulator Logo

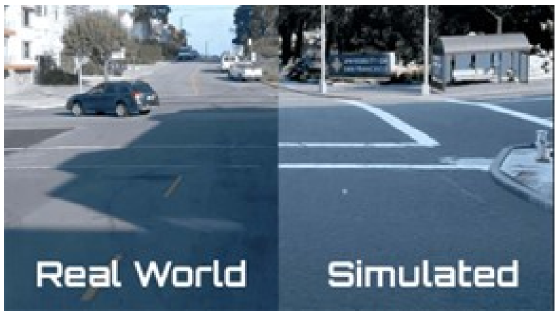

Figure 1.5: Comparing Waymo Simulator graphics and the real world.

Waymo possesses an advanced simulation platform[10] and is a leader in autonomous vehicle technology. The development and testing of autonomous vehicles are a crucial component of this platform. Considering the following features, the Waymo simulator holds a significant position among autonomous driving simulators:

- **Graphic Quality:** The Waymo simulator is highly successful in mimicking real-world environments and scenarios. The quality of graphics is particularly important for simulating the environmental perceptions of vehicle sensors because the realism of the simulation is crucial for accurately training algorithms.
- **Accuracy of the Physics Engine:** The physics engine is designed to realistically model the vehicle's movement dynamics, collisions, and surface interactions. Thanks to this accuracy, vehicles can simulate physical conditions in the real world accurately.
- **Sensor Simulation:** Waymo can model the data collection capabilities of various sensors, such as LIDAR, radar, and cameras. These simulations use advanced algorithms to accurately reflect the sensors' environmental perceptions.
- **Traffic and Pedestrian Simulation:** The simulator can be used to dynamically model traffic flow and pedestrian behavior. This feature is very important for understanding how autonomous vehicles will operate in complex urban environments and variable traffic conditions.
- **Weather Conditions:** The Waymo simulator can simulate various weather conditions to test vehicle performance under different weather conditions. This

is necessary to assess the reliability of autonomous vehicles in various weather conditions like rain, snow, and fog.

- **Ability to Simulate at Different Times of the Day:** This simulator has the capability to test the effectiveness of sensors and algorithms throughout the day, especially in night vision and twilight conditions.

1.3.2 SVL Simulator

Figure 1.6: SVL Simulator Logo

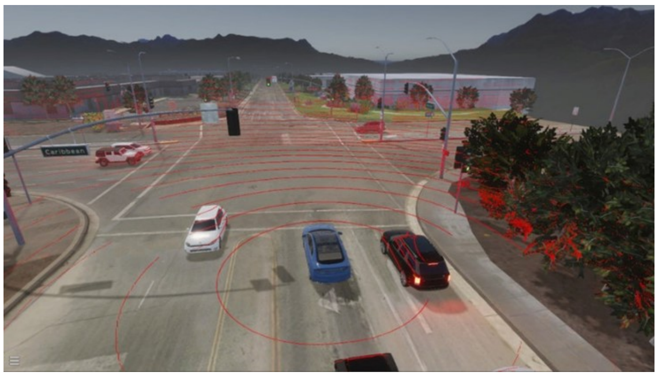

Figure 1.7: SVL Simulator Screen

The autonomous vehicle research and development community frequently uses the LGSVL Simulator[11], an open-source simulation platform. Especially academic and industrial researchers prefer this simulator because it offers realistic, flexible, and a wide variety of integration options. The LGSVL Simulator has been evaluated within the following features framework:

- **Graphic Quality:** The LGSVL Simulator provides highly realistic visual quality by using the Unity 3D game engine. This is very important for realistically modeling the environmental perceptions of vehicle sensors and realistically modeling various urban and rural environments. The quality of graphics enhances the accuracy of real-world conditions and the depth of the simulation.

- **Physics Engine Accuracy:** The LGSVL Simulator benefits from advanced physics engines to realistically model vehicle dynamics, tire-road interactions, and collision scenarios. This allows for an accurate representation of how vehicles move in the real world and interact within the simulation.

- **Sensor Simulation:** The platform supports a wide range of sensors, including LIDAR, radar, cameras, and ultrasound. Processing sensor data in the simulation and testing detection algorithms are very important because it can realistically model the detection capabilities and data streams of sensors.

- **Traffic and Pedestrian Simulation:** The LGSVL Simulator provides a comprehensive traffic and pedestrian simulation that includes intelligent traffic control systems and dynamic pedestrians. This enables the testing of autonomous vehicle algorithms with realistic traffic and social behaviors.
- **Weather Conditions and Simulation at Different Times of the Day:** The simulator can simulate different weather conditions and lighting conditions at various times throughout the day. This feature is important for assessing the performance of autonomous vehicle systems under various environmental conditions.

1.3.3 Sim4CV

Sim4CV is a simulation platform designed for researchers interested in computer vision and autonomous systems. This tool focuses on autonomous driving and autonomous flight simulations to test computer vision algorithms and model real-world scenarios [12]. Considering the following features, Sim4CV offers unique opportunities for autonomous driving research:

- **Graphic Quality:** Utilizing powerful game engines like Unreal Engine 4, Sim4CV provides high-quality graphics. This allows researchers to work with realistic images to evaluate how well algorithms adapt to real-world conditions.
- **Accuracy of the Physics Engine:** Sim4CV leverages the advanced features of Unreal Engine's physics engine to simulate vehicle dynamics, object interactions, and environmental factors. This ensures accurate modeling of vehicles and environmental objects, enhancing the realism of the simulation.
- **Sensor Simulation:** The platform can simulate cameras, LIDAR, and other sensors, which is crucial for computer vision research. The accuracy of sensor data is very important in the development and testing process of autonomous driving algorithms.
- **Traffic and Pedestrian Simulation:** Sim4CV is capable of simulating pedestrians and traffic flow, although the sophistication and details of these features depend on the version of the platform and the conditions used. Understanding how autonomous vehicles operate in complex social conditions is very important.
- **Weather Conditions and Simulation at Different Times of the Day:** Sim4CV can model variable weather conditions and simulate different times of

the day, which is important for testing how algorithms perform under various environmental conditions.

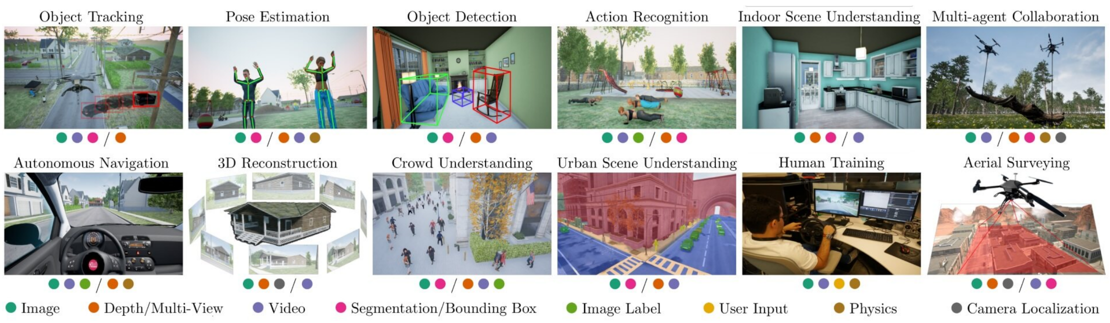

Figure 1.8: SIM4CV Simulator Logo

1.3.4 Carla Simulator

CARLA

Figure 1.9: Carla Simulator Logo

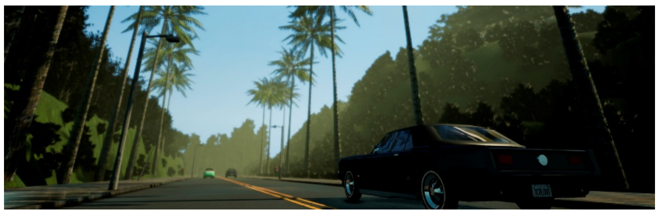

Figure 1.10: Carla Simulator Screen

The CARLA Simulator is an open-source autonomous driving simulation platform [13] that is frequently used by researchers. It has an extensive documentation page. It provides a wide range of features and tools for the development, testing, and validation of autonomous vehicles and driving algorithms. CARLA can be evaluated within the framework of the following features:

- **Graphic Quality:** With Unreal Engine 4, CARLA provides realistic and high-quality visual environments. This enhances the realism of the simulation and offers researchers an experience that closely mimics real-world conditions.
- **Accuracy of the Physics Engine:** CARLA utilizes Unreal Engine's advanced physics engine to accurately model physical processes such as collisions, vehicle dynamics, and surface interactions. This allows for accurate testing of how autonomous vehicle algorithms behave in a physical environment.

- **Sensor Simulation:** CARLA can simulate a wide range of sensors, including LIDAR, radar, cameras, and GNSS. The data from these sensors can be tested by mimicking real-world conditions.
- **Traffic and Pedestrian Simulation:** CARLA includes intelligent traffic control systems and realistic pedestrians. The dynamic traffic flow and behaviors of pedestrians are very important for autonomous vehicles to understand complex social interactions and respond appropriately.
- **Weather Conditions and Simulation at Different Times of the Day:** CARLA can simulate different weather conditions and lighting conditions at various times of the day. These features are used to assess the performance of autonomous vehicle systems under various environmental conditions.

Based on the general capabilities and user feedback, we can compare the simulators with a table. This comparison will provide an overview of how each simulator performs across various criteria:

Table 1.1: Comparative Feature Scoring of Simulators

Features / Simulators	Waymo Simulator	LGSVL Simulator	Sim4CV	CARLA Simulator
Graphics Quality	8/10	9/10	8/10	9/10
Physics Engine Accuracy	8/10	9/10	7/10	9/10
Sensor Simulation	8/10	9/10	7/10	9/10
Traffic and Pedestrian Simulation	8/10	9/10	6/10	9/10
Weather Conditions	7/10	8/10	6/10	9/10
Simulation at Different Times of Day	7/10	8/10	6/10	9/10

1.4 ADAS Applications

Advanced Driver Assistance Systems (ADAS) are becoming increasingly common in today's vehicles. Developed to enhance vehicle safety and the driving experience, these systems utilize various technologies such as cameras, sensors, and radars to identify potential hazards and are designed to alert the driver or automatically control the vehicle under certain conditions. ADAS technologies not only assist drivers in traveling more safely and comfortably but also have the potential to reduce traffic accidents. There are various applications of ADAS, which are crucial for the safety of the driver and also provide significant conveniences. Some of these applications can be briefly discussed as follows:

- **Automatic Emergency Braking (AEB):** The vehicle automatically brakes to avoid colliding with a vehicle or obstacles ahead [14]. A laser radar sensor detects the distance to the vehicle ahead and the relative speed. If there is an object in the area seen by the laser radar sensor at this speed, the brakes are activated to prevent a collision. Additionally, in some vehicles, it works in conjunction with seat belts activated by braking before a collision to help reduce injuries in cases where a collision is unavoidable.

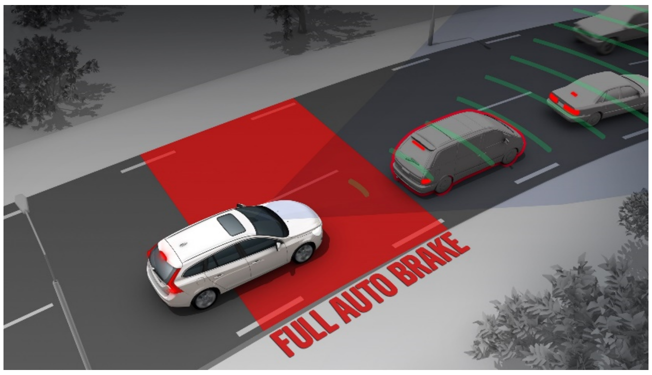

Figure 1.11: Automatic Emergency Braking system.

- **Traffic Sign Recognition System:** This system detects traffic signs and informs the driver about traffic rules such as speed limits and prohibition signs. Generally, an image is captured by a camera sensor placed on the vehicle's windshield, and the detected image is projected onto the user's screen. This technology aims to prevent accidents by ensuring that drivers do not overlook important warning signs on the road [15].

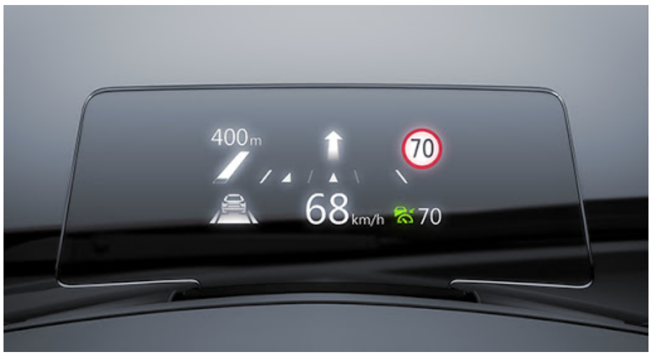

Figure 1.12: Traffic Sign Recognition System

- **Blind Spot Warning System:** This system identifies the presence of other vehicles in the vehicle's blind spots and informs the driver about them. Vehicles in the blind spot are detected using radar sensors located on the sides of the rear bumper. Typically, a warning is displayed to the driver in the side mirror, and if the driver signals to change lanes while there is a vehicle in the blind spot, an audible alert is issued. Additionally, in some models, changing lanes by braking is prevented to enhance safety [16].

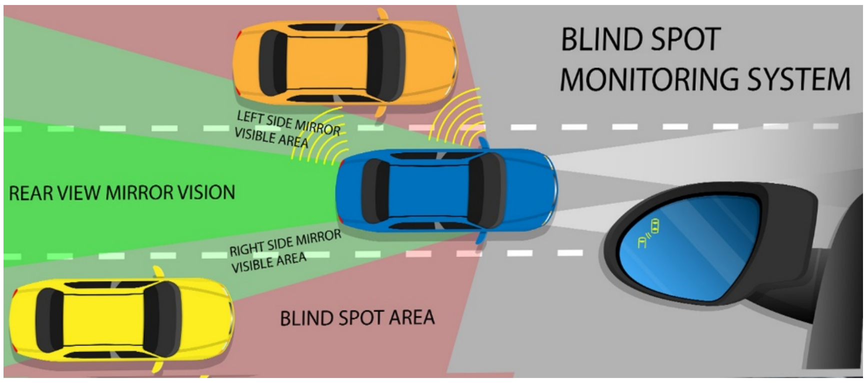

Figure 1.13: Blind Spot Warning System

- **Night Vision:** This system expands the driver's field of vision in low light conditions or at night through cameras or other sensors [17]. Utilizing thermal cameras and infrared lights, it detects living beings on the road ahead and provides audible or visual warnings to the driver.

Figure 1.14: Night Vison.

- **Parking Assistant:** This feature assists the driver in better positioning the vehicle while parking and, in some cases, can automatically park the vehicle. It utilizes parking sensors located around the vehicle to do this [18]. Parking sensors are generally electromagnetic sensors. Commonly, the system is used in a manner where the driver is still responsible for commands such as gas and gear changes.

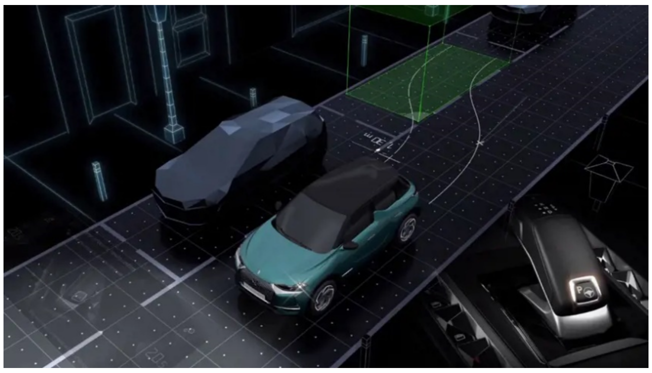

Figure 1.15: Parking Asistant

- **Fatigue Detection Systems:** Various studies have suggested that approximately 20% of all road accidents, and up to 50% on certain roads, are fatigue-related [19]. Although the implementation of this system varies among manufacturers, some brands warn the driver by detecting steering movements and how often the vehicle drifts out of its lane.

- **Automatic Headlight Control:** The effectiveness of vehicle headlights becomes even more crucial on roads with insufficient external lighting at night, such as forest roads. Oncoming drivers on these types of roads can be affected by the headlight beams. Systems like this reduce the light intensity when detecting an oncoming vehicle to minimize the adverse effects on the opposite driver. In more advanced models, the direction of the headlight beam is adjusted to achieve this effect.

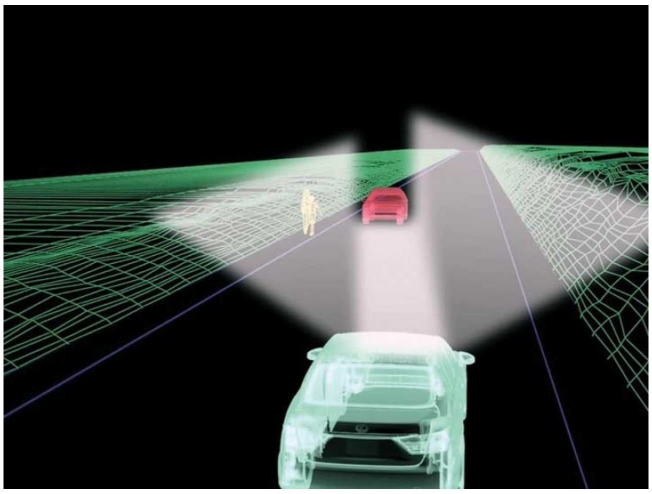

Figure 1.16: Automatic Headlight Control

- **Lane Keeping Assistant:** This system uses a camera sensor to detect the distance of the vehicle from the lane markings and activates when the determined distance falls below a certain threshold. It steers the steering wheel to direct the vehicle back into the lane, ensuring it remains within its lane boundaries.

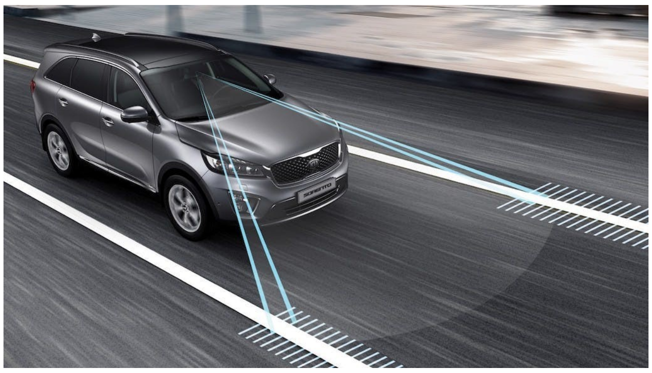

Figure 1.17: Lane Keeping Asistant

- **Adaptive Cruise Control (ACC):** This system is a more advanced version of the traditional cruise control system [20]. While a traditional cruise control maintains the vehicle at a set speed, this system adapts the speed based on the speed of the vehicle ahead. It utilizes two separate sensors to accomplish this. The first sensor is a radar sensor that measures the distance to the vehicle in front. The second sensor is a speed sensor that detects any decrease or increase in the vehicle's speed. Based on the information from these sensors, decisions are made and implemented to accelerate or decelerate. Generally, the desired distance and following speed can be adjusted by the user via controls on the steering wheel.

