POLITECNICO DI TORINO

Master's Degree in Mechanical Engineering

Master's Degree Thesis

Development of Path Tracking Control Strategies for Autonomous Vehicles and Validation Using a High-Fidelity Driving Simulator

Supervisors:

Prof. CARLO NOVARA

Prof. Mattia Boggio

Prof. Fabio Tango

Candidate:

Amir Aryan Rezaie

March 2025

Abstract

The development of Advanced Driver Assistance Systems (ADAS) represents a rapidly evolving field, focused on enhancing vehicle safety and efficiency through the automation of key driving functions, including steering, acceleration, and braking. These systems serve as the foundation for autonomous vehicles, which aim to operate without human intervention through the integration of sensors, control algorithms, and real-time decision making. However, evaluating control strategies in real-world conditions is often costly and impractical, thereby highlighting the need of using simulation environments for rigorous testing and validation. In this context, this thesis presents a simulation framework for evaluating control strategies in autonomous vehicle applications, integrating Simulink for controller design with the CARLA simulator for realistic testing scenarios. Two control methodologies, namely Proportional-Integral-Derivative (PID) control and Nonlinear Model Predictive Control (NMPC), are implemented to control both lateral and longitudinal vehicle dynamics. To improve the interface between the control system and the CARLA simulator, a dispatching function is developed to convert the desired acceleration outputs from the controller into appropriate vehicle input commands, namely throttle and brake commands. This function is designed based on vehicle data collected from the CARLA simulation environment. The proposed framework is tested in highway scenarios featuring both curved and straight road segments, including overtaking maneuvers. Simulation results validate the effectiveness of the control strategies, highlighting their performance and applicability in real-world driving conditions.

Keywords: ADAS, Path tracking, Control System, NMPC, PID Control, Simulink, CARLA simulator

Contents

List of Figures

Acronyms

AV
Autonomous Vehicle

ADS
Automated Driving System

ADAS
Advanced Driver Assistance Systems

IMU
Inertial Measurement Unit

LiDAR
Light Detection And Ranging

RADAR
Radio Detection And Ranging

SLAM
Simultaneous Localization And Mapping

GPS
Global Positioning System

PID
Proportional–Integral–Derivative

MPC
Model Predictive Control

NMPC
Nonlinear Model Predictive Control

LMI
Linear Matrix Inequality

CARLA
Car Learning to Act

LGSVL
LG Silicon Valley Lab Simulator

SIM4CV
Simulation for Computer Vision

LQR
Linear Quadratic Regulator

SMC
Sliding Mode Control

V2X
Vehicle-to-Everything

V2V

Vehicle-to-Vehicle

V2I

Vehicle-to-Infrastructure

V2P

Vehicle-to-Pedestrian

V2N

Vehicle-to-Network

ACC

Adaptive Cruise Control

AEB

Autonomous Emergency Braking

LKA

Lane Keeping Assist

TSR

Traffic Sign Recognition

APA

Automatic Parking Assist

BSD

Blind Spot Detection

RCTA

Rear Cross Traffic Alert

HIL

Hardware-in-the-Loop

SIL

Software-in-the-Loop

MIL

Model-in-the-Loop

VDM

Vehicle Dynamics Model

KDM

Kinematic Bicycle Model

TDM

Tire Dynamics Model

DST

Dynamic Single-Track Model

Chapter 1

Introduction

1.1 Autonomous Driving

Autonomous driving, also known as self-driving or driverless technology, refers to the technology enabling vehicles to navigate and operate independently without human intervention, leveraging a combination of sensors, machine learning algorithms, and real-time data processing, with the aim of enhancing road safety by reducing human errors, which are responsible for the majority of traffic accidents [1]. Additionally, AVs aim to improve traffic flow, lower emissions, and increase accessibility for individuals who are unable to drive due to age or disability [2].

The advent of autonomous vehicles (AVs) however is not merely a technological evolution; it is a reflection of deeper philosophical inquiries into human progress and our relationship with machines. Basically Autonomous Vehicle research is rooted in fundamental concerns about human efficiency, and societal transformation. One of the primary motivations for AV development is the reduction of traffic-related fatalities and injuries. Human error accounts for over 90 percent of road accidents, driven by distractions or fatigue [3]. The introduction of AVs aims to eliminate such risks through precise, data-driven decision-making. AVs are also often portrayed as environmentally and economically beneficial due to lower emissions with their efficient driving and electrification, allowing for greener cities and environment. Hence as AVs take over driving responsibilities, trust in artificial intelligence becomes important [4].

Humans naturally develop trust in systems when they behave consistently and predictably. Traditional human-driven cars, despite their risks, follow intuitive rules of behavior. AVs, however, operate based on complex probabilistic models, which might lead to unpredictable behavior in rare or ambiguous scenarios. AI-driven vehicles make decisions based on sensor data fusion, deep learning models, and probabilistic reasoning. Despite these challenges, autonomous driving research remains one of the most promising frontiers in AI and transportation science. By refining sensor accuracy, improving predictive models, and enhancing AI explainability,

researchers can reduce uncertainty and improve AV performance in real-world conditions. Furthermore, continuous advancements in machine learning and Regulatory control will ensure that AVs evolve into safer and more reliable alternatives to human drivers [5].

The journey toward autonomous driving began decades ago with the introduction of basic driver-assistance features, such as cruise control and anti-lock braking systems (ABS). Over time, these functions evolved into more sophisticated advanced driver-assistance systems (ADAS), including lane-keeping assist, adaptive cruise control, and automated parking [6]. The Society of Automotive Engineers (SAE) classifies autonomous driving technologies into six levels, from Level 0 (no automation) to Level 5 (full automation under all conditions At lower levels (1-2), vehicles primarily assist human drivers with functions like adaptive cruise control and lane-keeping assistance, while higher automation levels (3-5) allow vehicles to independently manage driving tasks under specific conditions or entirely autonomously [7]. Figure 1.1 below better represents the detailed classification of SAE levels:
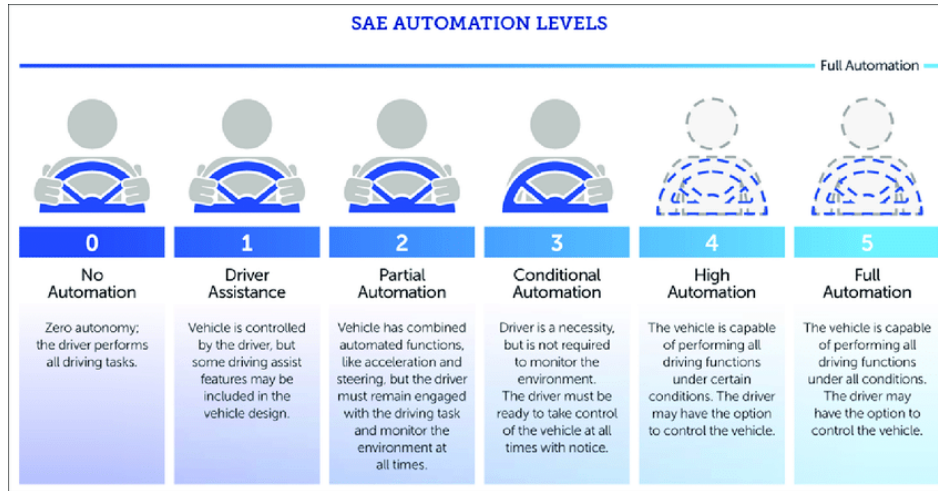
Figure 1.1: Levels of Automation by SAE

- **Level 0 No Automation** The human driver performs all aspects of the driving task at all times. Even though it may have basic safety features like anti-lock brakes or stability control, the driver is fully responsible for accelerating, braking, steering, and monitoring the environment.

- **Level 1 Driver Assistance** The vehicle can assist the driver with either steering or acceleration/braking, but not both simultaneously. The driver remains responsible for monitoring the driving environment, continuously supervises the automation feature and must be ready to take full control at any time. Features like Adaptive Cruise Control (ACC) assist the driver by maintaining a predefined distance from the leading vehicle, although steering continues to be controlled manually by the driver.

- **Level 2 Partial Automation** The vehicle can control both steering and acceleration/braking simultaneously under certain conditions. The driver, however, must monitor the environment and remain fully engaged. Features like Highway Driving Assist combine adaptive cruise control and lane-centering on well-marked highways. The driver is still the primary decision-maker and must supervise the system.

- **Level 3 Conditional Automation** The vehicle can manage most aspects of driving—steering, acceleration, braking, and environmental monitoring—under specific conditions, but the driver must be ready to take control when the system requests. The vehicle drives autonomously at lower speeds on highways or in heavy traffic, handling lane keeping, following distances, and object detection. The human driver does not need to monitor the driving environment continuously while the automated system is active. However, when the system encounters a scenario outside its operational design domain, such as unclear road markings or complex intersections, it will issue a "handover request," and the human must then take over [8].

- **Level 4 High Automation** The vehicle can drive itself without human intervention within a defined operational domain or set of conditions. If the system cannot cope with conditions outside its domain, it will bring the vehicle to a safe stop if a human does not intervene. Operating on fixed routes in urban areas with well-understood traffic rules and conditions. A human driver may not be required to take over at any time within the operational domain.

- **Level 5 Full Automation** The vehicle can handle all aspects of driving under all conditions that a human driver could manage—no human driver is needed at any point. The system can navigate complex roads, weather conditions, and unexpected scenarios without any human input. Fully Autonomous Vehicles in theory could drive from any point A to point B without human assistance, whether on highways, rural roads, or urban centers. Commercial or private vehicles that do not require steering wheels or pedals, since human control is unnecessary.