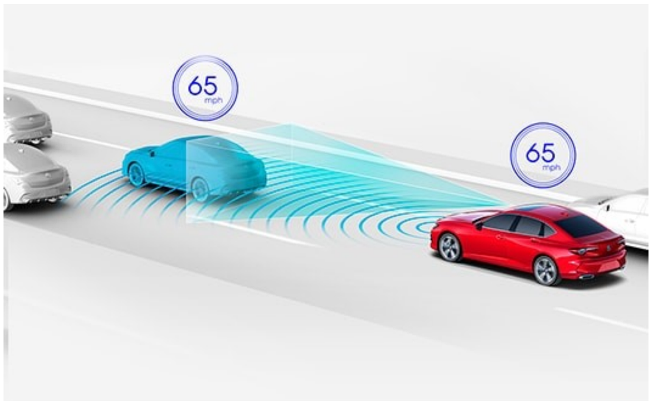

Figure 1.18: Adaptive Cruise Control

1.5 Related Works

The literature in the field of autonomous vehicle simulations focuses on a wide range of topics including various simulation techniques, sensor fusion methods, and the simulation of traffic and pedestrians. Undoubtedly, studies in these areas tend to focus more on sensor fusion methods. Sensor fusion, which combines data from multiple types of sensors to allow vehicles to clearly perceive their surroundings, is of vital importance in autonomous vehicle simulations. However, simulating the real world under varying traffic conditions is a topic of equal importance. There are two key issues to consider here. Firstly, the impact of shared smart transportation vehicles, which are becoming increasingly prevalent in our lives, on real-world traffic [21]. Secondly, the creation of real-life maps. Today, many commercial video games offer realistic environments. Researchers have created synthetic data sets using one of the most well-known video games, 'Grand Theft Auto V' [22],[23], [24]. However, this method is not practical due to some licensing requirements. As a solution to this problem, there is a method that uses OpenStreetMap data [25]. Undoubtedly, this approach will be a foundation for future studies.

Chapter 2

Interface Co-Simulation Design

2.1 Selecting the Suitable Simulator

The development and testing of autonomous driving technologies require a robust simulation environment. This environment must accurately model the real world, including vehicles, pedestrians, and various environmental conditions, while also providing comprehensive support for sensor simulation and enabling integration with analytical tools such as Matlab. After a detailed evaluation of existing simulators, including the Waymo Simulator, LGSVL Simulator, Sim4CV, and CARLA Simulator, based on critical features such as graphic quality, the accuracy of the physics engine, sensor simulation capabilities, the simulation of traffic and pedestrians, weather conditions, the ability to simulate different times of day, and compatibility with Matlab, CARLA Simulator has been identified as the most suitable choice for our research objectives. CARLA provides high-quality graphics with Unreal Engine 4 [26] and its physics engine accurately models vehicle dynamics and environmental interactions, offering a solid foundation for testing autonomous driving algorithms under various conditions. Moreover, its ability to simulate a wide range of sensors used in autonomous vehicles, such as cameras, LIDAR, radar, and GNSS, with high fidelity is crucial for the development and testing of perception algorithms. CARLA also excels in simulating dynamic traffic scenarios and pedestrian behaviors, facilitating comprehensive testing of autonomous driving systems in complex urban environments. The capability to simulate different weather conditions and times of day is important for assessing the performance of autonomous vehicle systems under various environmental conditions. For our research, the ability to integrate simulation data with Matlab for further analysis and algorithm development was a significant consideration, and CARLA's Python API facilitates easy integration with Matlab, providing a seamless workflow for data analysis and algorithm testing. Consequently, CARLA Simulator's advanced graphics, accurate physics engine, extensive sensor simulation capabilities, and effective traffic and pedestrian simulation set it as the ideal choice for our autonomous driving research. Its compatibility with Matlab further supports our analytical and development needs, making it the most suitable simulator for our project.

2.2 Carla and Anaconda Interfacing

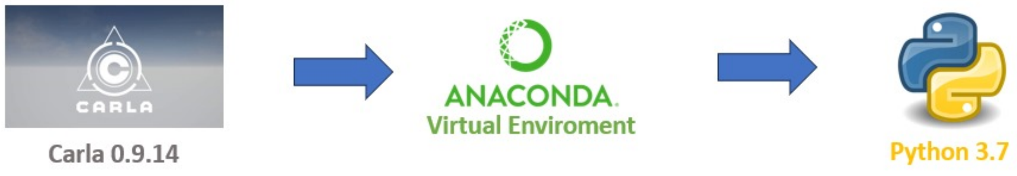

Figure 2.1: Carla Python Interfacing using Anaconda

To control and communicate with Carla, the Python API is used. Therefore, it is necessary to have a Python version compatible with the Carla version we plan to use. However, updates to Carla often require updating the Python version at regular intervals. This makes it mandatory to reconcile Carla and Python with each new update. However, by taking advantage of Anaconda's ability to use different Python versions through different digital environments, we can solve this problem. Thus, we can have multiple environments and Carla models on the same computer, which also reduces the burden of this process. The steps for setting up Carla-Anaconda communication are as follows:

- Downloaded Carla 0.9.14
- Downloaded Anaconda Navigator
- A virtual environment was created as follows:
  `create -name 'name_of_environment' python=3.7`
- Activate this environment:
  `activate 'name_of_environment'`
- The necessary Python modules were downloaded as follows:
  `pip install Carla, pygame, numpy, jupyter, opencv-python`

After these steps, the connection between Anaconda and Carla is established. When we load the Carla simulator, it comes with many example applications. Within the environment we created, we can run these examples to create traffic, manually drive a car with the keyboard, or drive a car with a steering wheel, among many other simulations.

2.3 Carla and Matlab Interfacing

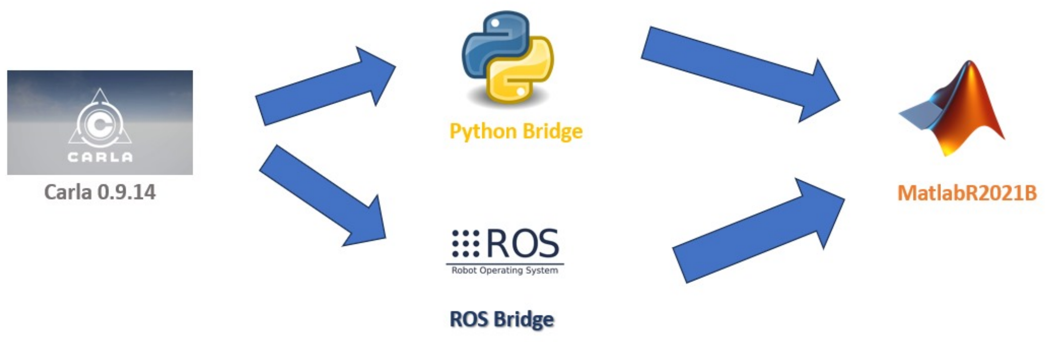

Figure 2.2: Carla Matlab Interfacing Using Pyton Bridge

There is no direct method for establishing communication between Carla and Matlab. At this point, we have two options to facilitate this communication. The first option is to use ROS bridge to connect Matlab and Carla. While this option is more advantageous for processing large amounts of data, this advantage is not necessary for our project. Additionally, its setup requires numerous compatibilities and it operates more optimally on the Linux operating system. Therefore, for ease of installation and ease of use, a Python bridge has been used in this project. The most important point to note here is the necessity for compatibility between the Matlab and Python versions. The steps required to establish the Carla-Matlab connection are as follows:

- First, we downloaded Matlab 2021B, which is compatible with Python version 3.7.
- Then, to make the `easy_install` feature operational via Python, the following steps are taken:
  - `pip install setuptools==33.1.1`

  - Add `C:\Python27\Scripts` to your 'path' (Environment variable) - `C:\Python34\Scripts. easy_install pip`
  - `easy_install carla-0.9.14-py3.7-win-amd64.egg`

- Finally, to establish the connection, the following steps are followed in Matlab:

  - `pyversion('D:\Program Files\Anaconda\envs\carla-simv2\python.exe')` (place your own path here)
  - `insert(py.sys.path, int32(0), 'D:\Program Files\Anaconda\envs\carla-simv2\Lib\site-packages\carla-0.9.14-py3.7-win-amd64.egg')` (place your own path here)
  - `py.importlib.import_module('carla')`

The last two steps must be typed into the command line at the start of every Matlab application. In this way, the necessary connection is established. Only in our application, by creating a virtual port, the Matlab-Carla connection is made ready.

2.4 Modeling the Car in Simulink Environment

In order to control a vehicle in the simulation, the controller must recognize the vehicle model. For this, a model of the vehicle is needed. Various models can be used when modeling the vehicle. Among these, the most commonly used ones usually offer a good balance between simplicity and accuracy, capable of representing the vehicle's motion dynamics and control systems. In line with the needs of this project, the Dynamic Single-Track (DST) model has been chosen. The DST model (bicycle model) can be modeled with single-track wheels (one front and one rear wheel). This is equivalent to a model where the right and left sides of a four-wheeled vehicle are considered equal.

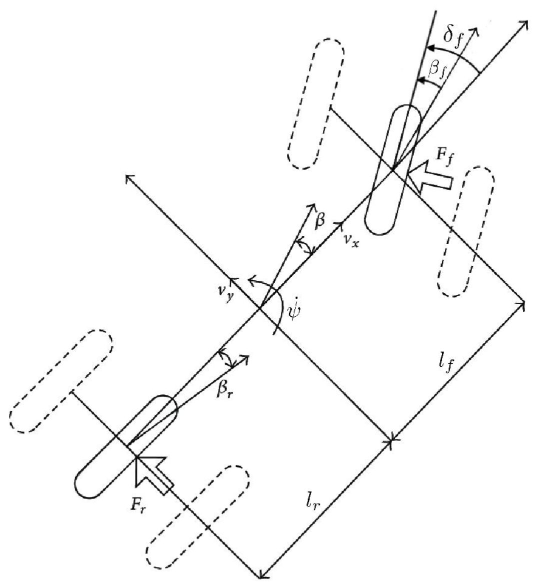

Figure 2.3: Single Track Model

Vehicle variables:

- δ_f: steering angle
- β: vehicle slip angle = angle between the vehicle longitudinal axis and velocity
- β_f, β_r: tire slip angles = angles between the tire longitudinal axis and velocity.

Vehicle parameters:

- CoG: center of gravity
- m, J: mass and moment of inertia
- l_f: distance CoG - front wheel center
- l_r: distance CoG - rear wheel center
- c_f, c_r: front/rear cornering stiffnesses.

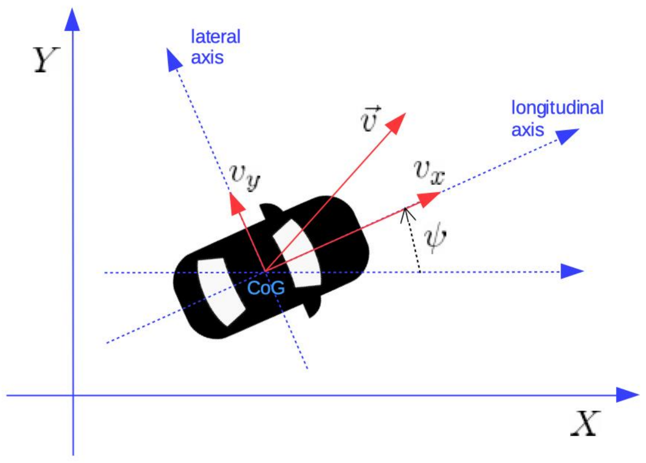

Figure 2.4: Vehicle Reference System

Vehicle Dynamic parameters:

- X, Y: coordinates of the vehicle CoG in an inertial reference frame
- ψ: yaw angle
- $\omega_\psi = \dot{\psi}$: yaw rate
- $\vec{v} \equiv V$: velocity vector in the inertial frame
- v_x: longitudinal speed = $\vec{v}$ component along the longitudinal axis
- v_y: lateral speed = $\vec{v}$ component along the lateral (transverse) axis
- a_x: longitudinal acceleration in the inertial frame.

The state equations of the DST model are:

$$\begin{aligned}
\dot{X} &= V_x \cos\psi - V_y \sin\psi \\
\dot{Y} &= V_x \sin\psi + V_y \cos\psi \\
\dot{\psi} &= \omega_\psi \\
\dot{V}_x &= V_y \omega_\psi + a_x \\
\dot{V}_y &= -V_x \omega_\psi + \frac{2}{m}(F_{yf} + F_{yr}) \\
\dot{\omega}_\psi &= \frac{2}{J}(l_f F_{yf} - l_r F_{yr})
\end{aligned}$$

where F_{yf} and F_{yr} are the lateral forces exchanged between tire and road.

For the tire model, there are several tire model options:

Linear (for $V_x = \text{const}$) tire model:

$$F_{yf} = -C_f \beta_f, \quad F_{yr} = -C_r \beta_r$$
$$\beta_f = \frac{(V_y + l_f \omega_\psi)}{V_x} - \delta_f, \quad \beta_r = \frac{(V_y - l_r \omega_\psi)}{V_x}$$

Nonlinear simplified tire model:

$$F_{yf} = -C_f \beta_f \cos \delta_f, \quad F_{yr} = -C_r \beta_r$$
$$\beta_f = \arctan\left(\frac{(V_y + l_f \omega_\psi)}{V_x}\right) - \delta_f,$$
$$\beta_r = \arctan\left(\frac{(V_y - l_r \omega_\psi)}{V_x}\right)$$

Nonlinear Pacejka's tire model:

$$F_{yf} = -f_p(\beta_f) \cos \delta_f, \quad F_{yr} = -f_p(\beta_r)$$

where β_f and β_r and $f_p(\beta)$ is given by the Pacejka's magic formula.

Pacejka's magic formula:

$$f_p(\beta) = p_1 \sin\left(p_2 \arctan\left(p_3\beta - p_4\left(p_3\beta - \arctan(p_3\beta)\right)\right)\right)$$

p_1: peak value, p_2: shape factor, p_3: stiffness factor, p_4: curvature factor. Linearizing this formula, we find $p_1 p_2 p_3 = C_f$ (or C_r).

In the real world conditions, these parameters are really hard to measure or estimate because they change based on the road conditions.

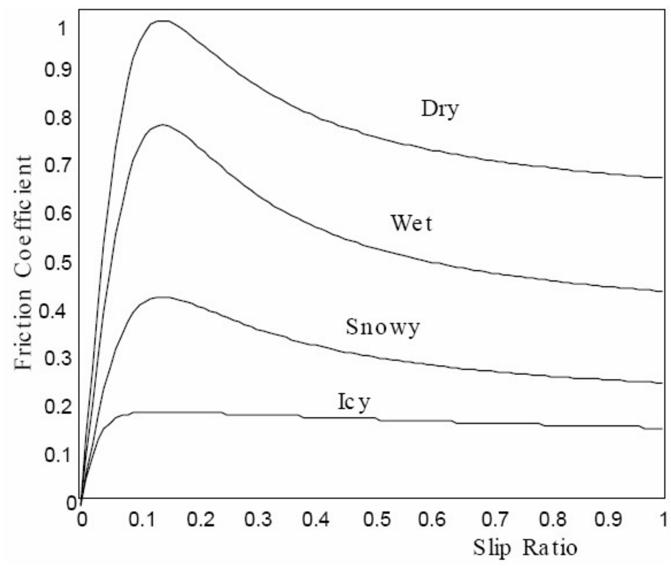

Figure 2.5: Friction Coefficents in different conditions

The real parameters of the vehicle in the simulation environment have been found with the help of Carla's `Vehicle.physics` command.

Vehicle Parameters Values:

- m: 1318 kg
- J: 2500 kg/m^2
- l_f: 1.54 m
- l_r: 1.51 m
- c_f: 15000 N/rad
- c_r: 15000 N/rad

2.4.1 Dispatching To Obtain Throttle-Brake Value From Acceleration Value

To control a vehicle in the Carla simulator, we need three pieces of information: steering angle, throttle, and brake commands. However, our model generates ax data. It is not possible to directly provide this data to Carla because this information only tells us about the acceleration or deceleration of the vehicle and the magnitude of these changes. Therefore, we need to scale this data to use it for throttle-brake information. To do this, we perform dispatching. In the context of control, "dispatching" usually refers to the process of distributing tasks or resources according to certain criteria or algorithms.

To accomplish this, we first need specific reference values. These reference values should be collected according to many different scenarios to ensure they are suitable for every scenario, not just one. In our case, data were collected for three different scenarios: when the vehicle completes a straight path in a sine wave pattern, when continuous braking and accelerating are performed on a straight road, and finally, when only accelerating is done and the vehicle slows down without braking due to its own weight. The data include the vehicle's ax acceleration value, and throttle and brake values. By examining these accelerations, the priorities of throttle and brake values are determined for acceleration and deceleration situations. For instance, it is not always necessary to brake for negative acceleration values; reducing the throttle can also achieve the necessary speed reduction. Therefore, each value has its weight. Also, there is a range within which each command should be applied. These ranges can be determined using an if/else structure based on the acceleration value. The commands to be applied within these ranges are found by dividing the *ax* value by certain weight values. Consequently, the throttle and brake values are instantaneously derived from the *ax* value and provided to the vehicle in the Carla environment for application. As explained above, these values were found by analyzing reference data and through trial and error. The code for the if/else structure that creates the intervals for applying these values and commands is provided below.

```
% Parameters of throttle-brake dispatching
g1= 2.4; %2.4
g2= 8;
ab= -2;
tbp=[g1 g2 ab];

function [th,br]=dispa(p,ax)
if ax>0
    th=ax/p(1);
    br=0;
elseif ax>p(3) && ax<=0
    th=0;
    br=0;
else
    th=0;
    br=-ax/p(2);
end
```

2.5 Data Gathering with Autonomous Driving Mode

A reference point was selected from the Carla Map to collect reference data. The vehicle was spawned at this reference point. A simulation duration was determined based on a finish point we had predetermined. The vehicle was moved using Carla's autonomous driving mode. During this time, the necessary reference position, yaw angle, and speed data were recorded into a `.txt` file at intervals of $T_s = 0.05$ seconds. Once the required simulation duration was completed, the vehicle was removed from the map.

```
%% Collect and Store the Simulation Data

clear all
close all
```

```
clc

insert(py.sys.path, int32(0), 'D:\Program Files\anaconda\envs\carla-sim\Lib\site-packages\carla-0.9.14-py3.7-win-amd64.egg');
py.importlib.import_module('carla')

            port = int16(2000);
             client = py.carla.Client('localhost', port);
             client.set_timeout(10.0);
             world = client.get_world();

             % Spawn Vehicle
             blueprint_library = world.get_blueprint_library();
             car_list = py.list(blueprint_library.filter("leon"));
             car_bp = car_list{1};
             spawn_point =  world.get_map().get_spawn_points();
             spawn_location = spawn_point{9};
%            spawn_point = py.random.choice(world.get_map().get_spawn_points());


%          spawn_point.location.x = 0;
%          spawn_point.location.y = 133.9472;
%          spawn_point.location.z = 0.6;
%          spawn_point.rotation.yaw = 0;

             obj.car = world.spawn_actor(car_bp, spawn_location); %% spawn the car

             obj.car.set_autopilot(true);
%              control = obj.car.get_control();
%              control.throttle = 0.5;
%              control.steer = 0;
%              obj.car.apply_control(control);

             fileID = fopen('vehicle_position_reference.txt', 'w');

             for i = 1:420
    % Vehicle Location
    vehicle_transform = obj.car.get_transform();
             orientation = vehicle_transform.rotation;

    vehicle_location = obj.car.get_location();
    x_position = obj.car.get_location().x; % X
    y_position = obj.car.get_location().y; % Y
    yaw_angle = double(deg2rad(orientation.yaw)); %yaw angle as radian
    x_velocity = obj.car.get_velocity().x; % X
%     throttle =  obj.car.get_control.throttle;
%      yaw_angle=[atan2(diff(y_position),diff(x_position));0];
      % Location  Data
     fprintf(fileID ,  '%f ,  %f,%f,%f\n',  x_position ,  y_position ,
    yaw_angle,x_velocity);


    % Sample  time
     pause(0.05);
             end


             fclose(fileID);
             pause(1)
             reference_pos_error=importdata('
    vehicle_position_reference.txt');
             pause(1)
             save('pid_reference',  'reference_pos_error');
             pause(1)
             obj.car.destroy();
```

2.6 Data Gathering with Manual Driving Mode

The first capability that we demonstrate with CARLA is localization, which allows our ego-vehicle to determine its pose in the world. Two coordinate frames: the map frame, which is a coordinate frame that is fixed at the initial position of the map, and the vehicle frame, which is a coordinate frame attached to the middle of the rear axle of the vehicle. For our particular experiment, we record the vehicle's accurate pose while traversing a curved-straight route in the Town10. In the Python API, there is a class that comprises all the localization information for an actor at a certain moment in time; its methods comprise Getters such as `get_acceleration`, `get_velocity`, `get_transform`, and `get_angular_velocity`- which in our case we utilize the `get_transform` method which includes both location of the object (X, Y, Z from the origin of the map) in meters, and its rotation characteristics from which we utilize the yaw values. More concisely, X and Y coordinates were our focus, as in the map chosen, the road is entirely flat. Hence only these two coordinates remain crucial for tracking the vehicle's trajectory. Also, the Yaw angle, describing the orientation of the map's coordinate system, is essential for indication of the direction the vehicle is facing. Therefore for curved paths, which our scenarios contain, it is important.

Some additional considerations also are to be noted, such as the sampling rate, which determines the time step of the data collection. We have chosen 0.1, indicating not too sparse to miss some critical dynamics, especially for sharp turns, and also not too frequent leading to redundant information.

After initializing the scenario and ego vehicle (based on the preferences), based on the duration of the scenario depending on the controls of the car (throttle, brake, and steering commands), we record the data of the vehicle at each time step; for better understanding the following Script used in the development is provided;

```
start_time = time.time()
duration = 60

def find_weather_presets():
    rgx = re.compile('.+?(?:(?<=_[a-z])|(?<=^[A-Z]))(?<!=[A-Z][a-z])$')
    name = lambda x: '_'.join(m.group(0) for m in rgx.finditer(x))
    presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
    return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]
```

```
def get_state(self,):
    acceleration = self.player.get_acceleration()
    steer_value = self.player.get_control().steer
    throttle_value = self.player.get_control().throttle
    brake_value = self.player.get_control().brake
    location = self.player.get_location()
    velocity = self.player.get_velocity()
    transform = self.player.get_transform()
    yaw = transform.rotation.yaw
    yaw_rate = self.player.get_angular_velocity().z
    yaw_rate = yaw_rate * math.pi/180
    yaw = yaw * math.pi/180
```

```
while True:
    clock.tick_busy_loop(60)
    if controller.parse_events(world, clock):
        return
    world.tick(clock)
    world.render(display)
    pygame.display.flip()

    global start_time
    global duration
    current_time = time.time()

    if current_time - start_time > duration:
        break
    world.get_state()

    time.sleep(0.1)
```

Chapter 3

Different Control Strategies in Simulation for Path Tracking

3.1 PID Path Tracking Lateral Control

Using PID control for vehicle management is a much more straightforward method compared to other types of control methods. As explained above, the objective in PID applications is to minimize the instantaneous error. To be able to perform any maneuver we want with the vehicle, we need to ensure both lateral and longitudinal control. Therefore, to provide lateral control, we must supply the correct δ_f data to the vehicle, and to provide longitudinal control, we must provide the correct ax data to the vehicle. Since ax data cannot be directly given to the vehicle, we use the throttle-brake acquisition method utilizing acceleration described in the system-identification section. To find these data, we need a reference. We obtained our references through the autonomous driving mode provided by the Carla simulator. This process will be explained in detail below. Additionally, to obtain the δ_f data, we use a reference matrix of size $N \times 3$ consisting of x position, y position, and yaw angle, while for obtaining ax data, a matrix of size $N \times 1$ consisting of V_x reference speeds is created. These references are compared with the data we receive from the vehicle instantaneously to create instant error values. These instant error values are e_{ct} or e_h for δ_f, and for ax, it is the difference between speeds. To reduce these two different error data, we need two different controllers. These two different PID controllers have different parameters. These parameters were obtained through trial and error. Normally, there are many methods to find the PID parameters as explained above, but since our system is not a linear system, the most straightforward way is trial and error. After the PID controller, the necessary data is given to the vehicle we created in the Carla environment, and the simulation is ready. All the blocks used in this application will be explained in detail.

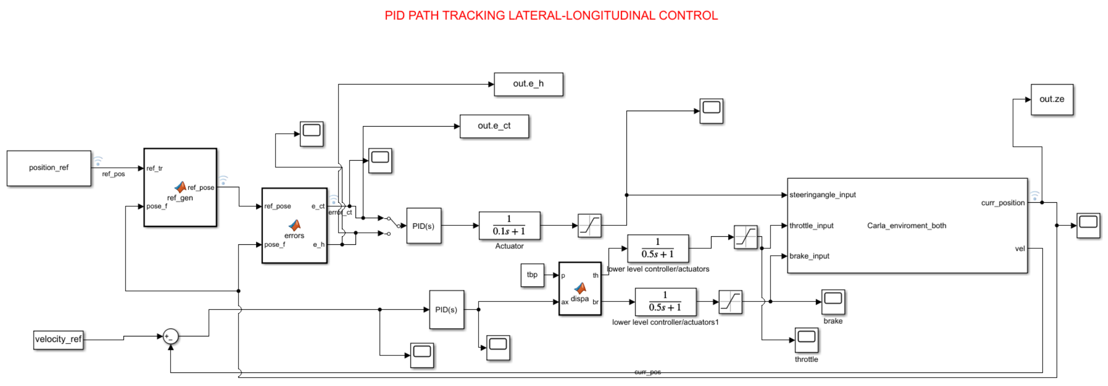

Figure 3.1: PID Control Simulink Model

3.1.1 PID Control

PID control is a commonly used term to describe a control system consisting of three components, which are the proportional (P), integral (I), and derivative (D) terms [27]. These terms form the basis of the standard three-term controller, with each letter in "PID" representing one of these components. PID controllers are widely employed in industrial settings and are often the central control element in complex control systems. Despite advancements in technology, the three-term PID controller has remained relevant, transitioning from analog to digital control systems seamlessly. It was one of the first controllers to be mass-produced for use in the process industries. The introduction of the Laplace transform facilitated the study of feedback control system performance, contributing to the widespread adoption of PID control in engineering. Theoretical analysis of PID control performance is greatly facilitated by the straightforward representation of an integrator using [1/s] and a differentiator using [s] in the Laplace transform. Conceptually, the PID controller is quite sophisticated and can be represented in three different ways.

Firstly, there is a symbolic representation (Figure 3.2), where each of the three terms can be adjusted to achieve different control actions. Secondly, there is a time-domain operator form (Figure 3.3), and finally, there is a Laplace transform version of the PID controller (Figure 3.4). This provides the controller with an *s*-domain operator interpretation and enables the discussion of PID controller performance to include the link between the time domain and the frequency domain.

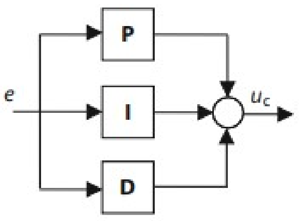

Figure 3.2: Symbolic Representation of PID Paramters

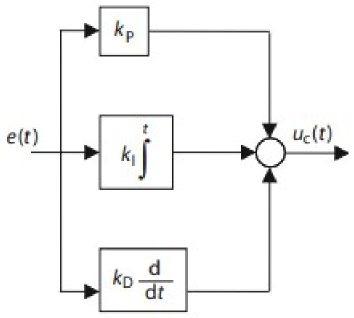

Figure 3.3: Time Domain Operator Form of PID Parameters

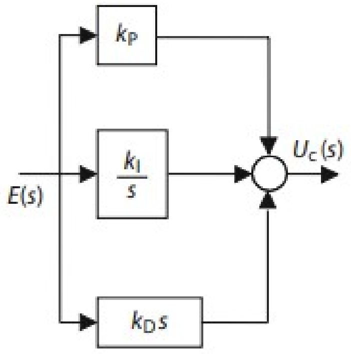

Figure 3.4: Laplace Transform of PID Parameters

- Symbolic forms: e, u_c
- Time domain forms: $e(t)$, $u_c(t)$
- Laplace domain forms: $E(s)$, $U_c(s)$
- K_p: Proportional Gain
- K_i: Integral Gain
- K_d: Derivative Gain

The proportional term generates a control signal directly proportional to the magnitude of the current error. The integral term accounts for the accumulated error over time, contributing this cumulative error to the control signal. The derivative term calculates the rate of change of the error, incorporating this rate into the control signal to respond more quickly to rapidly changing errors. By integrating these three terms, PID control achieves the fastest and most precise adjustment to the desired target.

The mathematical expression of PID control in the time domain can be expressed as follows:

$$u(t) = K_p e(t) + K_i \int e(t)\,dt + K_d \frac{de}{dt} \tag{3.1}$$

It is crucial to accurately adjust PID parameters. At this juncture, there are several distinct methods for tuning PID parameters. These include trial and error, the Ziegler-Nichols method, the Cohen-Coon method, and automatic tuning algorithms. Choosing among these methods entirely depends on the application at hand.

Ziegler-Nichols method: The Ziegler–Nichols technique is a widely used approach for adjusting the gains of P, PI, and PID controllers [28]. Initially, this method involves setting both integral and derivative gains to zero and incrementally increasing the proportional gain until the system reaches instability. The proportional gain at the brink of instability is referred to as K_{MAX}, with f_0 denoting the oscillation frequency at this point. Subsequently, the method reduces the proportional gain by a specific amount and determines the integral and differential gains based on f_0. The gains for P, I, and D are then established in accordance with a designated table.

	K_p	K_i	K_d
P controller	$0.5K_{\mathrm{MAX}}$	0	0
PI controller	$0.45K_{\mathrm{MAX}}$	$1.2f_0$	0
PID controller	$0.6K_{\mathrm{MAX}}$	$2f_0$	$0.125/f_0$

Tuning rules are highly effective for systems equipped with an analog controller, characterized by linearity, monotonic behavior, and a slow response. These rules excel in scenarios where the system's response is primarily governed by a single-pole exponential "lag" or exhibits behavior closely resembling one [29].

Cohen-Coon method: The Cohen-Coon tuning method addresses the sluggish steady-state response observed with the Ziegler-Nichols technique, especially in systems where there's a significant dead time or process delay compared to the open-loop time constant. It's particularly practical for cases with a substantial process delay, as smaller delays might lead to the prediction of excessively high controller gains. This method is specifically tailored for first-order models that include time delay, recognizing that the controller's response to disturbances is not immediate but progressive, as opposed to an instantaneous step disturbance [30]. The Cohen-Coon tuning approach is considered an 'offline' tuning method, which allows for the introduction of a step change to the system's input once it has reached a steady state. Following this step change, the system's output is observed and measured in terms of the time constant and the time delay. This measured response is then utilized to calculate the initial control parameters.

In the Cohen-Coon method, specific predefined settings aim to achieve minimal offset and a standard Quarter Decay Ratio (QDR) of 1/4. The Quarter Decay

Ratio signifies a damping characteristic where the amplitude of each successive oscillation decreases to a quarter of the amplitude of the preceding oscillation. These predetermined settings, which help attain the desired decay ratio and minimal offset, are detailed in a specific table.

	K_c	T_i	T_d
P	$\frac{P}{NL}\left(1+\frac{R}{3}\right)$	0	0
PI	$\frac{P}{NL}\left(0.9+\frac{R}{12}\right)$	$\frac{L(30+3R)}{9+20R}$	0
PID	$\frac{P}{NL}\left(1.33+\frac{R}{4}\right)$	$\frac{L(30+3R)}{9+20R}$	$\frac{4L}{11+2R}$

Where:

- P: percent change of input
- N: Percent change of output / τ
- L: τ_{dead}
- R: τ_{dead}/τ

PID controllers are extensively utilized across diverse engineering disciplines, playing a crucial role in addressing real-time engineering challenges by ensuring rapid and uniform system responses [31]. Within power systems, they are essential for controlling system behaviors, thereby contributing significantly to technological progress and societal development [32]. In industrial settings, PID controllers are valued for their ability to maintain stability, demonstrate favorable transient characteristics, and possess a straightforward and understandable configuration [33]. Their widespread adoption in engineering practices can be attributed to their simplicity, tolerance to modeling inaccuracies, and ease of use [34]. Specifically, in the context of electro-hydraulic servo systems, PID controllers are enhanced through various techniques, including parameter optimization, real-time adjustment methods, and integrated control approaches, to achieve exceptional accuracy, robustness, and swift reaction times [35].

The main sectors where PID controller is used are:

- **Manufacturing Industry:** PID controllers are used to conveniently control parameters such as temperature, pressure, speed in manufacturing processes. For example, plastic injection and metal casting processes require precise temperature control to achieve the desired quality.

- **Automation Systems:** PID controllers are used to control precise position and speed in automated assembly lines, robotic systems and other automation applications.
- **Process Control:** PID controllers are used to control important process variables such as liquid levels, pH levels, and chemical reaction rates in industries such as food processing, oil refining, and chemicals.
- **Energy Production and Distribution:** PID controllers are very important to ensure the balance of production and consumption in power plants and electricity distribution networks and to maintain the stability of the network.
- **HVAC (Heating, Ventilation and Air Conditioning) Systems:** Indoor temperature, humidity and air quality are controlled by PID controllers. These systems provide precise control to maximize energy consumption and at the same time maintain comfort levels.
- **Space and Aviation:** PID controllers are extremely precise and reliable for applications such as satellite positioning systems, orbital control of spacecraft and autopilot systems of aircraft.
- **Vehicle Control Systems:** Safety systems such as ABS (Anti-Lock Braking System) and ESP (Electronic Stability Program) direct the behavior of the vehicle as desired using PID controllers.

3.1.2 Finding The Closest Point In The Road

In path tracking applications, especially when utilizing PID controllers, it is crucial to accurately follow a predetermined path. The effectiveness of the tracking can be significantly influenced by how well the vehicle or object adheres to a reference trajectory. The MATLAB function *ref_gen* plays a pivotal role in this process by generating a reference pose from the reference trajectory based on the current position of the vehicle or object.

```
function ref_pose = ref_gen(ref_tr , pose_f)
```

The `ref_gen` function is designed to select the most relevant reference pose from a set of trajectory points. This selection is crucial for the PID controller to compute the necessary adjustments to minimize the deviation from the path.

- `ref_tr`: This input argument represents the reference trajectory, a matrix where each row corresponds to a point along the trajectory, with columns typically representing the x and y coordinates of these points.

- `pose_f`: This input denotes the current pose of the vehicle or object, usually a vector containing at least the current x and y positions.

1. Determine the Number of Points in the Trajectory
   (`N = size(ref_tr,1);`)

   This step calculates the number of points (`N`) in the reference trajectory by assessing the first dimension (rows) of `ref_tr`.

2. Calculate Distances Between Each Trajectory Point and the Current Pose

   ```
   (dis = vecnorm(ref_tr(:,1:2)' - pose_f(1:2)*ones(1,N));)
   ```

   Here, the Euclidean distance between each point on the reference trajectory and the vehicle's current position is calculated. This is achieved by subtracting the vehicle's position replicated `N` times (to match the trajectory points' dimensions) from each trajectory point and computing the norm of the resulting vectors, resulting in a distance vector `dis`.

3. Identify the Closest Trajectory Point
   (`[ ,c] = min(dis);`)

   This step locates the index (`c`) of the smallest value in `dis`, indicating the trajectory point closest to the vehicle's current pose.

4. Select the Reference Pose
   (`ref_pose = ref_tr(c,:)';`)

   The function then selects the closest trajectory point using the index `c`, assigning it as `ref_pose`. This output is a vector representing the x and y coordinates (and potentially additional information if included in `ref_tr`) of the closest point on the reference trajectory to the vehicle's current location.

In conclusion, The `ref_gen` function is integral to path tracking in control applications, providing a dynamic reference pose that guides the PID controller's adjustments. Determining the closest point on the reference trajectory to the current position ensures that the control strategy remains focused on minimizing path deviations, thereby enhancing tracking accuracy and efficiency.

```
function ref_pose=ref_gen(ref_tr,pose_f)
N=size(ref_tr,1);
% reference trajectory closest point to the vehicle
dis=vecnorm(ref_tr(:,1:2)'-pose_f(1:2)*ones(1,N));
[~,c]=min(dis);
```

```
ref_pose=ref_tr(c,:)';

end
```

3.1.3 Finding The Instantaneous Error

Heading and cross tracking errors can be suitably used for lane keeping control.

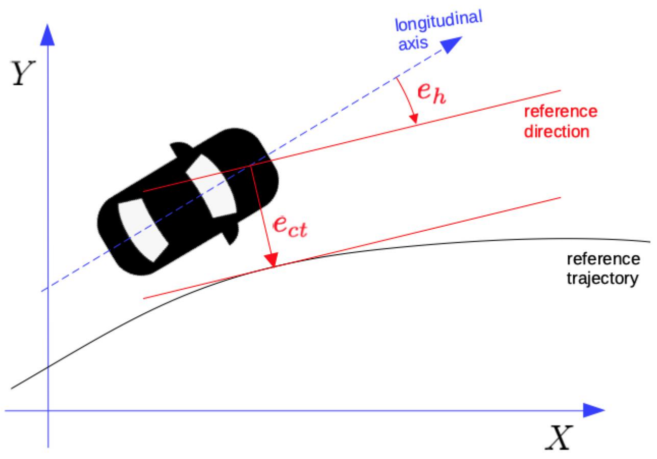

Figure 3.5: Cross-Track and Heading Error representation in Reference Trajectory

- **Heading error e_h:** angle between the vehicle longitudinal axis and the reference direction.
- **Cross-track error e_{ct}:** displacement from the center of the vehicle front axle to the closest point on the trajectory.

To calculate the two errors, we establish the following metrics:

- $p_a \doteq (X_a, Y_a, \psi)$: vehicle front axle pose
- $p_r \doteq \{p_{1r}, \ldots, p_{Nr}\}$: reference trajectory
- $p_r^i \doteq (X_{ir}, Y_{ir}, \psi_{ir}) \in p_r$: reference pose
- (X_r^C, Y_r^C): trajectory point closest to the vehicle: $c = \arg\min_i \sqrt{(X_i^r - X_a)^2 + (Y_i^r - Y_a)^2}$
- ψ_r^c: corresponding reference yaw angle

- Heading error: $e_h \doteq \psi_r^c - \psi$.
- Cross-track error: $e_{ct} \doteq (Y_r^c - Y_a)\cos\psi_r^c - (X_r^c - X_a)\sin\psi_r^c$ is the signed distance between the vehicle and the reference trajectory

Define the 3D vectors:

- $\boldsymbol{\rho} \doteq \begin{bmatrix} X_r^c - X_a \\ Y_r^c - Y_a \\ 0 \end{bmatrix}$ vector from (X_a, Y_a) to (X_r^c, Y_r^C)
- $\boldsymbol{\xi} \doteq \begin{bmatrix} \cos\psi_r^c \\ \sin\psi_r^c \\ 0 \end{bmatrix}$ reference direction of motion.

Their cross-product is $\boldsymbol{\xi} \times \boldsymbol{\rho} = \begin{bmatrix} 0 \\ 0 \\ e_{ct} \end{bmatrix}$.

Since $\boldsymbol{\xi}$ and $\boldsymbol{\rho}$ are orthogonal, $\|\boldsymbol{\xi} \times \boldsymbol{\rho}\| = \|\boldsymbol{\xi}\|\|\boldsymbol{\rho}\| = \|\boldsymbol{\rho}\|$. Hence;

$$|e_{ct}| = \|\boldsymbol{\rho}\| = \sqrt{(X_r^c - X_a)^2 + (Y_r^c - Y_a)^2}$$

The sign of e_{ct} is as follows:

- vehicle on the left of the reference trajectory $\rightarrow e_{ct} < 0$
- vehicle on the right of the reference trajectory $\rightarrow e_{ct} > 0$.