The development of autonomous vehicles has been driven by continuous improvements in artificial intelligence, sensor technology, and computational power. These advancements have enabled AVs to process large amounts of data in real-time, enhancing their ability to perceive and navigate complex environments. Google initiated one of the most ambitious self-driving car projects in history. This effort, which later evolved into Waymo (a subsidiary of Alphabet Inc.), was one of the first large-scale autonomous vehicle research initiatives focused on achieving full autonomy (SAE Level 4 and Level 5), relying heavily on LiDAR (Light Detection and Ranging) technology to create high-resolution, 3D maps of its environment. with the aim of creating a vehicle that could drive without human intervention in urban and highway environments [9].
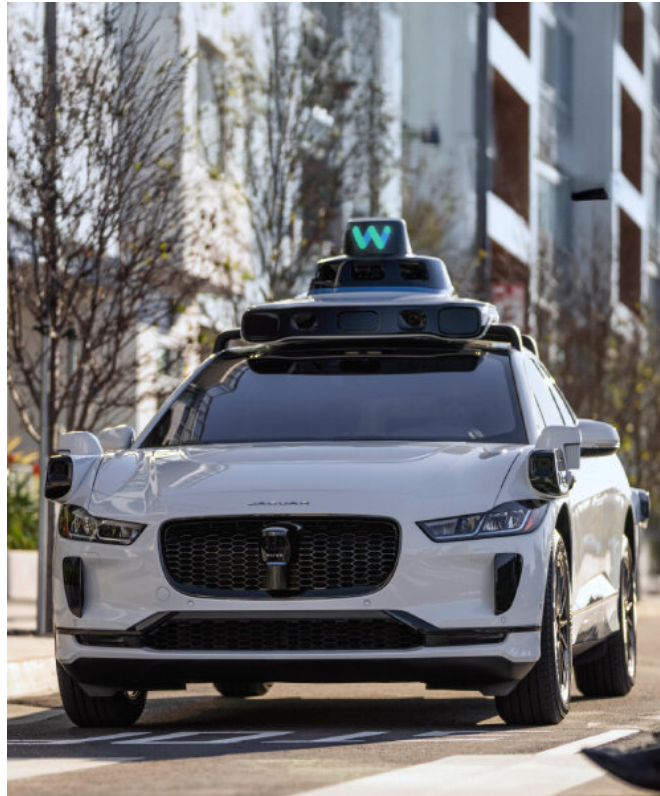
Figure 1.2: Waymo Autonomous Vehicle

Figure 1.3: Tesla Autonomous Vehicle

While Waymo pursued full autonomy from the starting point of research, Tesla

took a different approach—introducing semi-autonomous driving features aimed at enhancing human driving rather than replacing it entirely. Tesla's Autopilot does not use LiDAR. Instead, it relies on mounted surround cameras, radar, and ultrasonic sensors to perceive its environment, processing real-time video data to detect lane markings, vehicles, pedestrians, and road signs [10].

These two pioneering approaches, Waymo's safety-first strategy and Tesla's data-driven evolution, reflect broader debates in the use of artificial intelligence and engineering, demonstrating the complexity of autonomous driving innovation.

1.2   Functional Components of AVs

The architecture of an autonomous vehicle (AV) is a multi-layered system that combines hardware and software modules to enable the vehicle to perceive its surroundings, localize itself, make decisions, and execute control actions. This structured framework is essential for ensuring the safe and reliable operation of AVs in dynamic and unpredictable environments [11].

Early AV architectures were centralized and primarily relied on rule-based decision-making and pre-programmed algorithms. Therefore, these systems were to function merely in controlled environments, and face difficulties with scalability and real-time adaptability in more complex urban settings [12]. Modern architectures then integrated a combination of deep learning and edge computing as well as advanced communication protocols which shift AVs to process vast amounts of data in real-time, making autonomous driving safer and more efficient.
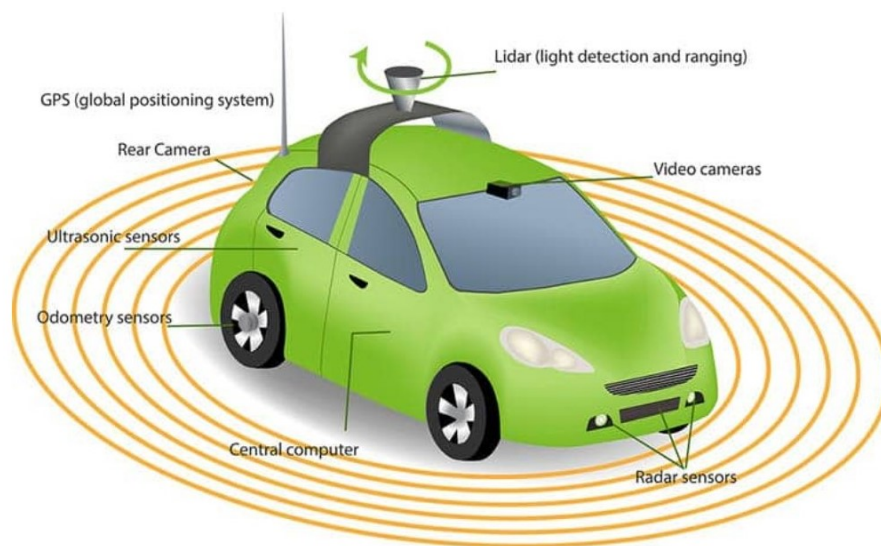
Figure 1.4: Array of Sensors Implemented in AVs

1.2.1 Layered Architecture

A well-structured AV system typically follows a layered architecture that compartmentalizes functions into discrete but interrelated modules. This separation of concerns is essential for ensuring modularity, maintainability, and scalability [13]. The key layers generally include:

- **Sensing and Perception Layer** In reality it functions like the eyes and brain of the AV, enabling the vehicle to understand and interpret its surroundings. This layer is responsible for collecting data from various sensors, processing it in real time, and creating a precise representation of the external environment. The processing involves identifying and classifying objects such as other vehicles, pedestrians, road signs, and obstacles. Sensor fusion techniques combine raw data to create a comprehensive environmental model. Advanced algorithms, including convolutional neural networks (CNNs) and transformer-based models, are employed to process also these raw data [14]. The system not only detects what is present but also tracks the movement of these objects over time, effectively predicting how they might behave in the near future, ensuring safe and informed driving decisions.

- **Localization and Mapping Layer** It can be thought as vehicle's internal navigation system. It provides not only the car with a detailed map of its surroundings but also continuously determines where the vehicle is within that map. The process begins with high-definition maps that contain a wealth of information about road layouts, lane markings, landmarks, and other static features of the environment. Autonomous vehicles generally rely on a combination of GPS, Inertial Measurement Units (IMUs), and Simultaneous Localization and Mapping (SLAM) algorithms to determine their position within an environment [15].

- **Planning and Decision-Making Layer** This layer takes all the detailed information gathered by the sensors and processed by the perception system and uses it to plan the vehicle's next moves. It doesn't just react to what's happening right now. Regarding the Path Planning, for example, it computes a geometric path from the current position to the destination, considering road topology, traffic rules, and obstacles. Techniques such as Model Predictive Control (MPC) and Reinforcement Learning (RL) are often used to optimize these trajectories [16]. It also predicts what might happen in the next step. For instance, if a vehicle is about to change lanes, this layer evaluates these possibilities and decides on the safest and most efficient path forward.

- **Control Layer** Executes the planned maneuvers by interfacing with the vehicle's actuators. Basically, it is like a bridge between the vehicle's decisions and its physical actions. After the Planning and Decision-Making layer, which figures out what the vehicle should do, it translates it into real-world commands to the car's actuators—components like the steering system, throttle, and brakes—to ensure that the vehicle follows the planned trajectory accurately and safely.

Each of these layers interacts with the others through well-defined interfaces, allowing developers to update or replace individual modules without compromising the entire system.

A significant aspect of modern AV architecture is the integration of Vehicle-to-Everything (V2X) communication. This communication serves as an additional layer of data input that complements onboard sensors [17]. While sensors like LiDAR, radar, and cameras provide useful but limited real-time local information, V2X enables vehicles to exchange data with nearby vehicles (V2V), infrastructure (V2I), pedestrians (V2P), and even cloud networks (V2N). This extended communication network allows an AV to gain insights beyond its line-of-sight—such as receiving early warnings about upcoming dangers or even traffic conditions [18].

Figure 1.5: V2X Technology

From a safety standpoint, by allowing vehicles to share critical information instantaneously, V2X supports cooperative maneuvers and proactive hazard avoidance. For instance, if one vehicle detects a sudden obstacle or slippery road conditions, it can immediately alert others, potentially preventing accidents before they occur. This level of communication not only augments the vehicle's perception but also contributes to the overall reduction of traffic incidents and enhances road safety for both autonomous and human-driven vehicles [19]. Hence the integration of V2X highlights the importance of developing common standards and protocols. Governments and industry bodies are working together to establish frameworks that ensure interoperability and secure data exchange among different manufacturers and systems, as these standards are crucial for fostering trust in the technology and ensuring that V2X-enabled vehicles can communicate seamlessly regardless of their origin, thereby laying the groundwork for widespread adoption.

Correspondingly autonomous vehicles generate and process massive amounts of data. To manage this computational load, AV architectures increasingly adopt a hybrid computing approach that leverages both edge and cloud resources [20]. Edge computing enables real-time processing of critical sensor data onboard the vehicle, reducing latency and ensuring immediate responsiveness. In parallel, cloud comput-

ing is used for tasks that require intensive computation or benefit from aggregated data across multiple vehicles—such as machine learning model training, fleet management, and large-scale data analytics [21]. This dual strategy not only enhances the vehicle's performance but also allows for continuous updates and improvements through over-the-air (OTA) software updates, ensuring that AV systems remain at the cutting edge of technology.