The MATLAB function `errors` is designed to calculate two types of errors in path-tracking applications, particularly for systems utilizing PID control: the cross-track error (e_{ct}) and the heading error (e_h). These errors are essential for adjusting the control inputs to minimize deviations from a desired path and orientation.

In conclusion, the `errors` function is a pivotal component in path-tracking control systems, providing essential metrics for error correction. Calculating both cross-track and heading errors, enables a PID controller or any other control strategy to correct positional and orientational deviations effectively. This dual-error approach ensures that the vehicle not only remains on the desired path but also maintains the correct orientation throughout its trajectory, which is critical for navigation precision and operational efficiency in autonomous or semi-autonomous vehicles.

3.1.4 Creating Carla Environment

To create a Carla environment, the Matlab s-function block is utilized. This block has three inputs and two outputs. The inputs are throttle-brake and steering angle, while the outputs are instant position and speed data. The vehicle is spawned at our starting point in the reference location. The steering angle is matched to the δ_f data we receive instantaneously using the `control.steering` command, while the throttle/brake data are obtained using the `control.throttle` and `control.brake` commands. After these values are found, they are applied to the vehicle with the `apply.control` command. The output values mentioned above are the same as the commands in the reference acquisition section, obtaining instant position in a 1×3 matrix and instant speed in a 1×1 matrix format. These matrices are then given as outputs. At the end of the application, the vehicle is removed from the map.

```
    classdef Carla_enviroment_both < matlab.System
    % Carla Enviroment for Lateral Control

    % Public, tunable properties
    properties
        steeringangle_input=0;
        throttle_input = 0;
        brake_input=0;

    end

    properties(DiscreteState)

    end

    % Pre-computed constants
    properties(Access = private)
        car;
    end

    methods(Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing
    constants
            port = int16(2000);
            client = py.carla.Client('localhost', port);
            client.set_timeout(10.0);
            world = client.get_world();

            % Spawn Vehicle
            blueprint_library = world.get_blueprint_library();
            car_list = py.list(blueprint_library.filter("leon"));
            car_bp = car_list{1};
            spawn_point = py.random.choice(world.get_map().
    get_spawn_points());
        spawn_point.location.x = -111.1204;
        spawn_point.location.y = 72.8989;
        spawn_point.location.z = 0.6;
        spawn_point.rotation.yaw = 85.6422;

            obj.car = world.spawn_actor(car_bp, spawn_point); %%
    spawn the car
            obj.car.set_autopilot(false)

        end

        function [curr_position,vel] = stepImpl(obj,
    steeringangle_input,throttle_input,brake_input)
            %
            pause(0.001);

            vehicle_transform = obj.car.get_transform();
            orientation = vehicle_transform.rotation;

            x_position = obj.car.get_location().x;
            y_position = obj.car.get_location().y;
%             curr_velocity = obj.car.get_velocity.x(); %% x
    represent longitudinal y, lateral
     control = obj.car.get_control();
    control.steer = rad2deg(steeringangle_input)/70;
     control.throttle = throttle_input;
     control.brake = brake_input;
     obj.car.apply_control(control);


%             yaw_angle = double(orientation.yaw);
             yaw_angle = double(deg2rad(orientation.yaw));

            curr_position = [x_position,y_position,yaw_angle]';
            vel =  obj.car.get_velocity().x;

        end

        function [distance,v] = isOutputComplexImpl(~)
            distance = false;
            v = false;

        end

        function [distance,v] = getOutputSizeImpl(~)
```

```
            distance = [3,1];
            v = [1,1];


        end

        function [distance,v] = getOutputDataTypeImpl(~)
            distance = 'double';
            v = 'double';


        end

        function [distance,v] = isOutputFixedSizeImpl(~)
            distance = true;
            v=true;

        end

        function resetImpl(~)
            % Initialize / reset discrete-state properties
        end
    end

    methods(Access= public)
        function delete(obj)
            % Delete the car from the Carla world
            if ~isempty(obj.car)
                obj.car.destroy();
            end
        end
    end
end
```

3.1.5 PID Values

While applying the PID control method, the most important aspect is finding the appropriate parameters. These parameters can be determined for a linear model using different calculation methods, which have been described in detail. However, our model is nonlinear, making it impossible to use these methods. Therefore, through the method of trial and error that I last described, different parameters have been calculated for each scenario. This is because a single set of parameters does not fit all scenarios, leading to the necessity of computing scenario-specific parameters. The parameters found are shown separately for each scenario;

Table 3.1: PID Values for Steering and Throttle in First Scenario

	Scenario I	
	Steering	Throttle
P	1	1
I	0.1	0.05
D	0.5	2

Table 3.2: PID Values for Steering and Throttle in Second Scenario

	Scenario II	
	Steering	Throttle
P	1	0.5
I	0.1	0.5
D	0.5	0.05

3.2 NMPC Path Tracking

In this application, the primary goal is to move the vehicle along a predetermined trajectory and optimize this process. For this purpose, we first need a reference. Unlike the PID application, this reference is obtained by manually driving the vehicle in the Carla simulator environment. To create two different scenarios, data has been collected by requiring the vehicle to perform different maneuvers from two different starting points. The gathered references are made suitable through a reference generator before being fed to the NMPC controller. This process essentially involves segmenting the reference trajectory according to the reference speed. After this process, the generated reference and the vehicle's real-time data are provided to the controller for use in NMPC. The NMPC controller uses the single-track model as the vehicle model. Hence, our NMPC controller produces two outputs. The first is the steering angle, δ_f. The other output, ax, is transformed into throttle and brake inputs and given to the Carla environment. All the blocks used in this application will be explained in detail.

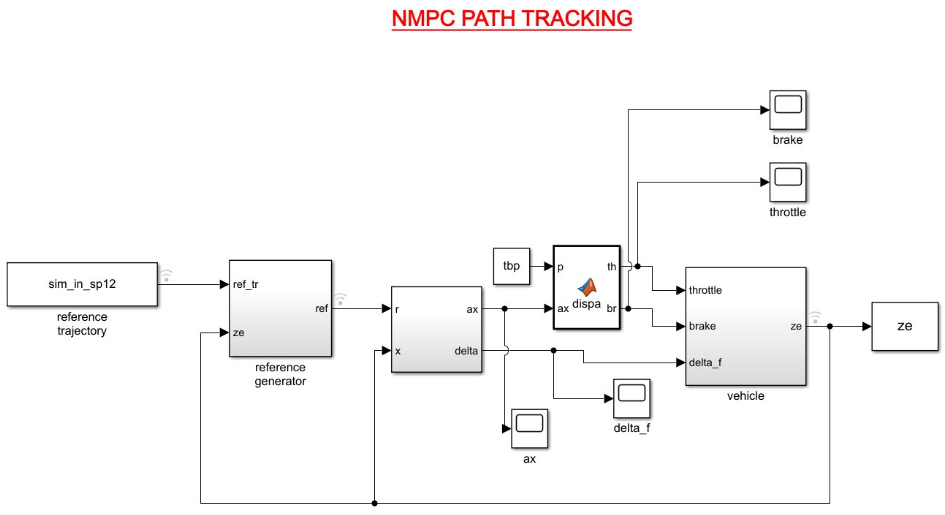

Figure 3.6: NMPC Control Simulink Model

3.2.1 NMPC Control

Model Predictive Control (MPC) is highly regarded for its versatility and effectiveness in a broad spectrum of industrial and technological settings. This is largely due to its proficiency in crafting control strategies for systems with multiple variables, accommodating constraints related to states, inputs, and outputs. To address the challenges posed by nonlinear dynamics, constraints, and non-convex performance indices, Nonlinear Model Predictive Control (NMPC) strategies have been developed [36], [37]. The NMPC method represents a significant challenge in its industrial application due to the difficulties in ensuring an adequately optimal solution to the optimization problem within the constraints of real-time requirements. Research into nonlinear optimal control began in the 1950s and 1960s, quickly leading to pivotal developments such as the maximum principle and dynamic programming. The practical implementation of Nonlinear Model Predictive Control (NMPC), particularly in optimal control, greatly simplifies the control design process for large and complex systems, positioning it as a compelling choice among various alternatives. However, this nonlinear control approach faces numerous obstacles, including implementation challenges in real-time embedded controllers, the application of nonlinear dynamic systems, and the estimation of states. The nonlinear nature of the problem may necessitate a substantial number of computations at each sampling moment due to the potential for multiple local minimum solutions, without ensuring the attainment of the best possible optimal solution. Consequently, NMPC requires the iterative resolution of an optimal control problem at each sampling instant in a receding horizon manner. Regrettably, there's no assurance that this receding horizon strategy of implementing a series of open-loop optimal control solutions will perform effectively or remain stable when applied to the closed-loop system. Nevertheless, in recent years, advancements in nonlinear optimization algorithms have led to the development of efficient NMPC implementations suitable for real-time applications (see, e.g., [38, 39, 40, 41]).

Delving further into the details, Nonlinear model predictive control (NMPC) represents a versatile and comprehensive method for controlling nonlinear systems. The methodology is for every time interval:

- A prediction is made for a specified period which is called prediction horizon using the system's model.
- The control input is selected based on the outcome that most closely aligns with the target performance which is called 'best prediction', achieved through an online optimization process.

NMPC enables the handling of constraints on inputs, states, and outputs, while systematically managing the balance between performance and control effort.

When we compare NMPC with Linear Quadratic Regulator (LQR) control, NMPC represents a nonlinear, finite-horizon variant of the LQR control.

NMPC can be used in automotive and aerospace systems, chemical operations, robotics, medical devices, and more.

When we consider the MIMO nonlinear system:

$$\dot{x} = f(x, u) \tag{3.2}$$

$$y = h(x, u) \tag{3.3}$$

where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^{n_u}$ is the command input and $y \in \mathbb{R}^{n_y}$ is the output.

Extending this concept to time-varying systems is direct and uncomplicated.

Assume the state is monitored in real time, using a sampling interval T_s. The measurements are:

$$x(t_k), \quad t_k = T_s k, \quad k = 0, 1, 2, \ldots$$

If the state isn't directly measured, the use of an observer or a model in the input-output format is required.

NMPC relies fundamentally on two principal actions: prediction and optimization.

At each time step $t = t_k$, predictions for the system's state and output are made for the interval $[t, t + T_p]$

- The prediction is derived through the process of integration for the *(Formula 3.2)*.
- $T_p \geq T_s$ is called the prediction horizon.

At any given moment τ within the interval $[t, t+T_p]$, the predicted output $\hat{y}(\tau)$ is determined by the 'initial' state $x(t)$ and the input signal:

$$\hat{y}(\tau) \equiv \hat{y}(x(t), u(t:\tau))$$

where $u(t:\tau)$ denotes a generic input signal in the interval $[t, \tau]$.

Within the time frame $[t, t+T_p]$, $u(\tau)$ acts as an open-loop input, meaning it does not rely on $x(\tau)$.

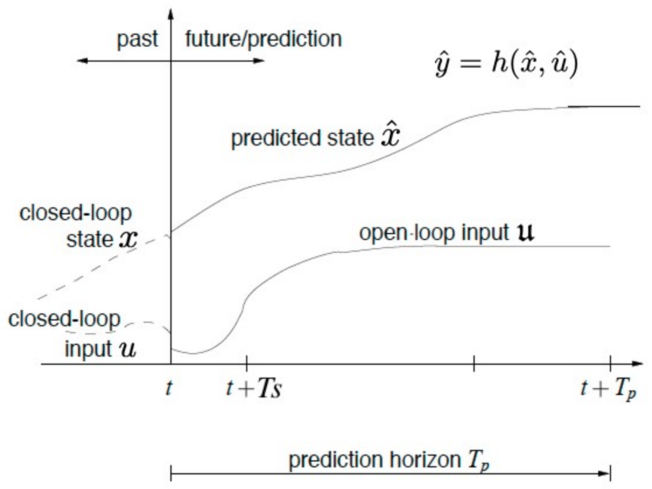

Figure 3.7: NMPC Control Parameters

At every moment $t = t_k$, we search for an input signal $u(t:\tau) = u^*(t:\tau)$, aiming for a prediction that

$$\hat{y}(x(t), u^*(t:\tau)) \equiv \hat{y}(u^*(t:\tau)) \tag{3.4}$$

matches the desired behavior within the interval $\tau \in [t, t+T_p]$.

The notion of desired behavior is formalized through the definition of the objective function:

$$J(u(t:t+T_p)) = \int_t^{t+T_p} \left(\|\hat{y}_p(\tau)\|_Q^2 + \|u(\tau)\|_R^2 \right) d\tau + \|\hat{y}_p(t+T_p)\|_P^2 \tag{3.5}$$

where $\hat{y}_p(\tau) = r(\tau) - \hat{y}(\tau)$ represents the predicted tracking error, with $r(\tau) \in \mathbb{R}^{n_y}$ being a reference signal to follow. The notation $\|\cdot\|_X$ denotes weighted vector norms, and their integrations represent (square) signal norms.

The input signal $u^*(t : t+T_p)$ is chosen as one minimizing the objective function $J(u(t : t+T_p))$.

The objective is to minimize, at each time t_k, the square norm of the tracking error $\|\hat{y}_p(\tau)\|_Q^2 = \|r(\tau) - \hat{y}(\tau)\|_Q^2$ over a finite time period.

The term $\|\hat{y}_p(t+T_p)\|_P^2$ emphasizes the significance of the final tracking error. The term $\|u(\tau)\|_R^2$ enables the management of the trade-off between performance and command activity. The square weighted norm of a vector $v \in \mathbb{R}^n$ defined as;

$$\|v\|_Q^2 = v^T Q v = \sum_{i=1}^{n} q_i v_i^2, \quad Q = \operatorname{diag}(q_1, \dots, q_n) \in \mathbb{R}^{n\times n}, q_i \geq 0. \tag{3.6}$$

The tracking error $\hat{y}_p(\tau) = r(\tau) - \hat{y}(\tau)$ is dependent on $\hat{y}(\tau)$, which is derived through integration of first formula.

Therefore, the minimization of J is subject to constraints:

$$\dot{x}(\tau) = f(\hat{x}(\tau), u(\tau)), \quad \hat{x}(\tau) = x(t), \quad \tau \in [t, t+T_p] \tag{3.7}$$

$$\hat{y}(\tau) = h(\hat{x}(\tau), u(\tau)) \tag{3.8}$$

Additional constraints may also apply such as obstacles and collision avoidance;

- the predicted state/output: $\dot{x}(\tau) \in X_c$, $\hat{y}(t) \in Y_c$, $\tau \in [t, t+T_p]$
- The input such as saturation input: $u(\tau) \in U_c$, $\tau \in [t, t+T_p]$

At every time $t = t_k$, for $\tau \in [t, t+T_p]$ the following optimization problem is addressed:

$$u^*(t : t+T_p) = \arg\min_{u(.)} J(u(t : t+T_p)) \tag{3.9}$$

subject to:

$$\dot{\hat{x}}(\tau) = f(\hat{x}(\tau), u(\tau)), \tag{3.10}$$

$$\hat{x}(\tau) = x(t) \tag{3.11}$$

$$\hat{y}(\tau) = h(\hat{x}(\tau), u(\tau)) \tag{3.12}$$

$$\hat{x}(\tau) \in X_c, \quad \hat{y}(\tau) \in Y_c, \quad u(\tau) \in U_c \tag{3.13}$$

where T_s is the sampling time, T_p is the prediction horizon, with $0 \leq T_s \leq T_p$.

Second optimization problem *(Formula 3.11)*:

- is in general non-convex;
- must be solved on-line, at each time t_k.

J is dependent on the signal $u(\cdot)$. Given that a signal is a function of time, J represents a function of a function. This type of mathematical object is frequently referred to as a functional.

Efficient numerical algorithms can be employed to solve the *(Formula 3.11)*. There are no assurances of finding a global minimum; generally, they yield a local minimum. From the perspective of control performance, a local minimum can be deemed satisfactory.

When examining the receding horizon strategy:
Assume that, at time $t = t_k$, the optimal input signal $u^*(t : t + T_p)$ has been determined by solving the aforementioned optimization problem.

- $u^*(t : t + T_p)$ is an open-loop input: it is contingent on $x(t)$ but not on $x(\tau)$, for $\tau > t$.
- If $u^*(t : t + T_p)$ is applied throughout the entire interval $[t, t + T_p]$, it does not enact a feedback mechanism, thereby lacking the ability to enhance precision, mitigate errors and disturbances, or adjust to changing scenarios.

The NMPC feedback control algorithm is realized through the implementation of what is known as a receding horizon strategy:

1. At time $t = t_k$:
   (a) compute $u^*(t : t + T_p)$ by solving the *(Formula 3.11)*;
   (b) apply only the first input value: $u(\tau) = u^*(t = t_k)$ and keep it constant for $\forall\tau \in [t_k, t_{k+1}]$.
2. Repeat steps 1a-1b for $t = t_{k+1}, t_{k+2}, \ldots$

When examining the closed-loop scheme:
Plant: $\dot{x} = f(x, u), \quad y = h(x, u)$

NMPC: on-line solution of the *(Formula 3.11)* and receding horizon strategy.

- The NMPC algorithm incorporates a plant model, which is utilized for making predictions.

- The prediction model takes the form $\dot{\hat{x}}(\tau) = f(\hat{x}(\tau), u), \quad \hat{y} = h(\hat{x}(\tau), u)$, where $f \approx f$, and $h \approx h$.
- Simplified models (f, h) are frequently employed.
- In the nominal case, $f = f$, $h = h$.

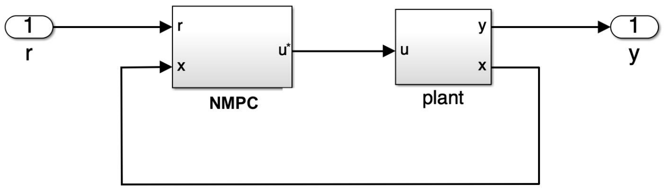

Figure 3.8: NMPC Control Scheme

When examining the choosing of the parameters:

- T_s: In numerous scenarios, the sampling time is predetermined and not selectable. If it is possible to choose, a trial and error method in simulation may be utilized, bearing in mind that T_s ought to be:
  - sufficiently small to effectively manage the plant dynamics, as per the Nyquist-Shannon sampling theorem;
  - not excessively small, to prevent numerical issues and slow down computation.
- T_p: It can be selected via a trial and error process in simulation, taking into account that
  - a 'large' T_p enhances the stability properties of the closed-loop system;
  - a 'too large' T_p might diminish the accuracy of short-term tracking.
- Choosing Q, R, P values can be similar to the parameters in LQR/LQRY.

Initial selection: Assuming that all variables exhibit similar ranges of variation, Q, R, and P can be selected as diagonal, non-negative matrices, with:

$$Q_{ii} = \begin{cases} > 0 & \text{when there are specific requirements on } y_i \\ = 0 & \text{otherwise} \end{cases}$$

$$P_{ii} = \begin{cases} > 0 & \text{when there are specific requirements on } y_i \\ = 0 & \text{otherwise} \end{cases}$$

$$R_{ii} = \begin{cases} > 0 & \text{when there are specific requirements on } u_i \\ \approx 0 & \text{otherwise} \end{cases}$$

Trial and error (in simulation): Adjust the values of Q_{ii}, R_{ii}, and P_{ii} until the requirements are met.