1.3 Application and Challenges

Advanced Driver Assistance Systems (ADAS) are designed to enhance vehicle safety and comfort by aiding the driver with real-time information and, in certain cases, automated corrective actions. Among widely implemented ADAS features are mentioned in the following:

• **Adaptive cruise control** Adaptive Cruise Control automatically adjusts the vehicle's speed to maintain a safe and preset distance from the vehicle ahead. It employs radar or camera sensors mounted at the front of the vehicle to continuously monitor the speed and distance of the preceding vehicle [22]. Based on sensor data, ACC automatically accelerates or decelerates, enhancing driving comfort on highways and reducing driver fatigue.
    vspace10pt
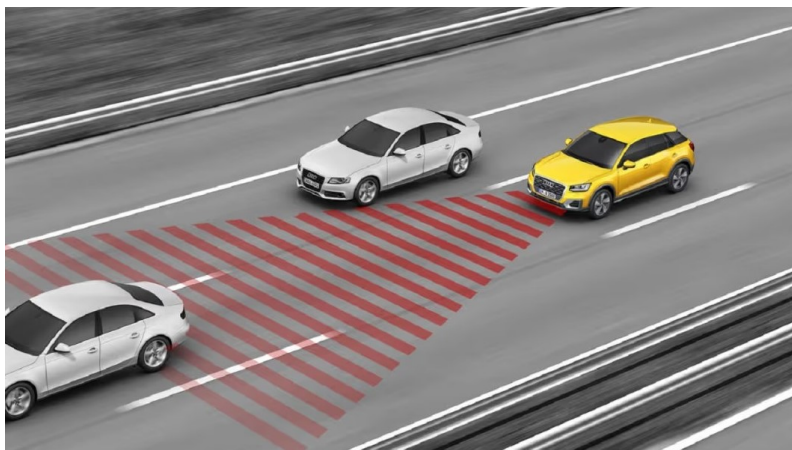
Figure 1.6: Adaptive cruise control

• **Lane keeping assist** These systems detect lane markings using camera sensors, typically mounted behind the windshield. Lane Departure Warning provides an alert (visual, audible, or tactile) if the vehicle unintentionally drifts from its lane [23]. Lane Keeping Assist actively intervenes by steering the vehicle back into the center of the lane, improving safety by preventing accidents caused by driver distraction or fatigue.
    vspace10pt
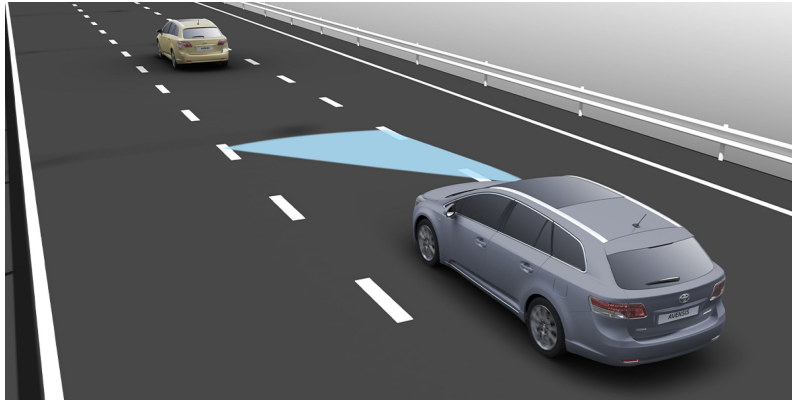
Figure 1.7: Lane keeping assist

- **Blind spot monitoring** Blind Spot Detection systems utilize radar or ultrasonic sensors placed at the sides or rear of the vehicle to monitor areas not easily visible in mirrors [24]. When a vehicle enters this blind spot area, BSD alerts the driver through visual indicators typically placed on side mirrors or through audible warnings, reducing the risk of collisions during lane changes.

    vspace10pt
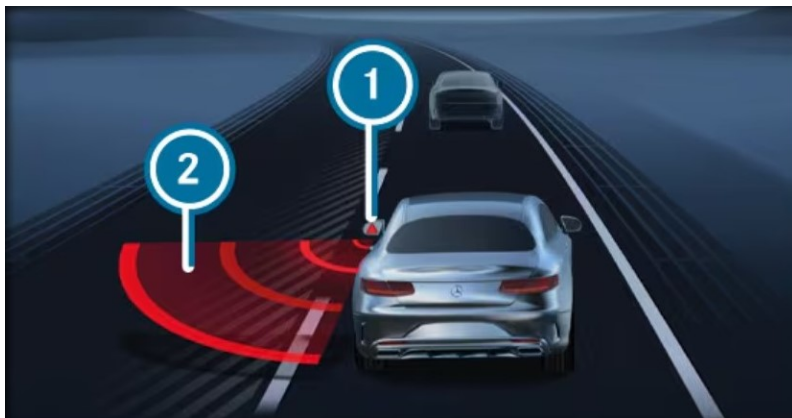
Figure 1.8: Blind spot monitoring

- **Automatic emergency braking** Automatic Emergency Braking uses sensors like radar, cameras, or LiDAR to detect obstacles, such as vehicles or pedestrians, ahead of the car. If a collision risk is detected, the system first alerts the driver, then automatically applies brakes to either prevent or reduce the severity of a collision. AEB significantly reduces the chances of front-end collisions and enhances overall safety [25].

    vspace10pt
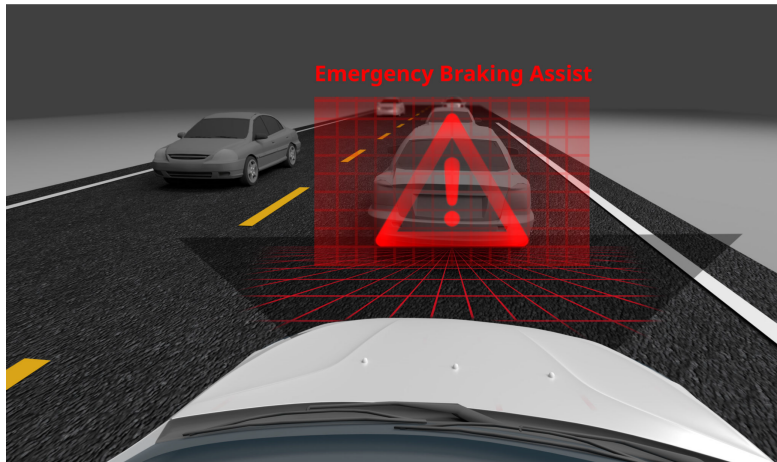
Figure 1.9: Automatic emergency braking

• **Traffic sign recognition** This feature employs camera sensors with image processing technology to detect and recognize road signs, including speed limits, stop signs, and warning signs. The recognized signs are displayed to the driver on the vehicle dashboard or head-up display, enhancing driver awareness and compliance with traffic rules.

vspace10pt
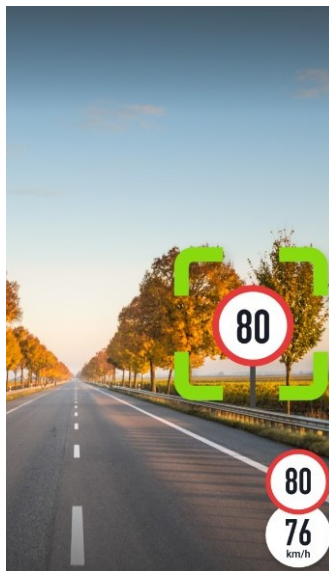
Figure 1.10: Traffic sign recognition

• **Driver drowsiness alerts** Driver Drowsiness Detection systems monitor driver alertness and behavior through steering inputs, eye-tracking cameras, or facial recognition technology. If the system detects signs of fatigue or distraction, it alerts the driver to take a break, reducing the risk of accidents due to drowsiness.

- **Rear Cross Traffic Alert** Rear Cross-Traffic Alert systems use radar or ultrasonic sensors placed at the rear bumper to detect vehicles approaching from either side while reversing out of a parking space. The system warns drivers through visual or audible alerts, significantly enhancing safety when reversing in busy areas.

vspace10pt
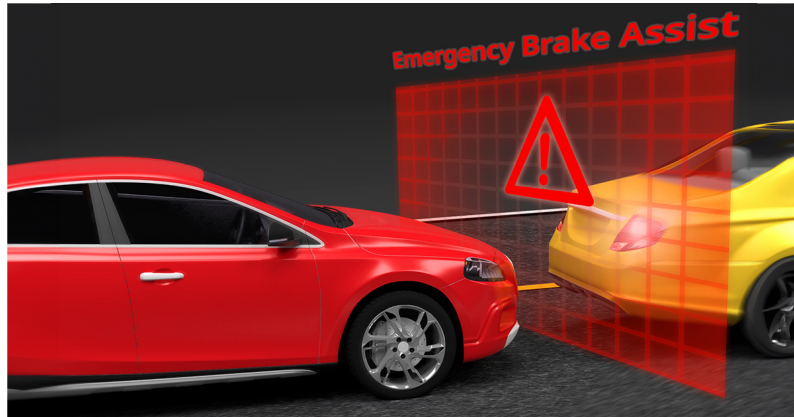
Figure 1.11: Rear Cross Traffic Alert

Despite the advancements in the field of AVs, several challenges still remain in the widespread adoption of autonomous vehicles. One of the primary obstacles is ensuring the safety and reliability of AV systems in diverse and dynamic environments [26]. Autonomous vehicles must be capable of handling a wide range of scenarios, including adverse weather conditions, unexpected pedestrian behavior, and complex urban infrastructure, which necessitate extensive testing and validation through simulation platforms such as CARLA and real-world driving trials.

Another critical challenge is the ethical and legal implications associated with autonomous driving. Determining liability in accidents involving AVs is a complex issue, as responsibility may fall on vehicle manufacturers and even software developers [27]. Moreover, ethical dilemmas arise in situations where AVs must make split-second decisions that involve potential harm to pedestrians or passengers. Addressing these concerns requires a collaborative effort among policymakers, engineers, and ethicists to establish clear regulatory frameworks [28].

Cybersecurity is another pressing concern in the realm of autonomous vehicles. As AVs rely heavily on interconnected systems, including cloud computing and vehicle-to-vehicle (V2V) communication, they become susceptible to cyber threats [29]. Unauthorized access to AV systems could lead to malicious interventions, posing risks to both passengers and other road users. Ensuring robust cybersecurity measures is essential to prevent potential cyber-attacks and maintain public trust in AV technology.

Furthermore, the integration of AVs into existing transportation infrastructure

poses logistical and financial challenges. Roads, traffic signals, and signage may need to be adapted to accommodate autonomous vehicles, requiring significant investment from governments and private stakeholders. Public acceptance and trust in AV technology also play a crucial role in its adoption. Many individuals remain skeptical about the safety and reliability of self-driving cars, necessitating public awareness campaigns and transparent safety demonstrations to build confidence in AV systems.

The economic implications of AVs also warrant careful consideration. While autonomous vehicles have the potential to reduce transportation costs and increase efficiency, they may also lead to job displacement, particularly in industries reliant on human drivers, such as trucking and taxi services. Policymakers must address these socio-economic concerns by implementing workforce transition programs and exploring new job opportunities in the evolving mobility landscape.

1.4   The Role of Simulation in AV Development

Simulation plays an essential role in the development and testing of automated driving systems (ADS), allowing safe and cost-effective evaluation of complex or rare scenarios which are not easily feasible in real-world settings [30]. The high complexity nature of real-world road environments requires extensive testing in diverse conditions—ranging from varying weather and lighting to intricate urban layouts. Moreover, physical testing under these conditions can be financially prohibitive, time-consuming, and potentially hazardous.