Action		Effect		Purpose
increasing Q_{ii}, P_{ii}	$\rightarrow$	decreasing the energy of x_i, y_i	$\rightarrow$	to minimize oscillations and shorten convergence time.
increasing R_{ii}	$\rightarrow$	decreasing the energy of u_i	$\rightarrow$	to decrease command effort and 'energy consumption

When examining the advantages and disadvantages of NMPC controller;

Advantages:

- General and flexible, suitable for complex MIMO (Multiple Input Multiple Output) systems.
- The formulation is intuitive, grounded in concepts of optimality.
- Incorporates constraints and input saturation, with the capacity for these elements to vary over time.
- Effectively handles the trade-off between performance and input activity.
- Determines optimal trajectories over a finite time period.
- Provides a unified approach for computing both the optimal trajectory and the control law.

Disadvantages:

- Involves a high computational cost for online processing.
- The optimization problem may encounter local minima.
- Faces issues with unstable zero-dynamics, similar to all methods.

3.2.2 Finding The Closest Point In The Road

This code is designed to generate reference positions and orientations for a vehicle to follow a pre-collected reference path (x and y position information), using the vehicle's current position and speed. The function is intended to be used within a

MATLAB Function block in Simulink. Let's explain its functionality step by step:

Inputs:

- $\mathbf{N_p}$: A parameter determining the number of reference points the vehicle is expected to follow over a future time interval, calculated based on the reference speed.
- **ref_tr**: The pre-collected reference path, a matrix where each row contains the x and y positions of a reference point.
- $\mathbf{z_e}$: A vector containing the current pose of the vehicle (x, y positions, and orientation).

Finding the Closest Point on the Reference Trajectory:

- The function first calculates the distance between the vehicle's current position and each point on the reference path. This is done by taking the vector norm of the difference between the vehicle's current x and y coordinates and the x and y coordinates of the points on the reference path.
- It then finds the smallest of these distances (i.e., the closest reference path point to the vehicle) and obtains the index (c) and coordinates (ref_pose) of this point on the reference trajectory.

Selecting the Relevant Section of the Reference Trajectory:

- The vehicle is expected to follow the next N_p points on the reference path, according to its reference speed. This represents a path forward from the vehicle's current position, covering N_p points on the reference trajectory.
- For this, the `linspace` function is used to select a total of $NS + 1$ evenly spaced points between the current closest point (c) and $c + N_p - 1$ (ir). Here, NS is a fixed number (for example, 50), which allows for a smoother sampling of the reference path.

Creating the Reference Vector:

- The x and y position information of these selected points (ref_XY0) is then taken, and this information is reshaped and transferred into the output vector ref. The `reshape` function is used to turn the ref_XY0 matrix into a single column vector, which is provided as a reference to the Simulink model.

This function is used to determine the reference path and orientation for a vehicle to follow at a given speed over a specified time interval.

```matlab
function [ref, ref_pose] = ref_gen(Np, ref_tr, ze)

N = size(ref_tr, 1);
pose = ze(1:3);

% Point of the reference trajectory closest to the vehicle.
dis = vecnorm(ref_tr(:,1:2)' - pose(1:2) * ones(1, N));
[~, c] = min(dis);
ref_pose = ref_tr(c,:)';

% Portion of the reference trajectory corresponding to
% the time interval [t, t+Tp] at the speed vx_ref.
NS = 50;
ir = round(linspace(c, c + Np - 1, NS + 1));
ref_XY0 = ref_tr(ir(1:NS), 1:2)';
ref = reshape(ref_XY0, 2 * NS, 1);

end
```

3.2.3 Creating Carla Environment

The Matlab S-function block was used to create the Carla environment. As inputs, it receives the throttle, brake, and δ_f commands from the NMPC controller. The outputs include x position, y position, yaw angle, x velocity, y velocity, and yaw rate. Initially, the vehicle is spawned at the same starting point as the reference. Then, control commands for throttle, brake, and δ_f are provided. The vehicle will move according to these commands. After the vehicle has moved, the data to be collected for z_e is obtained, and a z_e matrix of size Nx6 is created and outputted. At the end of the simulation, the vehicle is removed from the map.

```matlab
classdef Carla_enviroment_nmpc < matlab.System
    % Carla Enviroment for Lateral Control

    % Public, tunable properties
    properties
        steeringangle_input=0;
        throttle_input = 0;
        brake_input = 0;
    end

```

```
     properties(DiscreteState)

    end

    % Pre-computed constants
     properties(Access = private)
         car;
    end

    methods(Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing constants
            port = int16(2000);
            client = py.carla.Client('localhost', port);
            client.set_timeout(10.0);
            world = client.get_world();

            % Spawn Vehicle
            blueprint_library = world.get_blueprint_library();
            car_list = py.list(blueprint_library.filter("leon"));
            car_bp = car_list{1};
            spawn_point = py.random.choice(world.get_map().get_spawn_points());


        spawn_point.location.x = -113.648;
         spawn_point.location.y = -14.281;
        spawn_point.location.z = 0.6;
        spawn_point.rotation.yaw = 90;

            obj.car = world.spawn_actor(car_bp, spawn_point); %% spawn the car
            obj.car.set_autopilot(false)


        end

        function [ze] = stepImpl(obj,steeringangle_input,throttle_input,brake_input)

            vehicle_transform = obj.car.get_transform();
            orientation = vehicle_transform.rotation;

            x_position = obj.car.get_location().x;
            y_position = obj.car.get_location().y;
            x_velocity = obj.car.get_velocity().x;
            y_velocity = obj.car.get_velocity().y;
            w = obj.car.get_angular_velocity().z*pi/180;



    control = obj.car.get_control();
     control.steer = rad2deg(steeringangle_input)/70;
            control.throttle = throttle_input;
            control.brake = brake_input;
            yaw_angle = double(deg2rad(orientation.yaw));

            ze = [x_position,y_position,yaw_angle,x_velocity,
    y_velocity,w]';

            obj.car.apply_control(control);

        end

        function [distance] = isOutputComplexImpl(~)
            distance = false;

        end

        function [distance] = getOutputSizeImpl(~)
            distance = [6,1];


        end

        function [distance] = getOutputDataTypeImpl(~)
            distance = 'double';


        end

        function [distance] = isOutputFixedSizeImpl(~)
            distance = true;


        end

        function resetImpl(~)
            % Initialize / reset discrete-state properties
        end
    end

    methods(Access= public)
        function delete(obj)
            % Delete the car from the Carla world
            if ~isempty(obj.car)
                obj.car.destroy();
            end
        end
    end
end
```

3.2.4 NMPC Values

The most critical part of the control section is finding the correct values according to the model after it has been created. With the right parameters, the desired control can be achieved in the best way possible. For the NMPC method, control parameters were determined using only the vehicle model created in Simulink. Different reference δ_f and ax values were collected using a manual control method, and based on this reference, the initial NMPC parameters were identified through the Simulink model. Our vehicle model is similar to our model in Carla, but they do not match perfectly; therefore, it has been understood that better parameters can be obtained when used together with Carla. When readjusting the parameters, our expectation was to find parameters that work for every simulation and result in a faster vehicle. Thus, the best parameters obtained through trial and error are shown below;

Table 3.3: NMPC Values

	NMPC VALUES
R	$\begin{bmatrix} 0.1 & 0 \\ 0 & 1 \end{bmatrix}$
Q	$1000 \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
P	$100 \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Chapter 4

Simulation and Results

4.1 NMPC Path Tracking Results

Seeing the results of the simulation visually provides us with a general idea about the success of the application, but sometimes this can be misleading. In addition, the performance of the application can be interpreted differently by different people. Whether the outcome of the application is valid for us should depend on certain criteria. These criteria are called 'key performance indicators'. If the data we obtained on the simulation server meet these performance criteria, then our application is considered successful. In our application, the key performance criteria will be the cross-track error and heading error. Additionally, a comparison between the reference position and the vehicle's position will be made. However, making comparisons based solely on these two criteria can be misleading. This is because some controller values may show perfect performance in a specific scenario while showing undesirable performance in another scenario. Therefore, the most important key performance element is accuracy. Two different applications have been made using the NMPC (Nonlinear Model Predictive Control) controller. The reason for this is to demonstrate that NMPC parameters work effectively in every scenario. Based on these performance criteria, results for two different scenarios will be provided and interpreted in detail.

4.1.1 Scenario 1: Single Curve Long Straight Road

First, we must describe the scenario we will be conducting. It is crucial for the vehicle to perform both turning and maintaining a steady course on a lane smoothly, as these are the two most common maneuvers vehicles make. Moreover, to observe the controller's ability to perform these two movements consecutively and simultaneously, a scenario has been created where we first make a turn and then proceed along a lane. Carla Town10 has been used for this scenario. The reason for choosing this map is its inclusion of both main streets and side streets, requiring us to drive at different speeds. Additionally, since this map featuring a flat environment keeps the z-axis in the reference system constant, it ensures that the single-track model we used to create the vehicle model is more consistent. The most critical aspect of this scenario was ensuring the vehicle did not skid or depart from the lane while turning the curve and that it could adjust itself to the ideal speed and continue at a steady pace after completing the turn. To better understand the motion, some photos taken during the scenario have been shown in Figures 4.1-4.2.

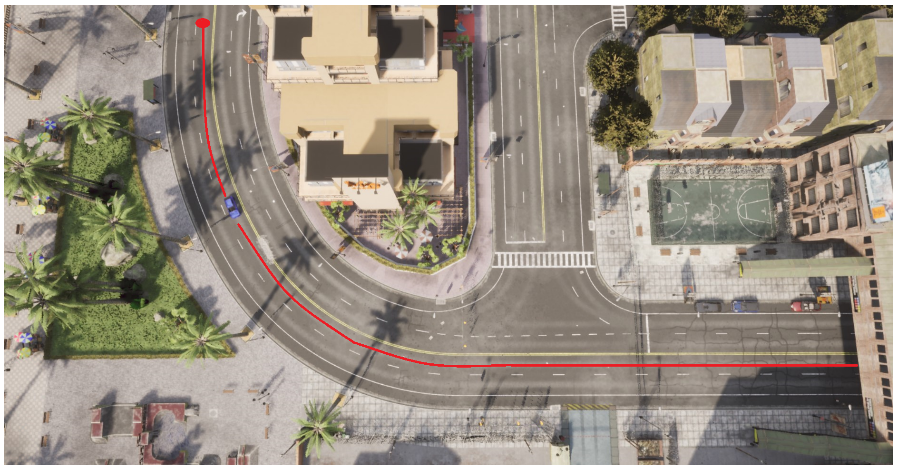

Figure 4.1: First Scenario Curve in Carla Simulator.

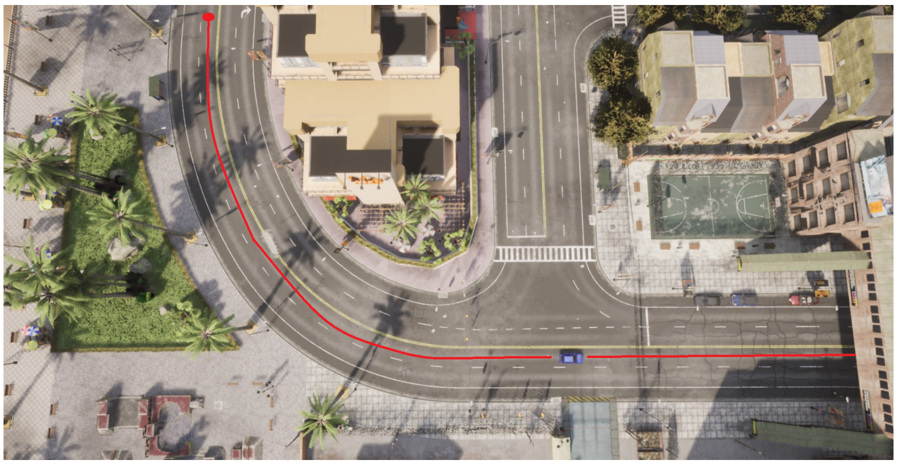

Figure 4.2: First Scenario Straight in Carla Simulator.

Secondly, we must examine our performance criteria. The most important of our performance criteria is the Cross-track error (e_{ct}). This error can briefly be described as the displacement from the center of the vehicle's front axle to the closest point on the trajectory. Another significant performance criterion is the heading error (e_h), which can be explained as the angle between the vehicle's longitudinal axis and the reference direction.

Seeing the vehicle's movement within a reference system is crucial. This provides us with information on how much the vehicle deviates from the reference system. Therefore, data collected using the vehicle's autonomous mode have been used as a reference. This reference represents an ideal driving scenario. By placing this reference drive into our coordinate system and overlaying the vehicle's position data obtained as a result of NMPC (Nonlinear Model Predictive Control) control, the vehicle's movement relative to the reference trajectory can be observed in Figure 4.3.

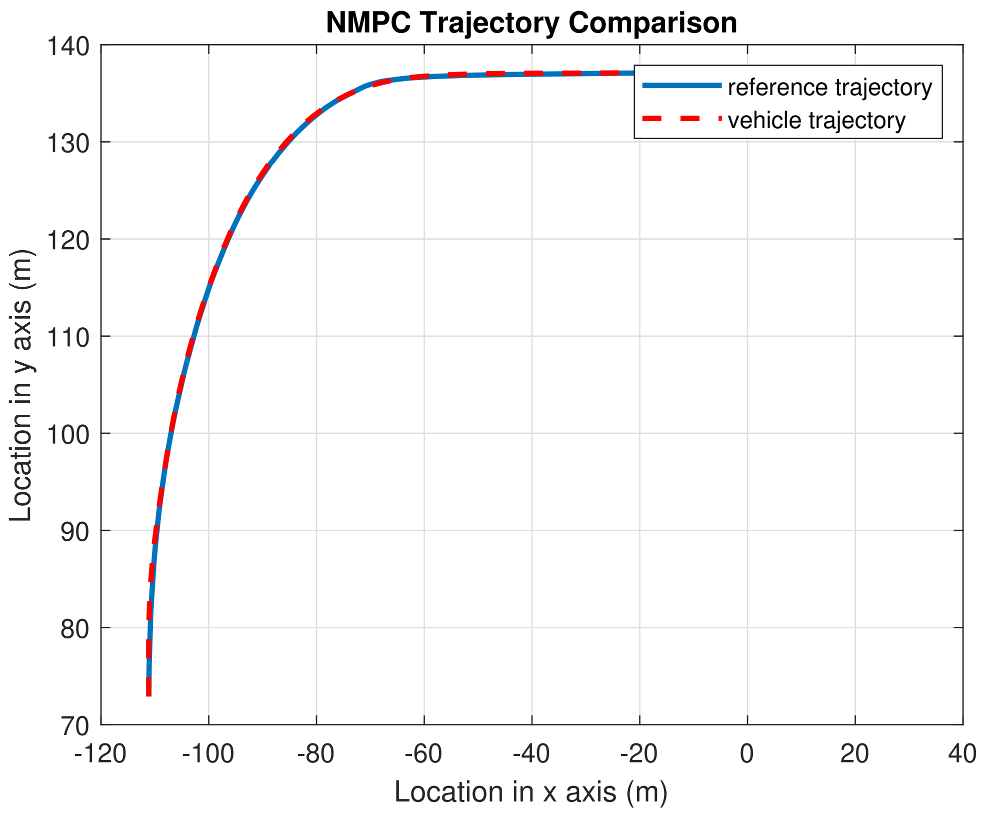

Figure 4.3: NMPC Trajectory Comparison

When examining the overall situation, the vehicle appears to follow the reference trajectory well. Given the simulation duration is long and the trajectory is extensive, such an overview can be misleading. To analyze the movement in detail, we can divide the motion into two parts. First, we can separate the turning motion interval [(-111,77)-(-60,136)] and the interval where the vehicle travels in a straight lane [(-60,136)-(-20,136)]. We can examine our first interval in Figure 4.4.

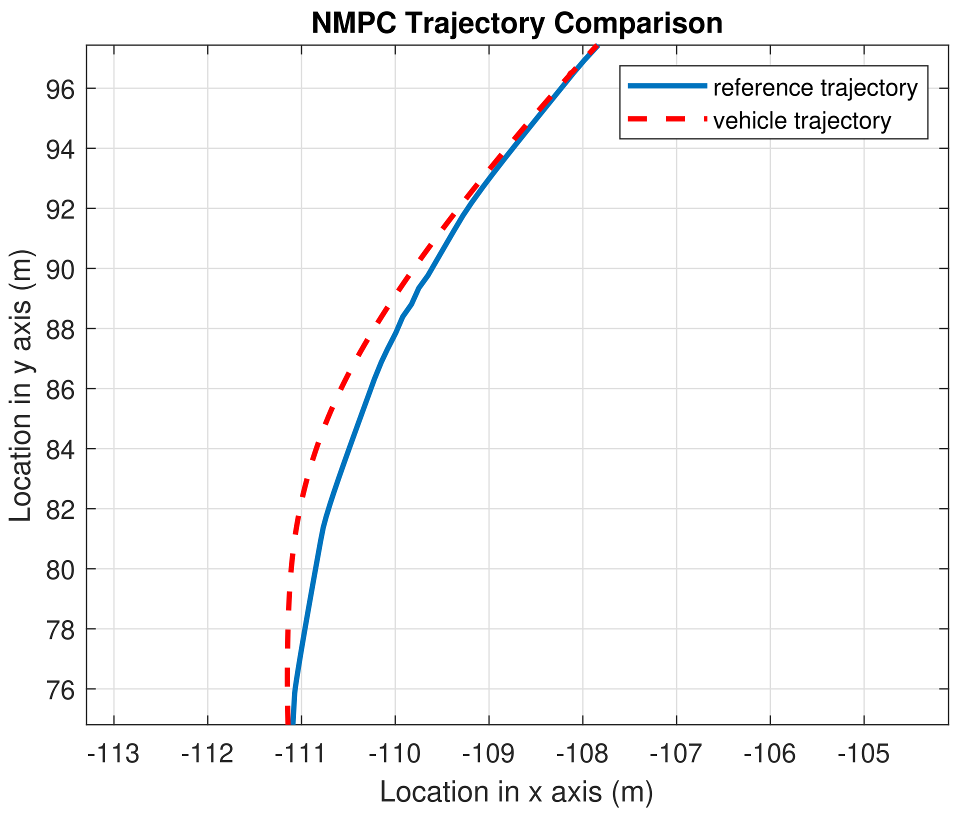

Figure 4.4: NMPC Trajectory Comparison in Curve

The issue of the vehicle missing the reference trajectory, especially at the beginning of its turning movement, is observed when it curves. Naturally, this reflects on the other error rates. The reason for this error is that in autonomous mode, the vehicle follows a sharper reference trajectory, while our controller delivers a smoother performance. Additionally, in autonomous mode, the vehicle responds more aggressively to significant changes in the y-axis in Carla. This explains the sharp reference value on the curve. The second interval is shown in Figure 4.5.