These digital environments also offer several key advantages. First, it enables the decoupling of system components—such as perception, sensor fusion, planning, and control—for individual assessment as well as integrated, closed-loop testing. This modular testing is essential for identifying vulnerabilities in each subsystem and understanding how errors or delays in one module may propagate through the entire system. For example, simulation can model sensor noise, latency, and failure modes, allowing developers to evaluate how robust the sensor fusion algorithms are in aggregating data from LiDAR, radar, and cameras under adverse conditions [31]. Simulation is invaluable for integrating and testing the multiple subsystems that comprise an autonomous vehicle—like perception, localization, planning, and control—in a cohesive and synchronized manner. It allows for iterative improvements and rapid prototyping, where updates to one part of the system can be evaluated in context with the rest of the vehicle's operations. This modular testing is crucial, as it ensures that even when individual components are well-tuned, their combined behavior under diverse conditions is safe and reliable.

Moreover, it enables iterative experimentation, namely one can adjust parameters, introduce new obstacles, or modify vehicle configurations to observe system responses of different models. This iterative process accelerates innovation by reducing the time between design, implementation, and analysis.
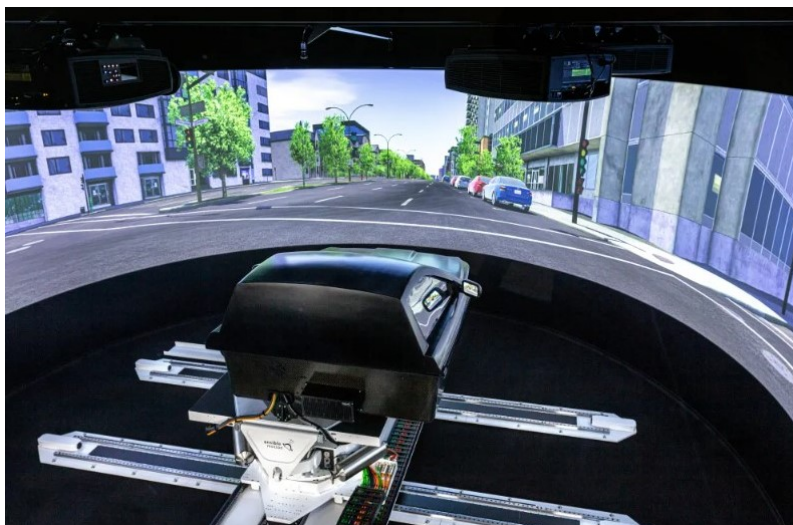
Figure 1.12: Simulation of ADAS Testing

Additionally, simulation allows for performance evaluation and benchmarking. Advanced metrics—such as accuracy in object detection, reaction time to critical events, and success rates in various maneuvers—can be gathered systematically [32]. These data-driven insights guide further refinements in software and hardware design. Furthermore, simulating multiple autonomous vehicles, often termed "multi-vehicle simulation," provides valuable information on coordination algorithms and infrastructure requirements. Within the AV sector, simulation frameworks are closely integrated with three fundamental methodologies: Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), and Hardware-in-the-Loop (HIL). These methodologies form a progression of testing stages, each leveraging simulation in distinct ways to ensure system reliability and performance before deploying physical prototypes on actual roads.

Model-in-the-Loop (MIL) In MIL testing, developers evaluate mathematical models or algorithmic representations of an AV's components within a simulated environment. The "loop" here refers to the closed feedback loop between the simulated plant and the control logic. This is usually performed at an early stage of development when the system's structure is still fluid, allowing for rapid iterations. An autonomous emergency braking algorithm can first be modeled in MATLAB/Simulink and tested against a virtual car-following scenario. By observing responses to different vehicle speeds and braking profiles, researchers can refine their model parameters and validate feasibility before committing to concrete software or hardware.

Software-in-the-Loop (SIL) SIL testing marks the transitions from purely model-based representations to the actual software code intended for deployment in the AV. The control algorithms, perception modules, and decision-making logic would be run on a host computer, while the environment such as vehicle dynamics, sensors, and roadway remains simulated. This allows one to check for real-time

performance issues and software bugs under controlled but realistic conditions. One of the primary advantages of SIL is its ability to replicate complex, dynamic driving scenarios within a controlled setting [33]. For instance, an autonomous driving system designed for urban navigation might face numerous challenges such as unpredictable pedestrian movement, variable traffic signals, and interacting vehicles.

Hardware-in-the-Loop (HIL) HIL testing integrates real hardware components—such as Electronic Control Units (ECUs), sensors, and actuators—into a simulated environment. In this setup, the physical hardware interacts directly with a virtual vehicle and its surroundings, effectively closing the loop between real-world inputs and simulated outputs. This method allows for comprehensive validation, not only of the software's performance but also of the behavior, reliability, and interaction of physical components under a wide range of operating conditions. A real steering actuator or braking mechanism can be connected to a high-fidelity driving simulator [34]. The simulator provides inputs that mimic road interactions, while the hardware responds in real time. Any discrepancies or latencies in hardware responses are detected early, allowing engineers to calibrate the system before road testing.

1.5   Advances in Autonomous Driving Simulators

Modern simulators employ sophisticated rendering engines and physics models to replicate real-world scenarios, from road geometry to environmental conditions. They also provide collaborative research, as many of these platforms are open-source, allowing academic institutions and industry stakeholders to contribute new features and modules. Among the most prominent CARLA, Sim4CV, and LGSVL (also known as SVL Simulator) can be named [35]. These platforms serve as powerful virtual environments where perception, planning, and control algorithms can be iterated rapidly and at scale.

1.5.1   CARLA

CARLA (Car Learning to Act) is an open-source simulator developed specifically for autonomous driving research. Built on the Unreal Engine, CARLA emphasizes high-fidelity rendering, realistic vehicle dynamics, and comprehensive sensor simulation [36]. Its open-source nature encourages community involvement, fostering continuous improvements and the addition of custom features. Researchers utilize CARLA to design numerous driving scenarios—ranging from urban intersections to suburban roads—each populated with pedestrians, other vehicles, and environmental factors such as changing weather. CARLA's extensibility is further demonstrated through its ability to integrate with external tools such as Python API and Simulink, facilitating the development and validation of advanced perception, planning, and control algorithms. Additionally, the simulator supports multiple map

environments, including complex urban layouts and highway scenarios, ensuring robust testing across diverse driving conditions [37]. CARLA adopts a structured methodology emphasizing the following characteristics:

Figure 1.13: CARLA Simulator Environment

• **Sensor Simulation and Data Generation** CARLA provides high-fidelity sensor simulations, emulating realistic sensor behavior, including RGB cameras, LiDAR, radar, IMU, GNSS, and semantic segmentation cameras. These virtual sensors generate data with realistic noise characteristics, occlusions, reflections, and distortions typically found in real-world conditions.

• **Environment Creation and Scenario Design** CARLA supports custom map creation using tools like RoadRunner or Unreal Engine Editor, allowing researchers to design specific test environments tailored to particular research objectives. Scenario definition can be scripted through CARLA's ScenarioRunner, enabling precise reproduction and automation of driving situations such as pedestrian crossings, vehicle merging maneuvers, or emergency braking scenarios.

• **Integration with Control Systems and Machine Learning Models** CARLA seamlessly integrates with external control and planning algorithms via Python and ROS interfaces. This interoperability enables researchers to directly test their autonomous driving algorithms within the virtual environment without modifications, facilitating rapid iterations.

• **Open Source and Community-driven** The open-source nature of CARLA promotes community engagement, enabling the rapid exchange of knowledge and continuous development of new features.

1.5.2 Sim4CV

Sim4CV (Simulation for Computer Vision) is based on bridging the gap between computer vision research and robotics or autonomous driving applications. The methodology behind Sim4CV revolves around generating high-fidelity synthetic images using advanced computer graphics techniques [38]. These images provide a rich dataset for training and evaluating machine learning models, bridging the gap between synthetic data generation and real-world applicability by providing photo-realistic virtual scenes enriched with precise, automatically generated annotations. The main advantages of this simulator consist of the following:
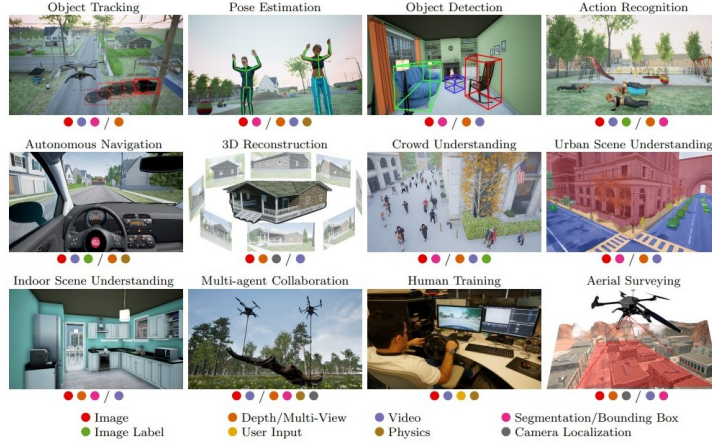
Figure 1.14: SIM4CV Simulator common Computer Vision Applications

• **Scene Creation** Sim4CV uses advanced rendering engines (commonly Unreal Engine or Unity) to create realistic urban and suburban environments, including dynamic conditions like variable weather (rain, fog, night lighting), multiple vehicle interactions, pedestrian movements, and complex infrastructure layouts.

• **Data Integration with Machine Learning Pipelines** The data generated from Sim4CV seamlessly integrates into existing machine learning frameworks such as TensorFlow, PyTorch, and Keras. Researchers can export data directly into compatible formats, facilitating rapid model training, validation, and benchmarking without needing additional preprocessing.

• **Photorealistic Rendering** Sim4CV employs advanced graphics engines (e.g., Unreal Engine, Unity) to ensure realistic appearance and lighting conditions that closely approximate real-world visuals.