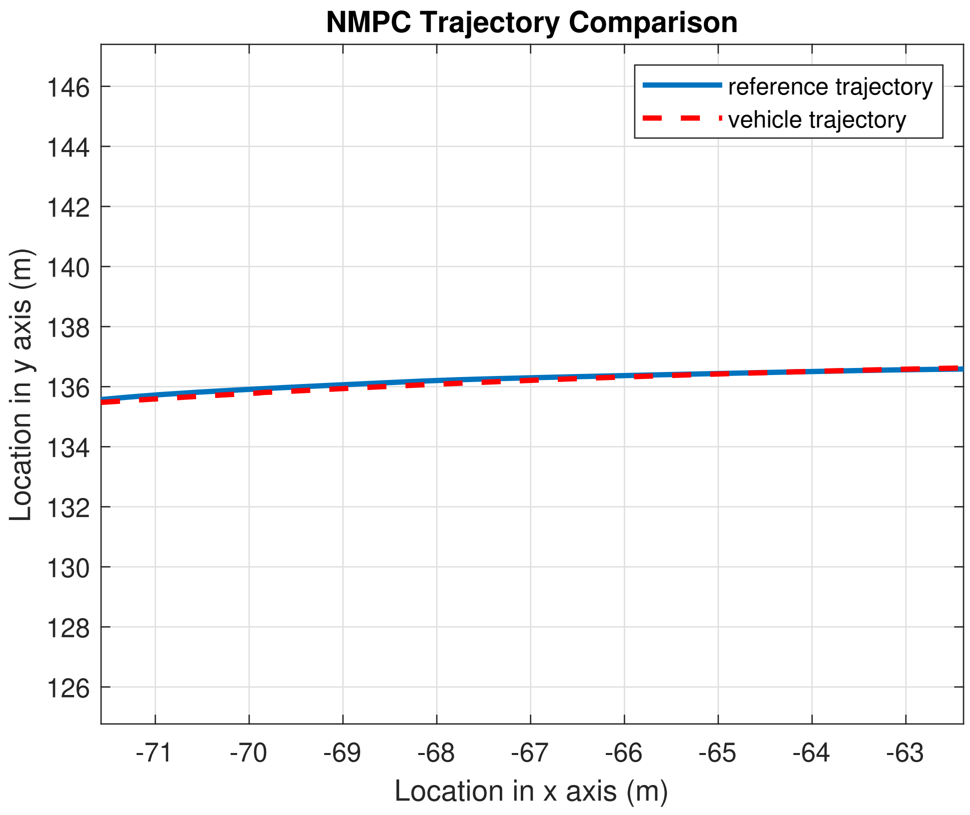

Figure 4.5: NMPC Trajectory Comparison in Straight Road

It is observed that the vehicle misses the reference trajectory especially at the beginning of the turning motion while taking the curve. Of course, this reflects on the other error rates. The reason for this error is that while the vehicle in autonomous mode draws a sharper reference trajectory, we receive a smoother performance from our controller. Additionally, the vehicle in Carla's autonomous mode responds more sharply to large changes in the y-axis. This explains the sharp reference value on the curve. If we examine the second interval;

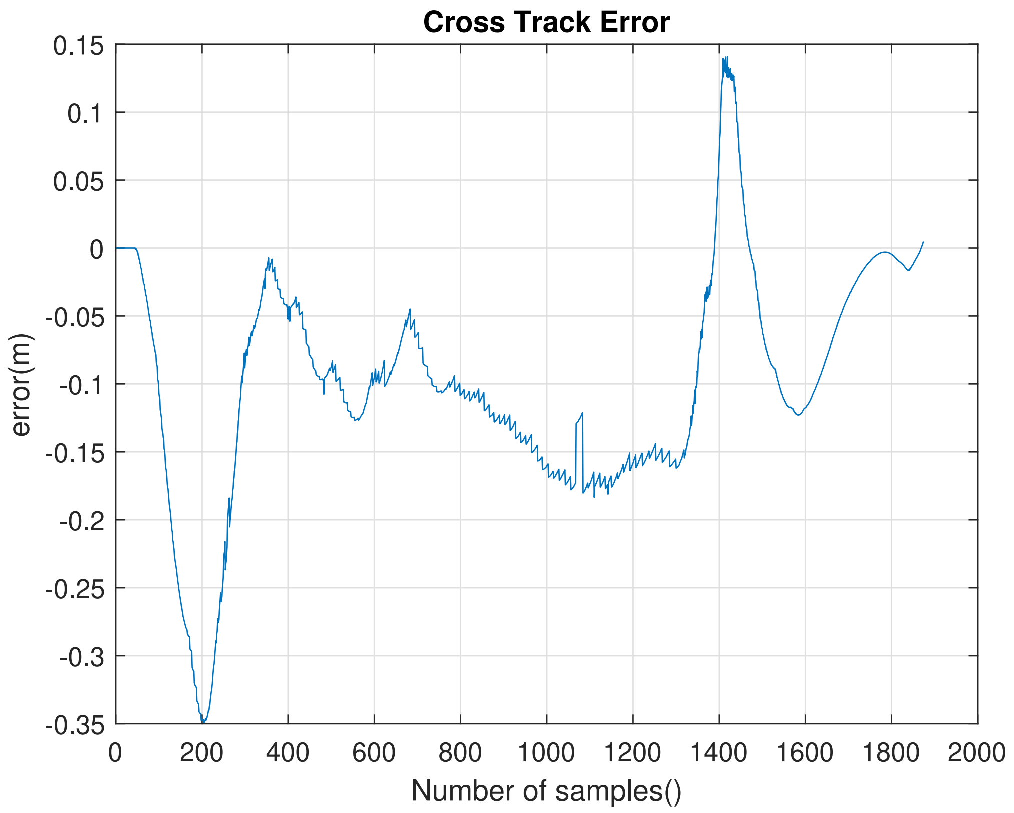

Figure 4.6: Cross Track Error

The vehicle's cross-track error (CTE), displayed in Figure 4.6, is observed to reach its highest value at the beginning. This is attributed to the vehicle creating a transient state initially, resulting in a deviation from the reference during the initial inertia. This value at the start of the curve has been compensated throughout the curve. At the end of the curve, the movement has changed, leading to a relatively high value at the start of the other condition, but this value has been compensated along the lane as well. When looking at the overall situation, the deviation values are very small relative to the reference, and the vehicle's performance can be considered very good.

The illustration of the heading error is provided in Figure 4.7. Similar to the cross-track error, it's observed that the vehicle initially deviates due to a transient state and then commits an error as its direction quickly changes at the start of the curve. Of course, these values are in very small amounts. Likewise, there has been a deviation during the transition to the second condition, which was subsequently compensated for, resulting in a satisfactory outcome.

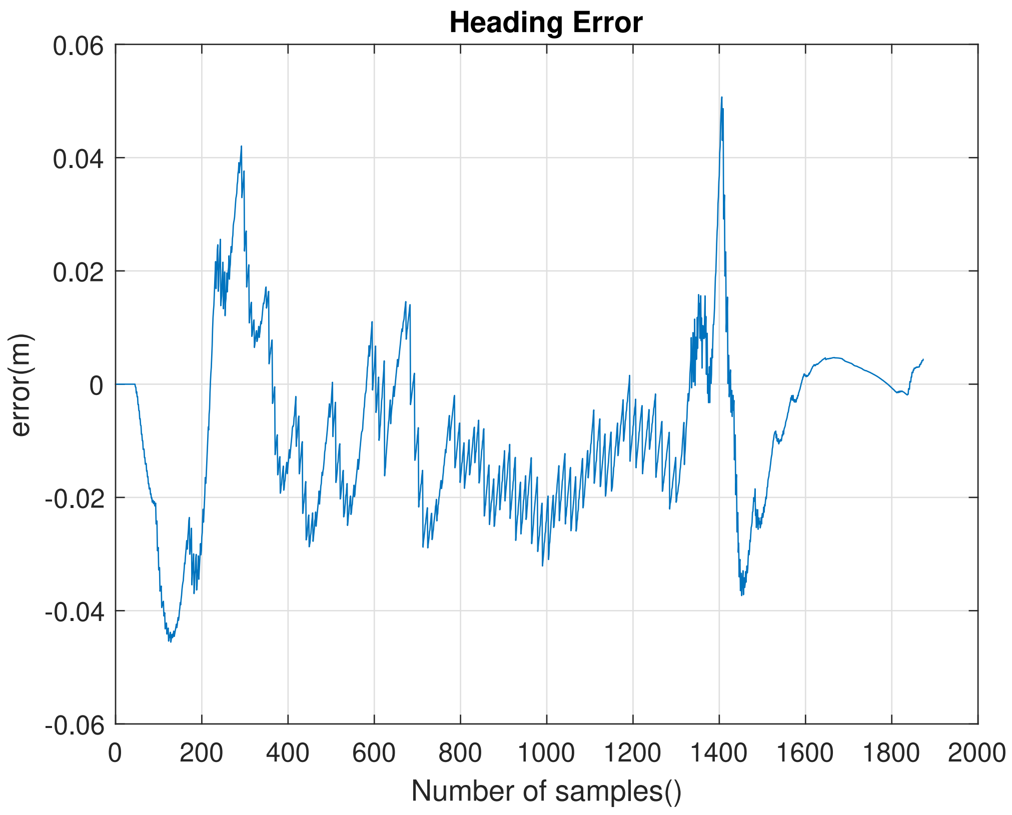

Figure 4.7: Heading Error

When we want to examine these errors numerically, we need to calculate the RMS (Root Mean Square) value. The RMS value helps us understand the average performance or effect of variable magnitudes, which is a fundamental tool for a wide range of applications and analyses. Our calculated RMS values and Max values for this scenario are presented in Tables 4.1 and 4.2.

Parameter	**RMS Value**
RMS(E_{ct})	0.1292
RMS(E_h)	0.0174

Table 4.1: RMS Values of E_{ct} and E_h in NMPC Control First Scenario.

Parameter	MAX Value
MAX(E_{ct})	0.3491
MAX(E_h)	0.045

Table 4.2: MAX Values of E_{ct} and E_h in NMPC Control First Scenario.

4.1.2 Scenario 2: Double Curve Long Straight Road

In the previous scenario, we examined how the vehicle would proceed in a straight lane following a turning motion. This scenario will investigate the vehicle's performance in a scenario involving two consecutive turns. Therefore, Carla Town10 was used, and this time, a simulation environment was prepared for the vehicle to navigate through a neighborhood. The vehicle's performance in executing two consecutive turning motions requires sharper outputs compared to a single smooth turn. The most crucial point in this scenario is for the vehicle to complete the second turn without skidding and while maintaining its speed after the first turn, and then to steadily increase its speed along the road after the turn is completed. Photos have been added to better understand the motion in Figures 4.8-4.9.

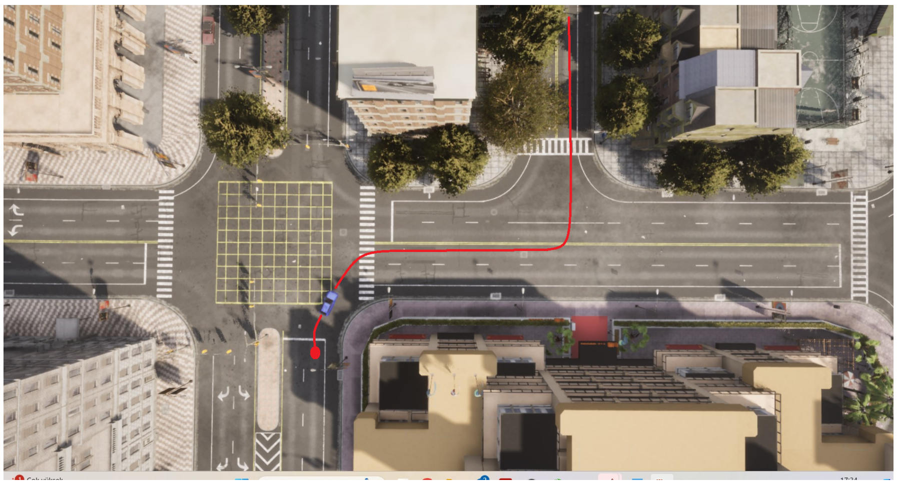

Figure 4.8: Second Scenario First Curve in Carla Simulator

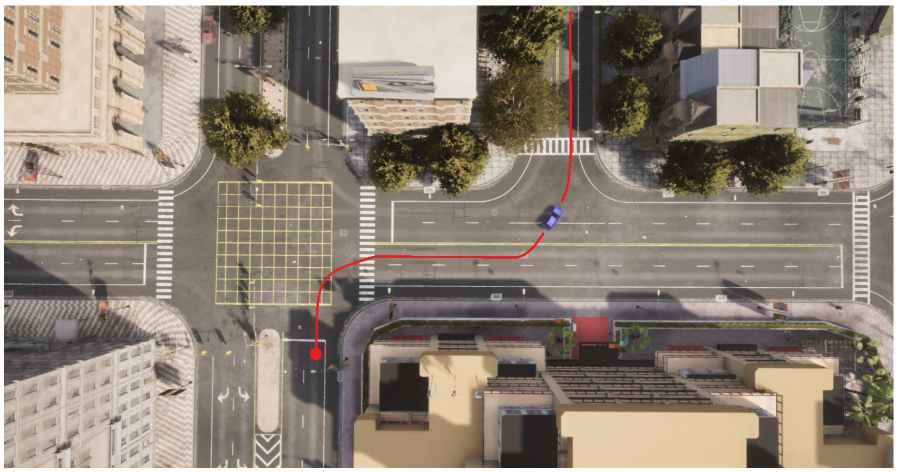

Figure 4.9: Second Scenario Second Curve in Carla Simulator

When examining the vehicle's deviation from the reference trajectory in Figure 4.10, it is observed that a satisfactory result has been achieved. The reason for this is that working with a more regular reference is due to the reference vehicle moving at a lower speed due to urban conditions. As seen, our controller has produced good results, almost achieving a complete match.

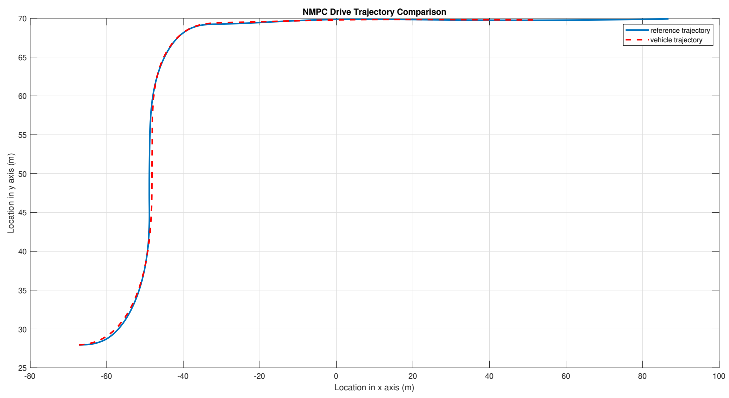

Figure 4.10: NMPC Trajectory Comparison

When we examine the cross-track error, depicted in Figure 4.11, generated based on these values, we encounter similarly good performance. Although our error amount increased due to the sudden changes we made during the turns, the error was compensated along the straight road afterwards.

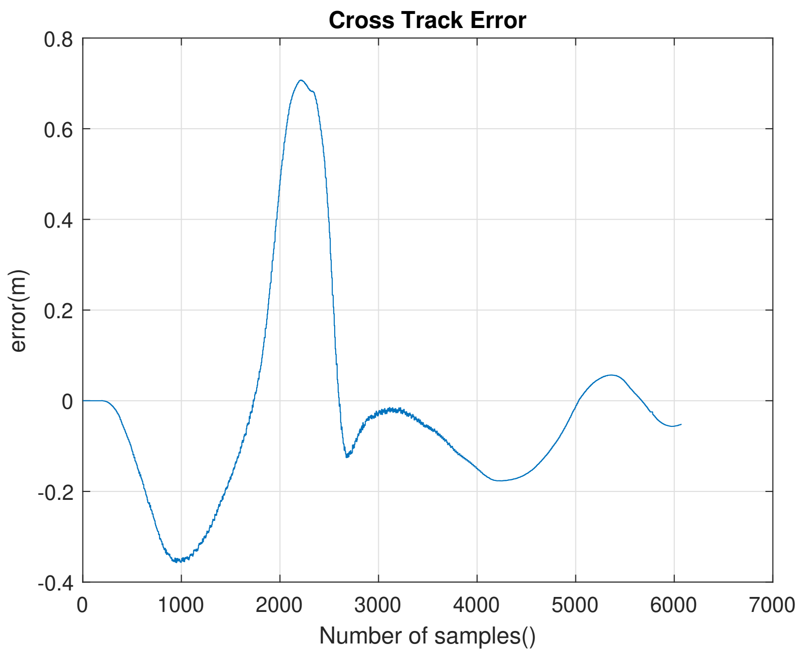

Figure 4.11: Cross Track Error

Figure 4.12 illustrates the heading error, which is a relatively more critical criterion in this scenario because the direction of the vehicle changes much more frequently. Consequently, the deviation amount is greater. When we examine the graph, we observe that the vehicle's deviation increases due to the directional changes in the curve, and then the error is compensated along the straight road.

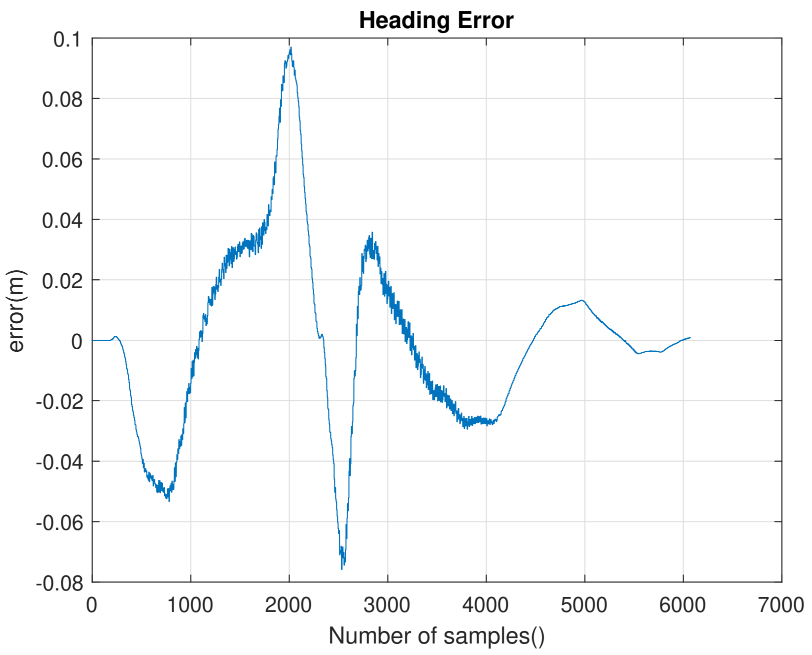

Figure 4.12: Heading Error

The data calculated on root-mean-square (RMS) and maximum errors are presented in Tables 4.3 to 4.4.

Parameter	**RMS Value**
RMS(E_{ct})	0.0851
RMS(E_h)	0.0488

Table 4.3: RMS Values of E_{ct} and E_h in NMPC Control Second Scenario.

Parameter	**MAX Value**
MAX(E_{ct})	0.7204
MAX(E_h)	0.091

Table 4.4: MAX Values of E_{ct} and E_h in NMPC Control Second Scenario.

4.2 PID Path Tracking Results

4.2.1 Scenario 1: Single Curve Long Straight Road

For the simulation conducted using PID control, similar to the NMPC Control, two scenarios will be considered. It is important to note that achieving comparable performance in both scenarios with fixed parameters in PID control is not feasible. As a result, we have fine-tuned the parameters to suit each scenario. Figure 4.13 shows the PID-controlled trajectory alongside the reference path for the first scenario. Upon examining the graph, the vehicle perfectly tracks the reference position. This is primarily because the PID controller directly minimizes the tracking error e_{ct}, resulting in minimal deviation.