1.5.3 LGSVL

The LGSVL Simulator is an advanced open-source simulation platform explicitly designed for autonomous vehicle (AV) development and testing. Created initially by LG Electronics and further supported by the AutonomouStuff and Open Robotics community, LGSVL has rapidly become one of the leading simulation tools for autonomous driving research, recognized widely for its realism, versatility, and ease of integration with popular autonomous driving software stacks [39]. Built on the Unity game engine, LGSVL excels in creating realistic and highly interactive virtual environments that closely mimic real-world driving conditions, allowing developers to perform thorough testing of perception, planning, and control algorithms in a safe, efficient, and repeatable manner. The primary methodology behind LGSVL Simulator revolves around providing a high-fidelity virtual environment that integrates seamlessly with existing software frameworks:
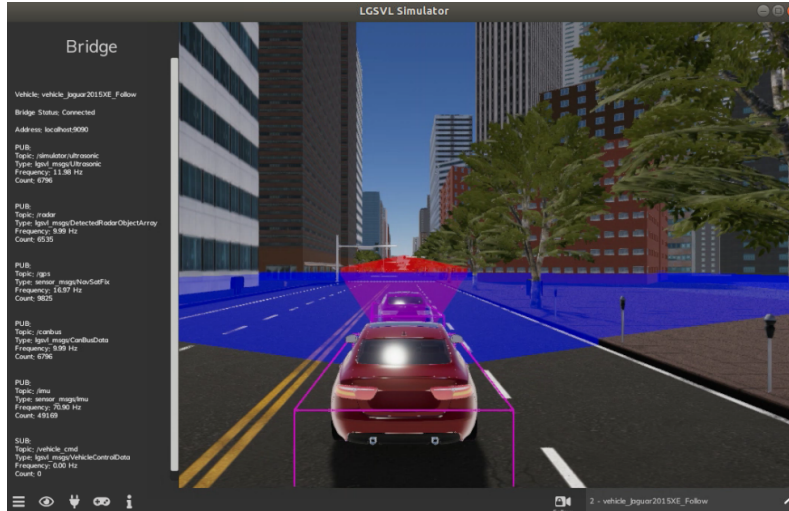
Figure 1.15: LGSVL Simulator Environment

• **Environment Generation and Realism** LGSVL offers highly detailed urban, suburban, and highway maps, often replicating real-world cities such as San Francisco, Sunnyvale, Seoul, and more. Accurate road geometries (intersections, highways, roundabouts). Realistic traffic control features (traffic lights, road signs). Dynamic traffic agents, pedestrians, cyclists, and other vehicles. Variable weather and lighting conditions (rain, fog, day/night cycles).

• **Scenario Creation and Control** LGSVL provides flexible tools for scripting diverse scenario configurations through Python APIs or the Web-based Scenario Editor as well as control of dynamic elements, including pedestrian movements, traffic densities, vehicle maneuvers, and traffic signal patterns. SIL testing in LGSVL allows AV software modules—such as perception, localization, planning, and control

17

algorithms—to run directly against simulated sensor data. This capability is vital for testing software functionality, performance, and robustness in diverse virtual environments without physical vehicles.

1.5.4 SCANeR Studio

SCANeR Studio is a comprehensive modular simulation platform developed by AVSimulation, widely used across the automotive industry and academia for the development, testing, and validation of autonomous driving systems and Advanced Driver Assistance Systems. It provides high-fidelity simulation capabilities that encompass vehicle dynamics, sensor simulation, real-time traffic behavior, and customizable virtual environments. SCANeR Studio supports all development stages – MIL, SIL, HIL, and Driver-in-the-Loop (DIL) – which makes it a powerful platform for rapid prototyping and system validation under varied driving conditions.

The architecture of SCANeR Studio is modular, where components such as vehicle models, sensors, scenario creation, environment rendering, and traffic generation are decoupled and interconnected, enabling flexibility and scalability for multiple use cases. Some of the key features that distinguish SCANeR Studio as a reliable simulation platform are outlined below:

● **Environment Generation and Realism**
SCANeR Studio offers dynamic and highly customizable environments, including highways, rural roads, and urban centers. The software provides detailed 3D environments with customizable lighting, weather and road friction. It provides realistic road infrastructure including signage, lane markings, traffic signals, and roundabouts, as well as dynamic entities such as AI-driven vehicles, pedestrians, and cyclists.

● **Scenario Creation and Control**
Users can design complex traffic scenarios using a visual scenario editor or scripting languages such as Python. The platform supports:

- Scenario generation for ADAS features like AEB, ACC, LKA, and overtaking.

- Euro NCAP and NHTSA protocol testing templates.

- Real-time control of vehicles, pedestrians, and environmental conditions.

- Integration with external controllers via Simulink, Python, or ROS.

● **Vehicle and Sensor Simulation**
The simulation supports both kinematic and dynamic vehicle models with customizable parameters for mass, tire behavior, suspension, and powertrain. Sensor simulation includes high-fidelity LiDAR, radar, RGB/IR cameras, GPS, ultrasonic sensors, and IMU.
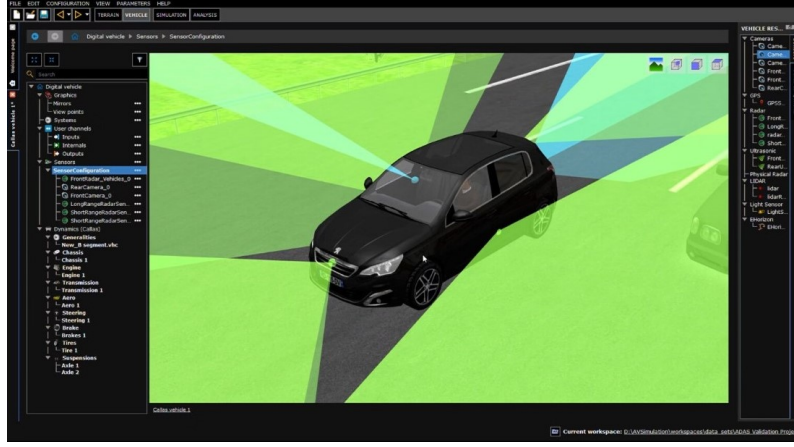
Figure 1.16: SCANeR Environment

Among the currently discussed simulators, Carla is a powerful simulation platform that stands out for several clear reasons, making it an excellent choice for our autonomous driving research. First, its highly realistic environments—with accurate physics, detailed urban settings, and a variety of sensor models—closely mimic real-world conditions. This means that experiments conducted in Carla yield results that are both reliable and applicable to practical scenarios.

Moreover, Carla is open source, which fosters a thriving community of researchers and developers who continuously enhance its features. This collaborative ecosystem ensures that the platform stays current with the latest technological advances while also providing robust support and resources for users. Another significant advantage is its flexibility. Carla's modular design and extensive API allow for easy customization and integration with various machine learning tools, making it adaptable to a wide range of research applications. Additionally, the platform enables consistent and repeatable testing, which is essential for validating experiments and ensuring reproducibility in our studies.

1.6   Thesis Outline

In the development of advanced control systems for autonomous vehicles, ensuring a consistent interface between the high-level algorithms and the vehicle's dynamic response is of high importance. Modern testing environments, such as the CARLA simulator, provide a virtual platform to precisely evaluate these control strategies before real-world implementation.

This thesis aims to develop and validate a comprehensive framework that evaluates control strategies for advanced driver assistance systems (ADAS), recognizing that ADAS plays a pivotal role in enhancing vehicle safety and efficiency by automat-

ing critical driving functions. At the core of this framework is a Nonlinear Model Predictive Control (NMPC) algorithm developed within Simulink. NMPC stands out for its ability to forecast and optimize control actions by considering future vehicle states and environmental conditions, making it ideally suited for managing the inherent nonlinearities of vehicle dynamics. Complementing this, the framework leverages the CARLA simulator as a dynamic data source and implementation environment. CARLA provides high-fidelity simulation of real-world driving scenarios, ranging from typical highway conditions to complex urban settings, which supplies the NMPC controller with realistic sensor data and environmental feedback. This integration creates a closed-loop system where the NMPC in Simulink can be rigorously tested and refined under a wide array of simulated conditions.

To enhance the interface between the control system and the CARLA simulator, a specialized dispatching function was developed. This function serves as an intermediary that converts the acceleration outputs generated by the control algorithms into specific vehicle input commands, namely throttle and brake signals. The design of this function is grounded in vehicle data obtained directly from the CARLA environment, ensuring that its responses accurately reflect the dynamic behavior of the simulated vehicle. By bridging the gap between abstract control outputs and concrete vehicle commands, this approach enables a more faithful and responsive interaction between the control system and the virtual vehicle model.

Successful integration and optimization of this system aim to improve path tracking accuracy, enhance responsiveness to disturbances (like obstacles or sudden route changes), and increase computational efficiency, enabling real-time operation. This approach highlights the effectiveness and practicality of using MATLAB integrated with CARLA for comprehensive autonomous vehicle testing and development.

Chapter 2

Vehicle Models

2.1   Vehicle Dynamics

Vehicle dynamics are the fundamental of any driving system's control strategy and form the basis for understanding and predicting how a vehicle behaves under various driving conditions. Kinematic models are the simplest form of vehicle dynamics models, where they ignore the physical forces acting on a vehicle and instead describe the motion based on geometric and velocity relationships [40]. These models are particularly useful for low-speed maneuvering and basic path planning tasks, where tire forces and inertia effects are negligible. Dynamic models however incorporate physical forces such as tire forces, inertia, slip angles, and friction. These models are more complex and accurate than kinematic models and are suitable for high-speed driving scenarios, such as on highways or during aggressive maneuvers.

When it comes to modeling vehicle dynamics for control and simulation, different methodologies can be adopted depending on the level of complexity required, the nature of the task, and the availability of data. These modeling approaches are generally categorized into white-box, black-box, and grey-box methodologies [41], each representing a different philosophy in how a system's behavior is represented.

White-box model, also referred to as a physics-based or first-principles model, relies entirely on known physical laws, mechanical relationships, and mathematical formulations derived from classical mechanics. These models are constructed by explicitly defining equations that describe how the vehicle responds to inputs like steering, throttle, and braking. Parameters such as mass, wheelbase, center of gravity, and tire stiffness are often either measured or estimated based on manufacturer data or literature. White-box modeling offers a transparent, interpretable structure and is especially suitable for controller design because it provides clear insights into system dynamics and how they affect the vehicle's behavior. Kinematic and dynamic bicycle models, commonly used in path tracking and motion planning algorithms, are good examples of this approach.

In contrast, **Black-box modeling** does not assume prior knowledge of the sys-

tem's internal structure. Instead, it focuses purely on observable input-output relationships, using machine learning or statistical techniques to infer the system's behavior from data. These models are often employed when the physical processes are too complex to model accurately or when the vehicle operates in unstructured or highly variable environments. Neural networks, support vector machines, and regression trees are examples of black-box modeling tools frequently used in autonomous driving applications, especially in data-driven motion planning or end-to-end learning systems. The strength of black-box modeling lies in its flexibility and capacity to approximate highly nonlinear behaviors, though it comes at the cost of interpretability and often requires large datasets for training and validation.
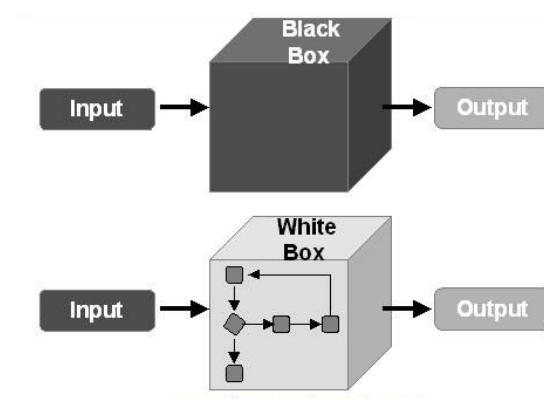
Figure 2.1: White/Black Box Concept

Between these two extremes lies grey-box modeling, a hybrid approach that blends the structure of physics-based models with the flexibility of data-driven refinement. In grey-box modeling, a base model is typically built using physical laws, but certain parameters or subsystems are calibrated or learned from data to enhance accuracy. For instance, a developer might start with a dynamic bicycle model but tune tire stiffness coefficients based on real driving data to better capture cornering behavior under specific conditions. This method provides a balance: it maintains interpretability while allowing the model to adapt to real-world variability, making it especially valuable in practical applications where partial system knowledge is available, but empirical adjustment is necessary.

Another important distinction in modeling approaches relates to the linearity of the system. Linear models are often derived by linearizing a nonlinear system around a fixed operating point, such as driving at a constant speed or along a specific path. These are favored for their simplicity and suitability for classical control techniques like PID or Linear Quadratic Regulator (LQR) [42]. However, their validity is limited to small deviations from the linearization point. In more dynamic or aggressive scenarios, such as overtaking, sharp cornering, or variable speed driving, nonlinear models are required to accurately reflect the system's behavior. Nonlinear modeling, although more complex, enables the use of advanced control strategies like Nonlinear

Model Predictive Control, which dynamically adapts to changing vehicle states and road conditions in real time.

Having discussed both kinematic and dynamic vehicle models, as the underlying methodologies used in their formulation, it becomes evident that the selection of a vehicle model for control design is not only a matter of physical accuracy but also an alignment with the goals and constraints of the application. Kinematic models, while computationally efficient and simple to implement, lack the ability to capture essential dynamic phenomena such as lateral slip, yaw inertia, and tire force saturation; the factors that become increasingly important in high-speed or aggressive maneuvers, such as in highway driving, overtaking, or sharp turning. On the other hand, dynamic models offer a richer representation of vehicle behavior but demand careful parameter identification and an appropriate modeling philosophy. Hence in this work, we adopt a white-box modeling approach by employing a Dynamic Bicycle Model as the predictive model within a Nonlinear Model Predictive Control framework. This choice is motivated by the need to balance physical fidelity and computational tractability.

2.2 Single-Track Model

The single-track model typically assumes that the vehicle is simplified as having two wheels—one at the front and one at the rear—aligned along a central imaginary line [43]. It considers only planar motion, focusing on lateral (sideways) movements and yaw (rotational motion around a vertical axis), while neglecting vertical dynamics such as roll, pitch, and suspension effects. The vehicle is modeled as a rigid body with a fixed geometry, disregarding any deformation or suspension dynamics. Additionally, the single-track model often simplifies longitudinal dynamics by assuming a constant velocity or using basic acceleration models. Tire behavior is usually represented through simplified linear models with small-slip angles, though nonlinear tire models can also be integrated when required [44].
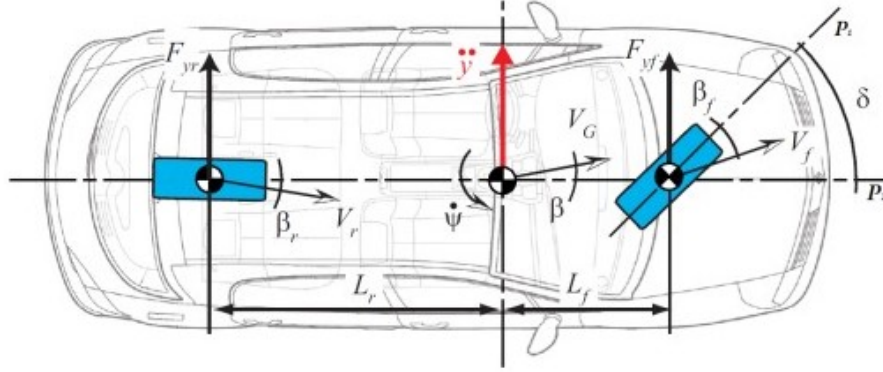
Figure 2.2: Bicycle Model

The Dynamic Bicycle Model is a physically meaningful representation of a vehicle's motion that captures the core lateral and longitudinal dynamics necessary for accurate path following, especially at medium to high speeds. The state of the system usually includes the vehicle's global position (x, y), heading angle ψ, longitudinal velocity v_x, lateral velocity v_y, and yaw rate r.
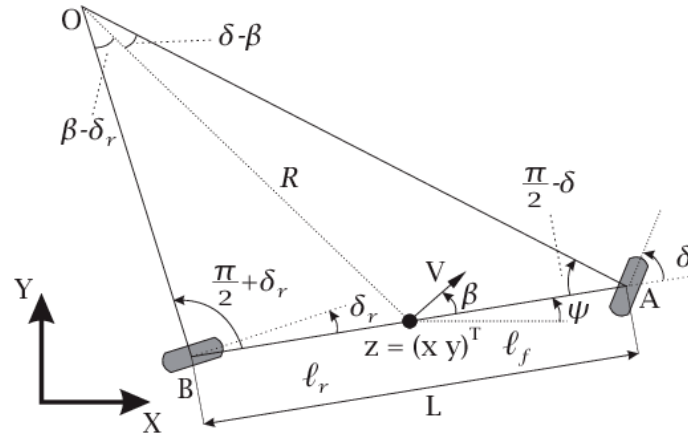
Figure 2.3: Bicycle Model Detailed View

The dynamic bicycle model uses Newton's second law in the lateral and yaw domains [45]. The vehicle states evolve according to the following equations:

$$\dot{x} = v_x \cos(\psi) - v_y \sin(\psi) \tag{2.1}$$

Global x-position rate: The vehicle's forward velocity projected along the x-axis of the global frame, accounting for the heading angle ψ.

$$\dot{y} = v_x \sin(\psi) + v_y \cos(\psi) \tag{2.2}$$

Global y-position rate: Similar to the x-position, this represents the motion in the y-direction in the global frame.

$$\dot{v}_y = \frac{1}{m}(F_{yf} + F_{yr}) - v_x r \tag{2.3}$$

Lateral acceleration: This equation models the vehicle's lateral motion. The total lateral force is the sum of front and rear lateral tire forces, which are divided by the mass m. The term $v_x r$ represents the centripetal acceleration.

$$\dot{r} = \frac{1}{I_z}(L_f F_{yf} - L_r F_{yr}) \tag{2.4}$$

Yaw acceleration: This describes the rotational motion of the vehicle. It calculates the yaw moment from the moment arms L_f and L_r and the lateral forces, scaled by the yaw inertia I_z.

$$\dot{\psi} = r \tag{2.5}$$

Heading angle rate: The heading angle changes at a rate equal to the yaw rate r.

The tire forces themselves are typically modeled as functions of the slip angles, which represent the difference between the direction in which the tire is pointed and the actual direction of travel. In a linear approximation (valid for small slip angles), the tire forces can be expressed as:

$$F_{yf} = -C_{\alpha f}\,\alpha_f, \quad F_{yr} = -C_{\alpha r}\,\alpha_r,$$

where $C_{\alpha f}$ and $C_{\alpha r}$ denote the cornering stiffness coefficients for the front and rear tires, and α_f and α_r are the front and rear slip angles, respectively. These slip angles

can be approximated by:

$$\alpha_f \approx \delta - \arctan\left(\frac{v + L_f\,r}{u}\right), \quad \alpha_r \approx -\arctan\left(\frac{v - L_r\,r}{u}\right),$$

with δ representing the steering angle applied at the front wheels. In many practical applications, the arctangent terms are linearized under the assumption of small angles, which further simplifies the model.

These equations describe the vehicle's lateral velocity (v_y), yaw rate (r), and global position (x, y), as well as its heading angle (ψ) as well as modelling the evolution of the vehicle's lateral motion, yaw rotation, and global position in the inertial (world) frame. They form the basis of the dynamic bicycle model, widely used in control-oriented applications such as Nonlinear Model Predictive Control (NMPC).

From a control design perspective, the dynamic bicycle model is especially advantageous because:

- **Essential vehicle behaviors**: It includes lateral tire slip, oversteering/understeering dynamics, and stability margins—phenomena ignored in simpler kinematic models.

- **Balance between accuracy and efficiency**: While more accurate than kinematic models, it remains computationally light compared to high-fidelity multibody or 3D models, making it suitable for real-time implementation in NMPC loops.

- **Analytically tractable**: The model is nonlinear but structured, which makes it compatible with direct transcription methods used in NMPC solvers.