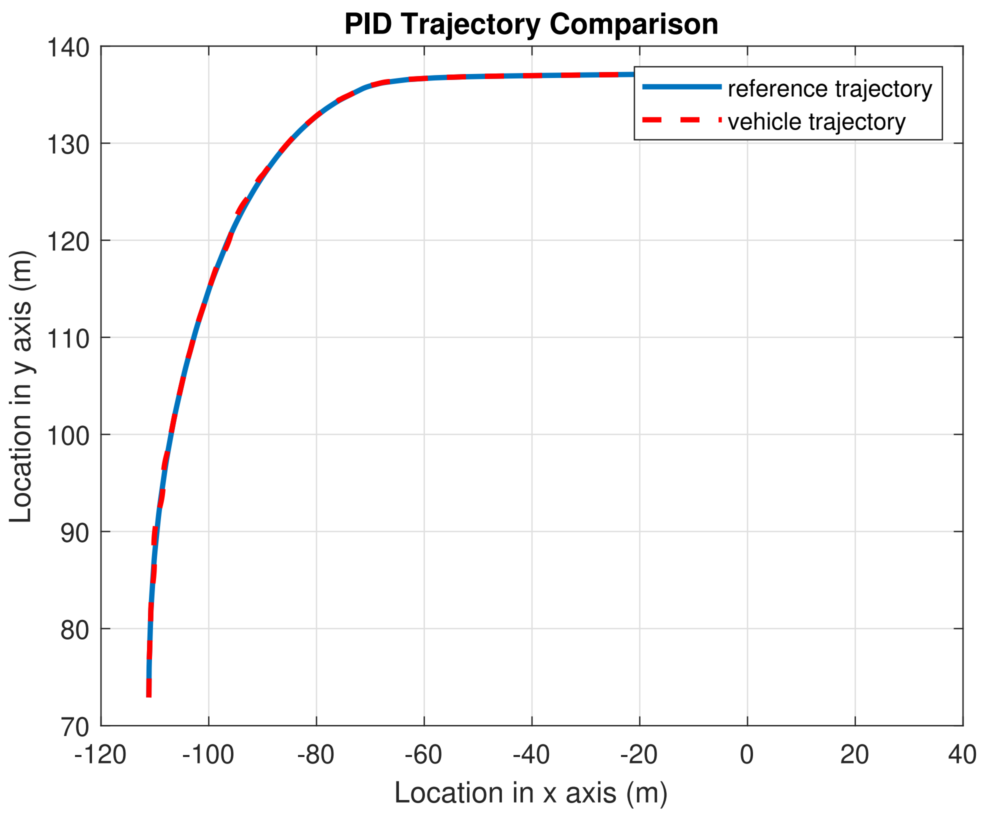

Figure 4.13: PID Control Trajectory Comparison

In Figure 4.14, the resulting cross-track error is presented. Although the value reaches high levels along the curve, it has been compensated afterward.

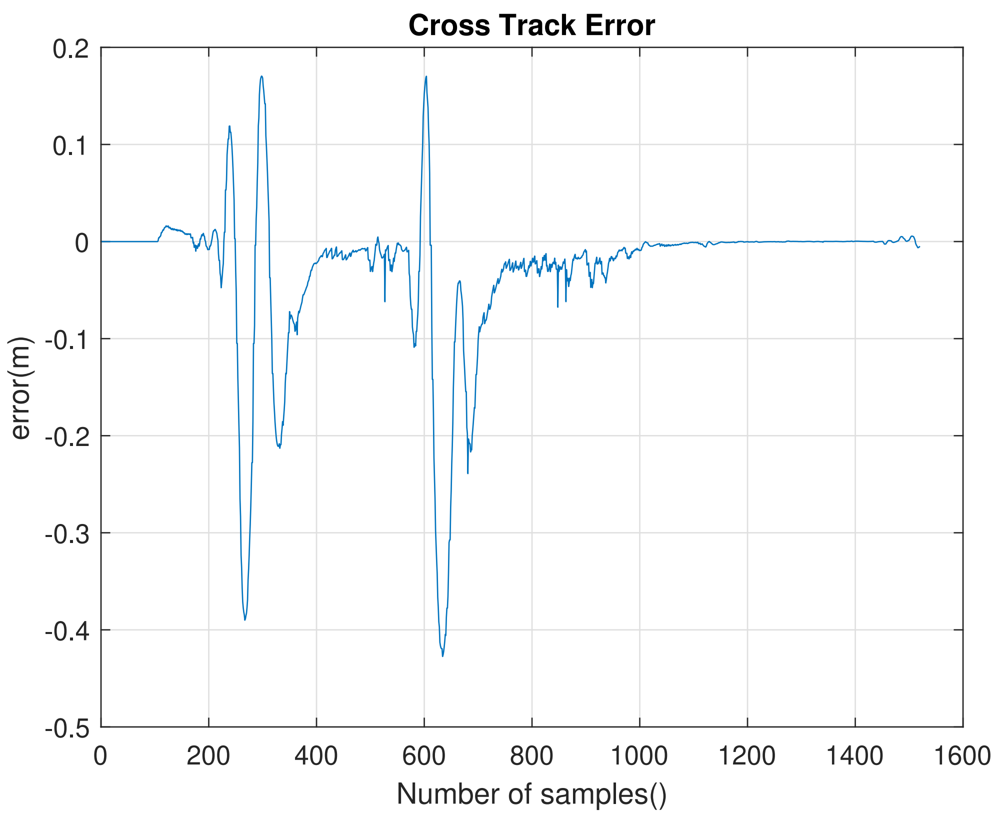

Figure 4.14: Cross Track Error

The heading error is shown in Figure 4.15. Initially, the vehicle exhibits some oscillation, but this quickly settles into stable behavior. Consequently, the heading error remains sufficiently low.

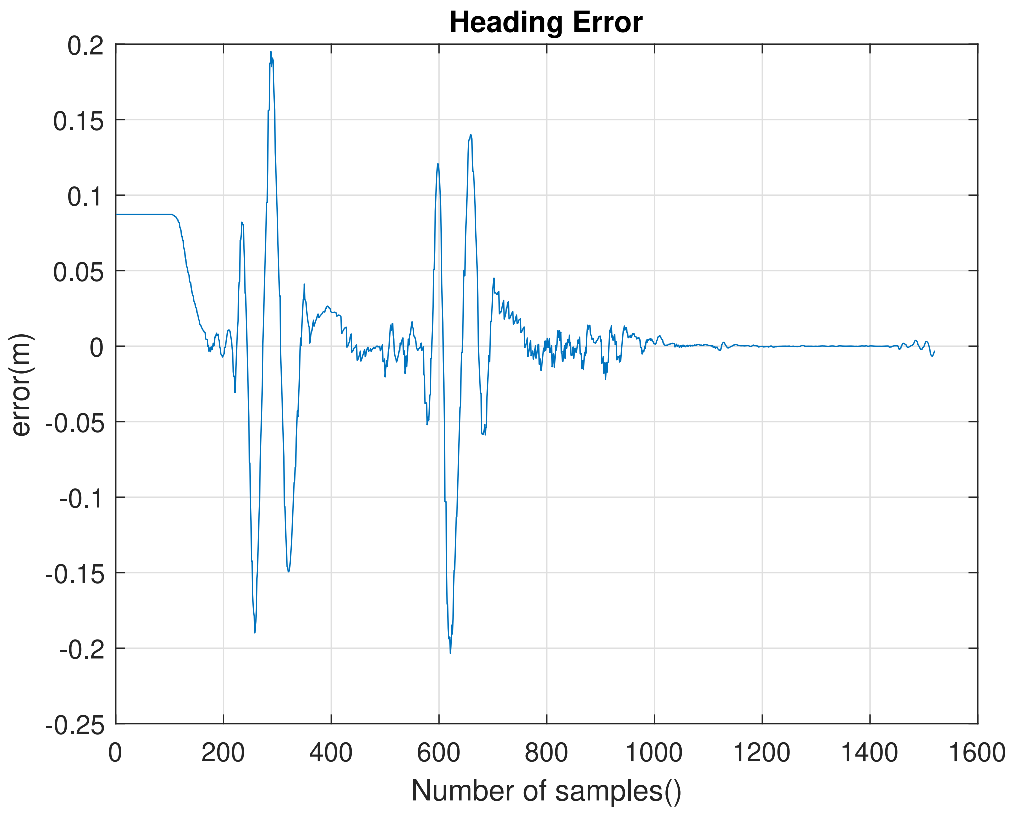

Figure 4.15: Heading Error

In Tables 4.5-4.6, the root-mean-square (RMS) and maximum errors are depicted.

Parameter	**RMS Value**
RMS(E_{ct})	0.0851
RMS(E_h)	0.0488

Table 4.5: RMS Values of E_{ct} and E_h in PID Control First Scenario.

Parameter	**MAX Value**
MAX(E_{ct})	0.3977
MAX(E_h)	0.2068

Table 4.6: MAX Values of E_{ct} and E_h in PID Control First Scenario.

4.2.2 Scenario 2: Double Curve Long Straight Road

In this case, similar to the first scenario, the reference used in the second scenario of NMPC control has been employed. With PID parameters specifically adjusted for this scenario, the results are as follows.

Figure 4.16 represents the resulting cross-track error. The cross-track error initially shows a low result at the first curve due to the vehicle's slow speed, then reaches high values for the second curve along with the acceleration on the straight section. Although a good result has been achieved overall, a significant value is observed on the second curve compared to our other control methods.

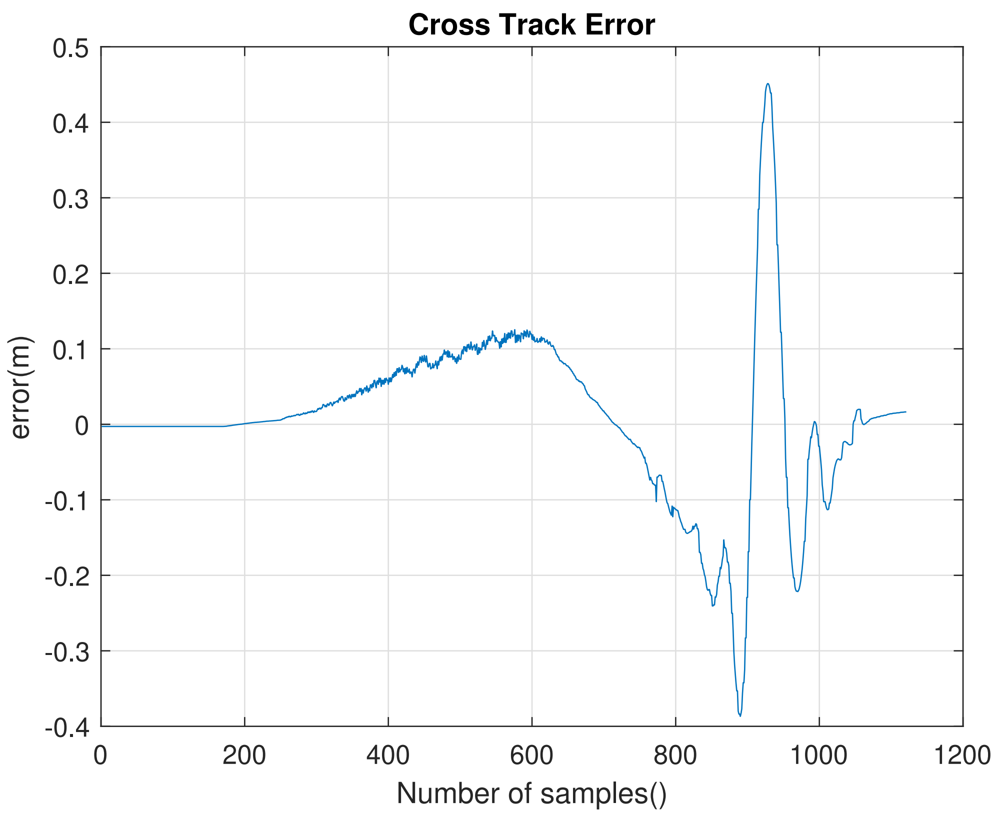

Figure 4.16: Cross Track Error

The heading error is shown in 4.17. The heading error presents a low deviation value up until the second curve, where the value slightly increases.

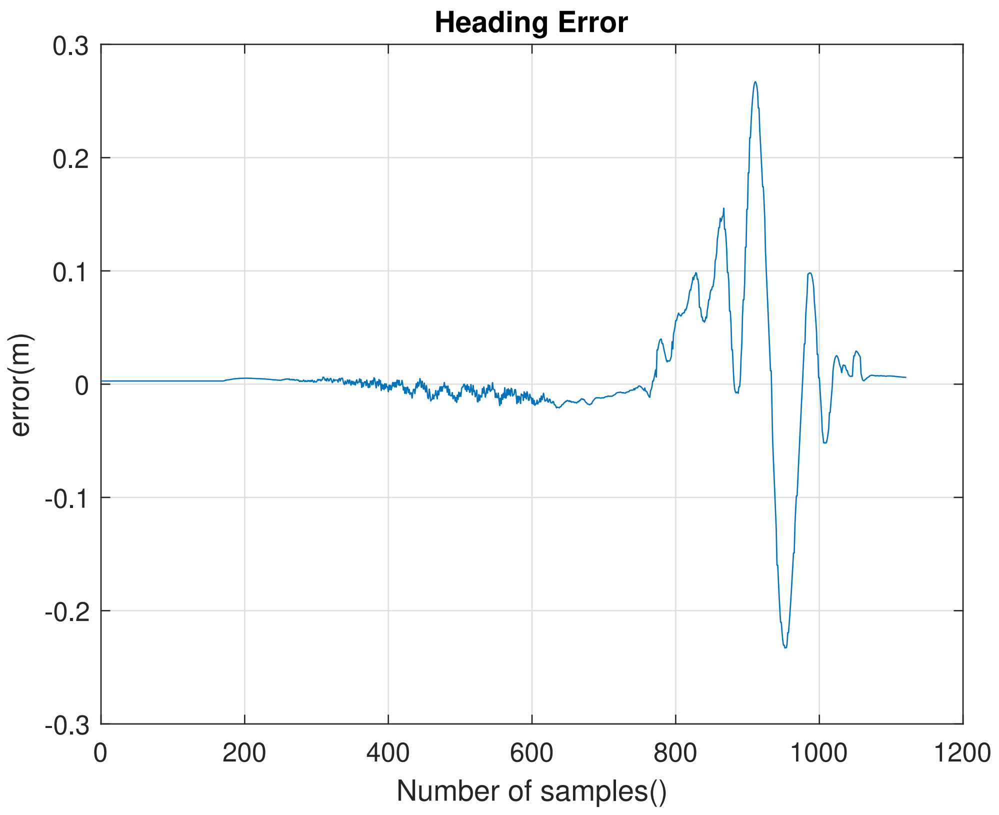

Figure 4.17: Heading Error

The root-mean-square (RMS) and maximum errors are illustrated in Tables 4.7-4.8.

Parameter	RMS Value
RMS(E_{ct})	0.1103
RMS(E_h)	0.0550

Table 4.7: RMS Values of E_{ct} and E_h in PID Control Second Scenario.

Parameter	MAX Value
MAX(E_{ct})	0.4513
MAX(E_h)	0.2670

Table 4.8: MAX Values of E_{ct} and E_h in PID Control Second Scenario.

4.3 Manual Path Tracking Results

The method of manually controlling a vehicle by a driver is currently the most common method. Undoubtedly, a vehicle operated by a human can create deviation and errors relative to an ideal reference. Therefore, one of the primary objectives of autonomous vehicles is to ensure safety; other control methods must perform at least as well as this method. We can briefly examine our performance criteria for manual control.

Figure 4.18 shows the comparison between the reference and the vehicle trajectories. It is observed that deviation from the reference occurs along the curve, and the reference is followed along the straight road after the curve. Our reference for this comparison, like the others, is Carla's autonomous driving mode.

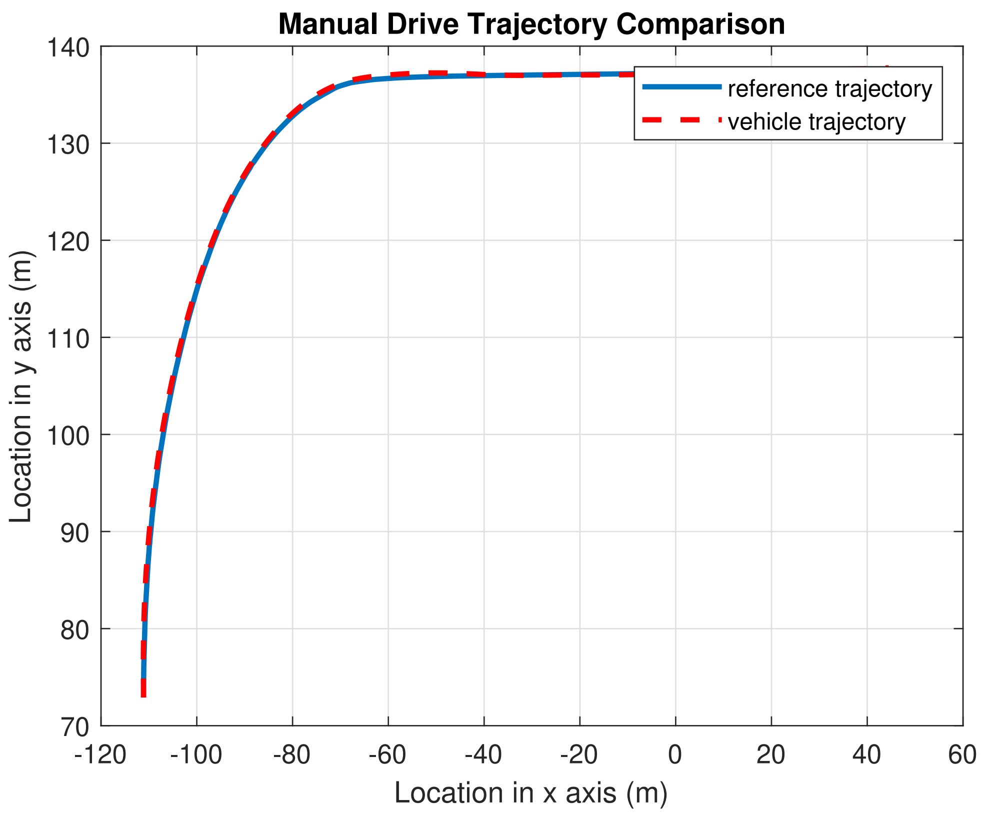

Figure 4.18: Manual Drive Trajectory Comparison

A visualization of the cross-track error is provided in Figure 4.19.

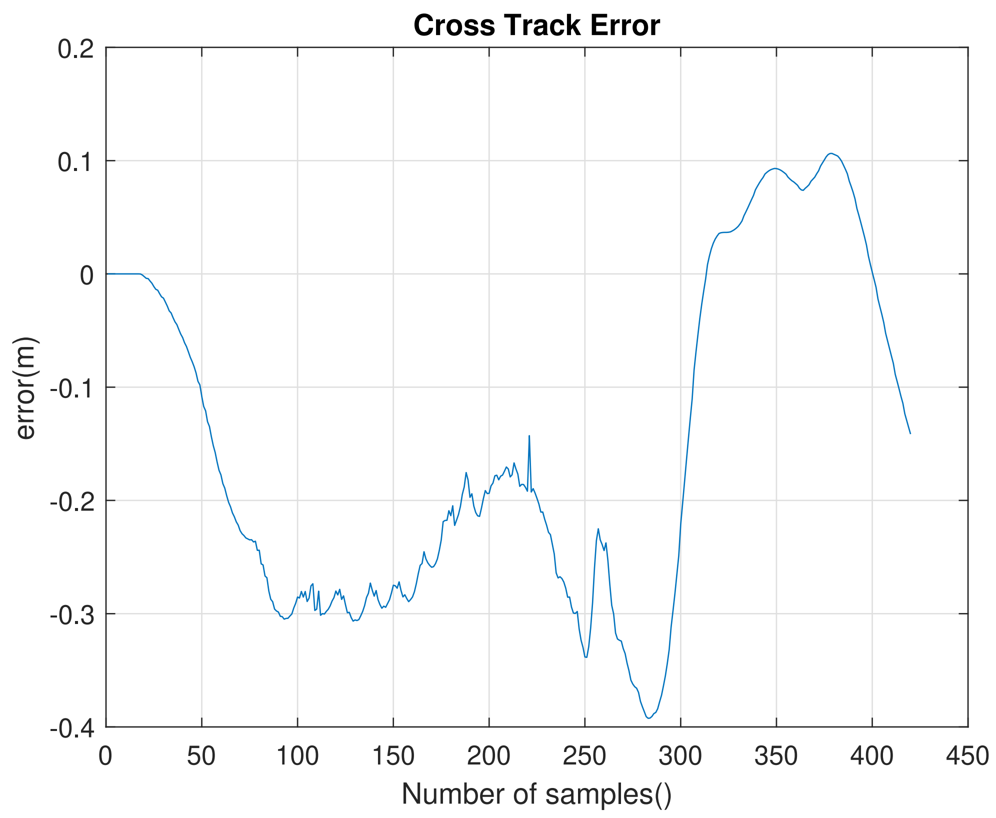

Figure 4.19: Cross Track Error

Due to the deviation made along the curve, the error amount reaches high values, then, with the straight road that follows, the deviation evolves from a negative to a positive value and is gradually compensated towards the end of the simulation. Lastly, Figure 4.20 illustrates the heading error.