After deriving the governing equations using Newton's second law and linear tire theory, we arrive at the following set of nonlinear differential equations, which describe the complete state evolution of the system. These equations can be used for control design, trajectory tracking, and vehicle simulation as follows: The state vector is defined as:

$$\mathbf{x} = \begin{bmatrix} X \\ Y \\ \psi \\ V_x \\ V_y \\ \omega_\psi \end{bmatrix} \qquad \text{and the input vector as:} \qquad \mathbf{u} = \begin{bmatrix} a_x \\ \delta \end{bmatrix}$$

$$\dot{X} = V_x \cos \psi - V_y \sin \psi$$

$$\dot{Y} = V_x \sin \psi + V_y \cos \psi$$

$$\dot{\psi} = \omega_\psi$$

$$\dot{V}_x = V_y \omega_\psi + a_x$$

$$\dot{V}_y = -V_x \omega_\psi + \frac{2}{m}(F_{yf} + F_{yr})$$

$$\dot{\omega}_\psi = \frac{2}{J}(l_f F_{yf} - l_r F_{yr})$$

The lateral tire forces are computed using the linear tire model:

$$F_{yf} = -C_f \alpha_f \qquad F_{yr} = -C_r \alpha_r$$

where the slip angles are given by:

$$\alpha_f = \delta - \frac{V_y + l_f \omega_\psi}{V_x} \qquad \alpha_r = -\frac{V_y - l_r \omega_\psi}{V_x}$$

Variable	Description
X, Y	Global position coordinates (in inertial frame)
ψ	Yaw angle (heading)
V_x	Longitudinal velocity (in vehicle frame)
V_y	Lateral velocity (in vehicle frame)
ω_ψ	Yaw rate (angular velocity around vertical axis)
a_x	Longitudinal acceleration input
δ	Front wheel steering angle input
F_{yf}, F_{yr}	Lateral tire forces (front and rear)
m	Vehicle mass
J	Yaw moment of inertia
l_f, l_r	Distance from CG to front and rear axles
C_f, C_r	Cornering stiffness of front and rear tires

However, the DST model does come with limitations. The assumptions of small slip angles and identical behavior between left and right wheels may not hold under extreme driving conditions, high lateral accelerations, or when tire nonlinearities become significant [46]. In such cases, more complex models, such as the double-track model that accounts for individual wheel dynamics and load transfers, might be required for higher fidelity simulations.

2.3 Path Planning and Control Strategies

Path planning is a key component in autonomous driving systems, responsible for determining a feasible and optimal path that an autonomous vehicle can follow from its initial position to a desired destination [47]. This process involves generating trajectories while considering multiple factors, such as road conditions, traffic scenarios, obstacles, vehicle dynamics, safety constraints, and passenger comfort.

In path planning, the vehicle continuously evaluates its surrounding environment through data gathered from sensors, including radar, LiDAR, cameras, and GPS. Based on this information, it predicts future states, identifies potential hazards, and formulates an optimal route. This planning typically involves two main tasks: global and local planning:

Global planning creates an overall route from the origin to the destination using maps and navigation systems, while local planning makes real-time adjustments and precise trajectory refinements to respond to immediate environmental conditions and obstacles. Effective path planning ensures that the vehicle navigates safely and efficiently, maintaining optimal trajectories for both comfort and safety. It also integrates closely with control strategies, such as PID or NMPC, to execute precise maneuvering and trajectory following [48]. The integration of these elements results in reliable and adaptive autonomous driving capable of safely handling diverse and dynamic driving conditions.

PID controllers are classical feedback control mechanisms that continuously calculate the difference between a desired setpoint and the current state of the vehicle. They then apply corrections based on proportional, integral, and derivative terms to minimize this error over time. PID controllers are known for their simplicity, ease of implementation, and computational efficiency [49]. However, they primarily react to current errors without explicitly predicting future states or accounting extensively for system constraints and nonlinear dynamics. As a result, their performance may degrade in complex or highly dynamic driving scenarios. The basic principle behind a PID controller is to continuously compute an error signal, $e(t)$, and then adjust the control input $u(t)$ (for example, the steering angle) according to the following law:

$$u(t) = K_p \, e(t) + K_i \int_0^t e(\tau) \, d\tau + K_d \, \frac{de(t)}{dt},$$

where:

- K_p is the proportional gain, which provides an output that is directly proportional to the current error.

- K_i is the integral gain, which accounts for the accumulation of past errors and helps eliminate steady-state error.

- K_d is the derivative gain, which predicts future error based on its current rate of change, thereby enhancing the system's stability and response.

In a typical path planning application, the error $e(t)$ is defined as the lateral deviation of the vehicle from the reference path. In addition, a heading error may also be incorporated to ensure that the vehicle not only converges to the path but also aligns its orientation correctly. The PID controller computes a corrective steering command that minimizes these errors over time.

The effectiveness of the PID approach in path planning relies on careful tuning of the gains K_p, K_i, and K_d. Proper tuning ensures that the controller provides a swift response to deviations without overshooting or inducing oscillations. This tuning process can be accomplished through empirical methods, simulation-based optimization, or more systematic techniques such as Ziegler–Nichols tuning.

One of the main advantages of using PID control for path planning is its simplicity and low computational cost, making it suitable for real-time applications on platforms with limited processing power [50]. However, its performance is inherently linked to the linearity of the system; in highly nonlinear scenarios or when dealing with large errors, the PID controller may require additional enhancements or gain scheduling to maintain optimal performance.

In contrast, Nonlinear Model Predictive Control (NMPC) significantly enhances autonomous driving performance by explicitly considering the vehicle's nonlinear dynamics and predicting future states and control actions. NMPC operates by formulating and solving an optimization problem at each time step over a finite time horizon, considering a prediction horizon that accounts for vehicle constraints, road curvature, velocity, and potential obstacles [51]. This proactive approach allows NMPC to anticipate future scenarios and optimize the control inputs accordingly. NMPC explicitly considers the nonlinear dynamics of the vehicle and the constraints that govern both state and input variables which makes it especially well-suited in complex or dynamically changing environments. However, it involves greater [52] computational complexity due to solving non-convex optimization problems.

The core principle of NMPC is the repeated solution of an optimal control problem at discrete time intervals. At each step, the current state of the vehicle is measured or estimated, and the future evolution of the system is predicted over a horizon of length N. A discrete-time representation of the vehicle model,

$$\mathbf{x}_{k+1} = f\big(\mathbf{x}_k, \mathbf{u}_k\big),$$

is used to capture the relationship between states and control inputs. The state vector $\mathbf{x}_k$ might include position, heading, velocities, and yaw rate, while the control inputs $\mathbf{u}_k$ typically include steering angle and longitudinal acceleration or throttle commands.

A reference path or trajectory is defined in advance, often specified by a series of waypoints or a continuous function that the vehicle must follow. The NMPC controller then aims to minimize a cost function that measures tracking error relative to this path, while also penalizing large or abrupt control inputs. A typical objective function might include terms for lateral deviation, heading error, velocity tracking,

and control effort:

$$J = \sum_{k=0}^{N-1} \left(w_e\, e_k^2 + w_\psi\, (\psi_k - \psi_{\text{ref},k})^2 + w_u\, \Delta\mathbf{u}_k^2 \right),$$

where e_k is the lateral tracking error, ψ_k and $\psi_{\text{ref},k}$ are the current and reference headings, and $\Delta\mathbf{u}_k$ represents changes in control inputs. The weights w_e, w_ψ, and w_u determine the relative importance of path tracking accuracy versus control smoothness.

An essential feature of NMPC is its ability to incorporate constraints directly into the optimization. These constraints may reflect actuator limits (e.g., maximum steering angle, acceleration, or braking), physical boundaries on velocity, or even collision-avoidance requirements if the environment is known [53]. By embedding these limits into the predictive model, NMPC ensures that all candidate solutions remain within feasible operating bounds, thereby reducing the risk of unsafe maneuvers.

Once the cost function and constraints are defined, a nonlinear optimization solver is employed at each control step to find the optimal sequence of control inputs over the horizon. Only the first set of inputs from this sequence is applied to the vehicle. The horizon then shifts forward, and the process repeats at the next time step. This receding-horizon approach allows the controller to continuously adapt to new information, such as updated state estimates or changes in the reference trajectory.

A key advantage of NMPC for path planning lies in its predictive capability. By simulating how the vehicle will respond to control actions before they are applied, the controller can proactively adjust steering and acceleration to stay close to the desired path [54]. This results in smoother trajectories and improved handling, even under varying road or traffic conditions. However, this comes at the cost of increased computational requirements, as the nonlinear optimization must be solved in real time. Efficient numerical solvers and appropriate discretization strategies—such as direct collocation or multiple shooting—are therefore critical for practical implementations [47][55][56].

Chapter 3

CARLA Simulator

3.1 CARLA Features and Architecture

The CARLA simulator is an open-source urban driving simulator designed to advance research in autonomous driving. It is actually based on a client-server architecture, wherein the server handles rendering and simulation tasks such as sensor data generation, physics calculations, weather conditions and vehicle dynamics while the client manages agent-related logic such as defining how the simulation evolves-defining scenarios and implementing control algorithms-within the virtual environment. Communication between these components is facilitated through a Python API to incorporate Python scripting into the simulation. This modular design allows for experimentation and in-depth evaluation of different autonomous driving algorithms within realistic traffic and environmental scenarios. Unreal Engine 4 (UE4) serves as the fundamental rendering and physics platform for the CARLA simulator, providing the real-time graphical and dynamic physical interactions required for high-fidelity autonomous driving research, providing the groundwork for accurate motion modeling, encompassing vehicle dynamics such as friction, collisions, and inertial forces that mirror real-world scenarios.

Regarding the Configuration employed for the vehicles, it has to be noted that CARLA employs a left-handed coordinate system with the z-axis pointing upward. This configuration governs the representation of vehicle positions, orientations, and the surrounding environment. Using Python API we can customize the virtual environment and seamlessly extract simulation data. By running Python scripts in parallel with the active simulation, one can instantiate autonomous vehicles, configure the surrounding setting, and retrieve sensor outputs for further analysis. This approach makes it possible to feed real-time data directly into lateral control algorithms, thereby realizing a fully automated control loop. In doing so, one can efficiently develop, test, and refine autonomous driving solutions within a reliable simulation framework. The Pictures below present various snapshots of town10, the main urban-structured map within the simulator. This environment features both dense urban areas characterized by narrow streets, intersections as well and also areas with multiple lanes along with different weather and daylight setting.

Such diversity allows users evaluate the performance of their driving models and algorithms under a wide range of urban scenarios and road conditions.