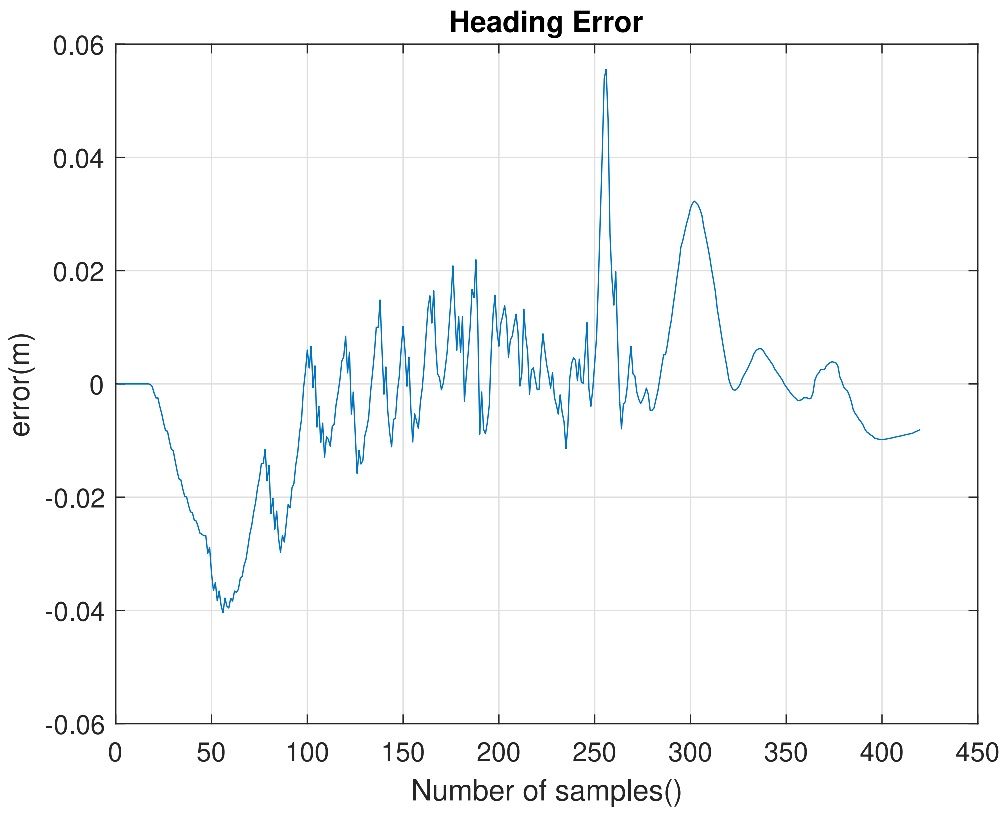

Figure 4.20: Heading Error

Similarly, although the heading error increases due to the turn in the initial part of the curve, it is minimized afterward and reaches zero values along the straight road. Tables 4.9-4.10 display the root-mean-square (RMS) and maximum errors.

Parameter	RMS Value
RMS(E_{ct})	0.2117
RMS(E_h)	0.0147

Table 4.9: RMS Values of E_{ct} and E_h in Manual Control.

Parameter	MAX Value
MAX(E_{ct})	0.1064
MAX(E_h)	0.0555

Table 4.10: MAX Values of E_{ct} and E_h in Manual Control.

4.4 Comparison in Different Controls

The results based on the performance criteria of NMPC and PID control methods have been provided and evaluated in detail above. Following these evaluations, there is a need for comparison. While the comparison is between the two different control methods that have been thoroughly examined above, comparing these methods with a vehicle driven by a driver provides the best insight. Manual driving sets the acceptable value, and a vehicle controlled autonomously should perform at least as well as manual driving in terms of safety.

Table 4.11: Comparison in Different Controls

Control method	RMS e_{ct}	RMS e_h	MAX e_{ct}	MAX e_h
NMPC Control Scenario 1	0.1292	0.0174	0.3491	0.045
PID Control Scenario 1	0.1191	0.0488	0.3977	0.2068
Manual Drive Control	0.2117	0.0187	0.1064	0.0555
NMPC Control Scenario 2	0.0851	0.0488	0.7204	0.091
PID Control Scenario 2	0.1103	0.0550	0.4513	0.2670

Table 4.11 provides a comparative analysis of different control methods, showcasing the root-mean-square (RMS) and maximum error values for cross-tracking error and heading error across different control scenarios. When examining the table, it's seen that the PID control method provides the best value in terms of cross-track error. However, it should not be forgotten that the PID control method is fundamentally designed to directly suppress this value. Therefore, our PID parameters are specific to the application and the values changed based on two different scenario. On the other hand, NMPC control has shown performance very close to PID control, displaying good performance with very little deviation from the reference axis. When manual driving is considered according to this performance criterion, it is observed to perform much worse than the other reference methods. From this perspective, the effectiveness of the NMPC and PID controllers becomes more evident. On the other hand, when examining heading error, it is easily seen that the best control method is the NMPC control. The vehicle has not oscillated at all during simulations and has always maintained the correct orientation. Meanwhile, the manual control method has produced a very close result, setting the acceptable limit. Finally, looking at the PID control, it has performed worse than these two values, and some oscillations in the vehicle were observed during the simulation. Consequently, when these methods are examined, NMPC control stands out as the unquestionably best control method due to its

consistency in providing the same values under different simulation conditions and its significantly better performance even compared to manual driving under the same simulation conditions.

Chapter 5

Conclusion

The aim of this thesis work is to compare which control method performs better in the implementation of Path tracking, a fundamental application of ADAS, in an open-source simulator environment, Carla simulator. Throughout the process, two different scenarios requiring various maneuvers and speeds have been emphasized. The necessary references for creating the scenarios have been collected either by using Carla's autonomous driving feature or by manually driving the vehicle and recording the data. Initially, dispatching was performed to provide the necessary commands for controlling our vehicle in the Carla Simulator. Subsequently, simulations were conducted for these two scenarios with the PID control method, using different PID values. Secondly, focus was placed on the NMPC control method. The essential requirement of this control method, the vehicle model parameters, was approximately determined, and simulations for the mentioned two scenarios were conducted. All scenarios were compared with the reference collected using Carla's autonomous driving feature, which represents perfect driving. Moreover, the comparison focused on Cross-track error and Heading error criteria. Finally, manual driving, which provides us with the most reliable reference for comparisons, was compared with autonomous driving. The comparisons showed that the NMPC control method produced less error compared to other control methods. Additionally, the consistent good performance of the NMPC control method with the same values across different simulations demonstrates its suitability for the mentioned ADAS application, highlighting a solution to a problem in the automation process of vehicles and advancing solutions in this field.

5.1 Future Works

This thesis work has demonstrated that NMPC control emerges as the optimal solution for Path tracking which is one of the foundational ADAS applications.

However, achieving better results with NMPC control will be possible by utilizing a more accurate vehicle model. Furthermore, the data regarding road conditions in the Carla environment should be obtained more precisely. With a more precise vehicle model and road model, the errors we have focused on will decrease, and better outcomes can be achieved at higher speeds. Finally, with the necessary improvements, simulations for different ADAS applications in the Carla environment can be conducted using an improved NMPC controller.

Bibliography

[1] Jennifer Shuttleworth. *SAE Standards News: J3016 automated-driving graphic update.* July 2019. URL: https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic (cit. on p. 1).

[2] Paul Lienert and Maria Caspani. *Americans still don't trust self-driving cars, Reuters/Ipsos poll finds.* Apr. 2019. URL: https://www.reuters.com/article/idUSKCN1RD2QV/ (cit. on p. 1).

[3] Barbara Lenz, Markus Maurer, and J Gerdes Christian. *Autonomous Driving Technical, Legal and Social Aspects.* Saint Philip Street Press, May 2016. DOI: 10.1007/978-3-662-48847-8 (cit. on p. 4).

[4] Waymo. *Home.* https://waymo.com/. Accessed: 2024-02-19 (cit. on p. 4).

[5] A. R. Plummer. «Model-in-the-Loop Testing». In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 220.3 (2006), pp. 183–199. DOI: 10.1243/09596518JSCE207. eprint: https://doi.org/10.1243/09596518JSCE207. URL: https://doi.org/10.1243/09596518JSCE207 (cit. on p. 6).

[6] Stephanie Demers, Praveen Gopalakrishnan, and Latha Kant. «A Generic Solution to Software-in-the-Loop». In: *MILCOM 2007 - IEEE Military Communications Conference.* 2007, pp. 1–6. DOI: 10.1109/MILCOM.2007.4455268 (cit. on p. 6).

[7] Hosam K. Fathy, Zoran S. Filipi, Jonathan Hagena, and Jeffrey L. Stein. «Review of hardware-in-the-loop simulation and its prospects in the automotive area». In: ed. by Kevin Schum and Alex F. Sisti. Orlando (Kissimmee), FL, May 2006, 62280E. DOI: 10.1117/12.667794. URL: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.667794 (visited on 02/20/2024) (cit. on p. 6).

[8] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. «CARLA: An Open Urban Driving Simulator». In: *CoRR* abs/1711.03938 (2017). arXiv: 1711.03938. URL: http://arxiv.org/abs/1711.03938 (cit. on p. 8).

[9] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. «AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles». In: *CoRR* abs/1705.05065 (2017). arXiv: `1705.05065`. URL: `http://arxiv.org/abs/1705.05065` (cit. on p. 8).

[10] Cole Gulino et al. «Waymax: An Accelerated, Data-Driven Simulator for Large-Scale Autonomous Driving Research». In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. 2023 (cit. on p. 9).

[11] Guodong Rong et al. «LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving». In: *arXiv preprint arXiv:2005.03778* (2020) (cit. on p. 10).

[12] Sim4CV. *Home*. `https://sim4cv.org/`. Accessed: 2024-02-11 (cit. on p. 11).

[13] CARLA. *Open-source simulator for autonomous driving research*. `https://carla.org//`. Accessed: 2024-02-10 (cit. on p. 12).

[14] Erik Coelingh, Andreas Eidehall, and Mattias Bengtsson. «Collision Warning with Full Auto Brake and Pedestrian Detection - a practical example of Automatic Emergency Braking». In: *13th International IEEE Conference on Intelligent Transportation Systems*. 2010, pp. 155–160. DOI: `10.1109/ITSC.2010.5625077` (cit. on p. 13).

[15] Meng-Yin Fu and Yuan-Shui Huang. «A survey of traffic sign recognition». In: *2010 International Conference on Wavelet Analysis and Pattern Recognition*. 2010, pp. 119–124. DOI: `10.1109/ICWAPR.2010.5576425` (cit. on p. 14).

[16] Guiru Liu, Mingzheng Zhou, Lulin Wang, Hai Wang, and Xiansheng Guo. «A blind spot detection and warning system based on millimeter wave radar for driver assistance». In: *Optik* 135 (2017), pp. 353–365. ISSN: 0030-4026. DOI: `https://doi.org/10.1016/j.ijleo.2017.01.058`. URL: `https://www.sciencedirect.com/science/article/pii/S0030402617300797` (cit. on p. 15).

[17] Lukas Küpper and Josef Schug. «Active Night Vision Systems». In: (2002). SAE Technical Paper 2002-01-0013. DOI: `10.4271/2002-01-0013`. URL: `https://doi.org/10.4271/2002-01-0013` (cit. on p. 15).

[18] Yuyu Song and Chenglin Liao. «Analysis and review of state-of-the-art automatic parking assist system». In: *2016 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. 2016, pp. 1–6. DOI: `10.1109/ICVES.2016.7548171` (cit. on p. 15).

[19] Wikipedia contributors. *Fatigue detection software — Wikipedia, The Free Encyclopedia*. Accessed: 2024-01-25. 2024. URL: `https://en.wikipedia.org/wiki/Fatigue_detection_software` (cit. on p. 16).

[20] *Adaptive Cruise Control — Deeper Learning*. Web page. Accessed: 2024-02-24. Paragraph: "Speed and distance sensors. ACC uses information from two sensors: a distance sensor that monitors the gap to the car ahead and a speed sensor that automatically accelerates and decelerates your car. ACC uses information from these sensors to adjust your speed and maintain the set distance from the car in front of you." 2024. URL: `https://mycardoeswhat.org/deeper-learning/adaptive-cruise-control/` (cit. on p. 17).

[21] José L. F. Pereira and Rosaldo J. F. Rossetti. «An integrated architecture for autonomous vehicles simulation». In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. Trento, Italy: Association for Computing Machinery, 2012, pp. 286–292. ISBN: 9781450308571. DOI: `10.1145/2245276.2245333`. URL: `https://doi.org/10.1145/2245276.2245333` (cit. on p. 18).

[22] Stephan R. Richter, Zeeshan Hayder, and Vladlen Koltun. «Playing for Benchmarks». In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2232–2241. DOI: `10.1109/ICCV.2017.243` (cit. on p. 18).

[23] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. «Playing for Data: Ground Truth from Computer Games». In: *CoRR* abs/1608.02192 (2016). arXiv: `1608.02192`. URL: `http://arxiv.org/abs/1608.02192` (cit. on p. 18).

[24] Guodong Rong et al. «LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving». In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. 2020, pp. 1–6. DOI: `10.1109/ITSC45102.2020.9294422` (cit. on p. 18).

[25] Gran Christoffer Wilhelm. «HD Maps for Autonomous Vehicles». Accessed: 2024-03-01. MA thesis. Norwegian University of Science and Technology (NTNU), 2019. URL: `https://www.ntnu.edu/documents/1284037699/1285579906/Gran-ChristofferWilhelm_2019_Master_NAP_HDMaps.pdf/79ef2eec-c9e2-454b-bf14-08d585cf8826` (cit. on p. 18).

[26] Tim Sweeney. *Welcome to Unreal Engine 4*. Mar. 2014. URL: `https://www.unrealengine.com/en-US/blog/welcome-to-unreal-engine-4` (cit. on p. 19).

[27] M.A. Johnson. «PID Control Technology». In: *PID Control: New Identification and Design Methods*. Ed. by Michael A. Johnson and Mohammad H. Moradi. London: Springer London, 2005, pp. 1–46. ISBN: 978-1-84628-148-8. DOI: `10.1007/1-84628-148-2_1`. URL: `https://doi.org/10.1007/1-84628-148-2_1` (cit. on p. 33).

[28] Tibor Nagy, Florian Enyedi, Eniko Haaz, Daniel Fozer, Andras Jozsef Toth, and Peter Mizsey. «Flexible and efficient solution for control problems of chemical laboratories». In: *29th European Symposium on Computer Aided Process Engineering*. Ed. by Anton A. Kiss, Edwin Zondervan, Richard Lakerveld, and Leyla Özkan. Vol. 46. Computer Aided Chemical Engineering. Elsevier, 2019, pp. 1819–1824. DOI: `https://doi.org/10.1016/B978-0-12-818634-3.50304-0`. URL: `https://www.sciencedirect.com/science/article/pii/B9780128186343503040` (cit. on p. 35).

[29] Microstar Laboratories collaboratives. *Ziegler-Nichols Method*. ScienceDirect Topics in Computer Science. Accessed: 2024-02-20. URL: `https://www.sciencedirect.com/topics/computer-science/ziegler-nichols-method#:~:text=The%20Ziegler%2DNichols%20Method%20is,transfer%20function%20with%20dead%20time` (cit. on p. 35).

[30] James Bennett, Ajay Bhasin, Jamila Grant, and Wen Chung Lim. *PID Tuning via Classical Methods - Cohen-Coon Method*. Chemical Process Dynamics and Controls (Woolf) on LibreTexts. Chapter 9.3.5 Cohen-Coon Method, Accessed: 2024-02-21. URL: `https://eng.libretexts.org/Bookshelves/Industrial_and_Systems_Engineering/Chemical_Process_Dynamics_and_Controls_(Woolf)/09%3A_Proportional-Integral-Derivative_(PID)_Control/9.03%3A_PID_Tuning_via_Classical_Methods#:~:text=The%20Cohen%2DCoon%20method%20is,evaluate%20the%20initial%20control%20parameters` (cit. on p. 35).

[31] *Design of Bio-Inspired Optimized Integer And Fractional Pid Controller*. 2022. DOI: `10.18090/samriddhi.v14i03.02` (cit. on p. 36).

[32] *PID controller with computational optimization*. 2023. DOI: `10.1016/b978-0-12-821204-2.00007-6` (cit. on p. 36).

[33] *Design of Robust PID Controllers for SOFC Stacks*. 2022. DOI: `10.1109/ccta49430.2022.9966041` (cit. on p. 36).

[34] *A Summary of PID Control Algorithms Based on AI-Enabled Embedded Systems*. 2022. DOI: `10.1155/2022/7156713` (cit. on p. 36).

[35] *Review of Research on Improved PID Control in Electro-hydraulic Servo System*. 2023. DOI: `10.2174/1872212117666230210090351` (cit. on p. 36).

[36] Frank Allgöwer and Alex Zheng, eds. *Nonlinear Model Predictive Control*. 1st ed. Progress in Systems and Control Theory. Basel: Birkhäuser Basel, 2000. ISBN: 978-3-7643-6297-3. DOI: `10.1007/978-3-0348-8407-5`. URL: `https://doi.org/10.1007/978-3-0348-8407-5` (cit. on p. 45).

[37] S. Joe Qin and Thomas A. Badgwell. «An Overview of Nonlinear Model Predictive Control Applications». In: *Nonlinear Model Predictive Control*. Basel: Birkhäuser Basel, 2000, pp. 369–392 (cit. on p. 45).

[38] Boris Houska, Hans Joachim Ferreau, and Moritz Diehl. «An auto-generated real-time iteration algorithm for nonlinear MPC in the microsecond range». In: *Automatica* 47.10 (2011), pp. 2279–2285 (cit. on p. 46).

[39] Efstathios Siampis, Efstathios Velenis, Salvatore Gariuolo, and Stefano Longo. «A Real-Time Nonlinear Model Predictive Control Strategy for Stabilization of an Electric Vehicle at the Limits of Handling». In: *IEEE Transactions on Control Systems Technology* 26.6 (2018), pp. 1982–1994. DOI: `10.1109/TCST.2017.2753169` (cit. on p. 46).

[40] Mattia Boggio, Carlo Novara, and Michele Taragna. «Trajectory planning and control for autonomous vehicles: a "fast" data-aided NMPC approach». In: *European Journal of Control* (2023), p. 100857. ISSN: 0947-3580 (cit. on p. 46).

[41] Mattia Boggio, Carlo Novara, and Michele Taragna. «Nonlinear Model Predictive Control: an Optimal Search Domain Reduction». In: *IFAC-PapersOnLine* 56.2 (2023), pp. 6253–6258 (cit. on p. 46).