Figure 3.1: Carla Street View 1

Figure 3.2: Carla Street View 2

Figure 3.3: Carla Street View 2-rainy weather

The main element of CARLA's simulation framework is the World object, which functions as an abstraction layer for overseeing the environment, spawning actors, and adjusting simulation parameters such as weather conditions and traffic behavior. Moreover, CARLA provides a Traffic Manager module to govern vehicle behavior in autopilot mode, a high-level control system that dictates how AI-controlled vehicles interact within the simulated environment. This would also enable users to personalize traffic dynamics by fine-tuning factors like speed limits, and compliance with traffic regulations. Users can also choose between deterministic traffic patterns, where all vehicles follow strict rules, or stochastic behavior, where vehicles exhibit variations in driving styles, introducing uncertainty into the simulation. Parameters of blueprint vehicles are also specified by the client through the CARLA client API, allowing for flexible configuration of each sensor's 3D position, orientation relative to the vehicle's coordinate system, field of view (FOV), and sensing depth, and generally favourable properties for each simulated vehicle. Additionally, CARLA provides precise data on the locations and bounding boxes of all dynamic objects, including pedestrians, bicycles, and other moving entities in the environment, a feature essential for training and assessing different algorithms. In Figure 3.4, the Traffic Manager architecture inside CARLA is shown. As explained above it acts as an intermediary between the simulation state and the control logic, namely making decisions for vehicles in autopilot mode. The Agent Lifecycle State Management (ALSM) component continuously scans the environment to monitor the position, velocity, and movement of all vehicles and pedestrians. In other words, it ensures real-time updates of vehicle states while removing inactive entities from the registry (an array of vehicles and pedestrians). The simulation state part is a cache store of the position, velocity, and additional information of all the vehicles and pedestrians in the simulation. We briefly explain the calculation strategy of movements of each vehicle in the autopilot mode:

Localization Vehicles plan their route dynamically using waypoints stored in a simplified map grid. At intersections, paths are chosen randomly to simulate real-world behavior.

Collision Detection Bounding boxes are placed along the vehicle's trajectory to detect and avoid potential crashes.

Traffic Light Handling Vehicles obey stop signs, junction priority rules, and traffic signals based on their position in the environment.

Motion Planning The vehicle's speed and direction are determined using a PID controller, ensuring smooth and efficient movement.

Vehicle Lights Control Headlights, brake lights, and turn signals are activated based on environmental conditions (e.g., rain, fog) and driving behavior (e.g., braking, turning). Therefore, the generated commands are aggregated into a command array and transmitted to the CARLA server as a batch, ensuring synchronized execution within the same simulation frame for consistent vehicle behavior.
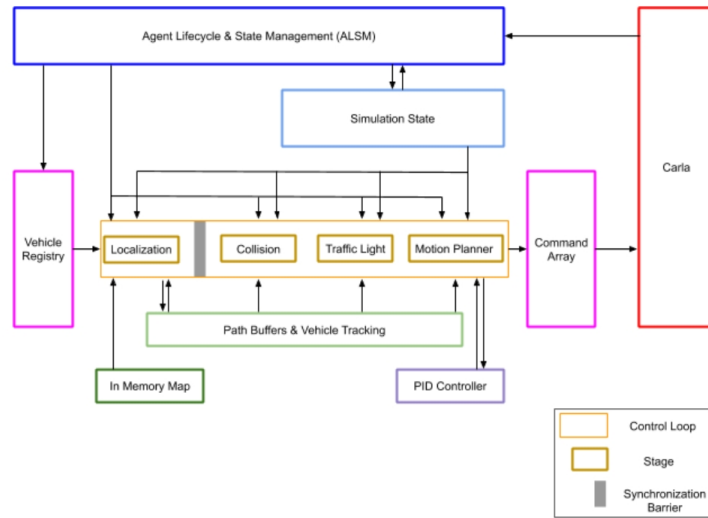
Figure 3.4: Traffic Manager Architecture (courtesy of CARLA)

3.2   Anaconda Interface

Anaconda features a structured environment for running CARLA simulations, ensuring that all required dependencies are installed and managed without conflicts. By leveraging Anaconda, one can establish virtual environments tailored to CARLA, simplifying the process of handling multiple Python versions and associated packages. This approach basically reduces compatibility issues and supports integration

with CARLA's Python API. Additionally, Anaconda includes development tools such as Spyder and Jupyter Notebook, which streamline the workflow for writing, testing, and debugging simulation scrips.

Figure 3.5: Integration Environment of Carla and Python using Anaconda

Hence for a proper interaction between Anaconda, Python, and CARLA an Anaconda virtual environment is created and configured with the appropriate Python version and necessary dependencies. Subsequently, the CARLA Python API is integrated into this environment, enabling communication with the CARLA server. Once this setup is complete, one can utilize Python scripts to control vehicles, process sensor outputs, and model complex driving scenarios. The following are the corresponding versions of software and libraries used in this project making of python and CARLA integrated, allowing direct control of the simulation environment:

- CARLA Installation (Version 0.9.14)

- Python version 3.7

- Python libraries: carla, pygame, numpy, opencv

Once these steps are completed, Python and CARLA become integrated, allowing direct control of the simulation environment. As such we can establish multiple virtual environments dedicated to CARLA, enabling parallel development and testing of different configurations or dependencies without the risk of library conflicts.

It is important to note that Pygame serves as the foundational framework for CARLA's graphical interface and its basic functionality. It is actually a Python-based library and widely used for rendering 2D graphics, handling events, and managing user interactions which makes it a suitable choice for CARLA's visualization and interaction components. Therefore it is used to create a display window for the simulation. This window can show live camera feeds from the simulated sensors, such as front-facing cameras or LiDAR visualizations, and also includes a heads-up display (HUD) that presents critical information like speed, location, and simulation frame rate. This visual interface is essential for users to observe the simulation in real time

and assess the performance of different control algorithms. By employing Pygame's event-driven framework, CARLA can efficiently handle both user-generated events and system-generated updates. This ensures that the simulation remains responsive, allowing real-time adjustments to the simulation parameters and control inputs, which is especially valuable when evaluating autonomous driving algorithms such as PID controllers or NMPC. Below is an example of the CARLA interface, showcasing real-time sensor data and vehicle information displayed within the simulation:

Figure 3.6: CARLA Simulation Interface using Pygame

As shown in Figure 3.6 various sensors are displayed in the default mode to provide real-time feedback on the vehicle's perception of its surroundings. the sensors can be selectively specified sensors based on user preference. As an example, below a snippet of activating these sensors are provided:

```
import carla

client = carla.Client('localhost', 2000)
world = client.get_world()

blueprint_library = world.get_blueprint_library()
vehicle_bp = blueprint_library.filter('vehicle.model3')[0]
spawn_point = world.get_map().get_spawn_points()[0]
vehicle = world.spawn_actor(vehicle_bp, spawn_point)

camera_bp = blueprint_library.find('sensor.camera.rgb')
lidar_bp = blueprint_library.find('sensor.lidar.ray_cast')

camera_transform = carla.Transform(carla.Location(x, z))
camera = world.spawn_actor(camera_bp, camera_transform, attach_to=
    vehicle)

```

```
17  lidar_transform = carla.Transform(carla.Location(x=0, z=2.5))
18  lidar = world.spawn_actor(lidar_bp, lidar_transform, attach_to=
       vehicle)
19
20  sensors = [camera, lidar]
```

<div align="center">Listing 3.1: World and Sensor Definition</div>

3.3   Carla Autonomous mode

CARLA simulator features a built-in autonomous driving mode, known as CARLA Autopilot, which employs a rule-based control approach. This mode is designed primarily for autonomous trajectory following. The methodology of the controller is Proportional-Integral-Derivative (PID) control strategy to autonomously drive a vehicle along a predefined or dynamically computed reference trajectory. The PID controller in CARLA consists of two main parts:

Longitudinal Controller (Speed Control) manages the speed of the vehicle, maintaining or adjusting it according to a predefined reference speed. Using PID calculations, the controller adjusts throttle and braking signals to minimize the error which is defined as the difference between the target speed (reference velocity) and the actual vehicle speed. Accordingly, this part of the controller ensures smooth acceleration and braking, aiming for minimal overshoot and steady cruising speeds.

Lateral Controller (Steering Control) controls the steering to ensure the vehicle closely tracks the desired trajectory, primarily by managing the steering angle in order to correct deviations promptly and smoothly. Similar to the longitudinal controller, the integral and derivative terms in this case correct accumulated offsets and dampen steering responses, respectively.

This implemented controller inside CARLA simulator offers a straightforward solution for vehicle trajectory tracking and is suitable for controlled scenarios. However, it can sometimes be less reliable and more prone to errors compared to advanced methods like Nonlinear Model Predictive Control for several reasons. Firstly, it does not anticipate future states or situations, which are frequently encountered in real environments. Additionally, it may struggle during complex maneuvers involving lane changes, sudden obstacle avoidance, sharp turns, or changing road conditions. Finally, it is highly sensitive to tuning parameters of the PID gains. In other words, parameters tuned for one scenario may not be generalized well to other scenarios. In contrast, NMPC can explicitly account for such disturbances within its predictive model, offering improved robustness to real conditions and environment in reality.

3.4 Carla and MATLAB Integration

The integration between Simulink and CARLA is essential for developing, testing, and validating autonomous vehicle control strategies in a high-fidelity simulation environment.

Figure 3.7: CARLA Matlab Interface using Python

This connection is facilitated through MATLAB's Function block, which acts as an interface for data exchange between Simulink and CARLA's Python API. By establishing a real-time communication link, Simulink can control the ego vehicle within CARLA while simultaneously receiving feedback on its motion, allowing for closed-loop trajectory tracking and controller evaluation. To be more exact, this connection allows access to the CARLA world and manipulation of simulation parameters.

Looking at the schema below, inputs and outputs to the corresponding Block, it ensures a continuous exchange of data between the controller and the simulation environment; Simulink sends control commands (throttle, brake, and steering angle) to CARLA, while CARLA returns vehicle state feedback (position, velocity, yaw) to Simulink.

```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)
```

```matlab
17                car;
18          end
19
20      methods(Access = protected)
21          function setupImpl(obj)
22              % Perform one-time calculations, such as computing
                    constants
23              port = int16(2000);
24              client = py.carla.Client('localhost', port);
25              client.set_timeout(10.0);
26              world = client.get_world();
27              %edited_11_March
28              blueprint_library = world.get_blueprint_library();
29 %              cup_car = py.list(blueprint_library.filter("
   charger_police"));
30 %              static_object = cup_car{1};
31             spawn_points = world.get_map().get_spawn_points() ;
32 %              spawn_location2 = spawn_points{303};
33 %              spawn_location2.location.x = 180.43829345703125;
34 %              spawn_location2.location.y = 39.93829345703125;
35 %              spawn_location2.rotation.yaw = -45;
36 %              static_car = world.spawn_actor(static_object,
   spawn_location2);
37
38 %              control2 = py.carla.VehicleControl(throttle = 0,
   steer = 0, brake = 1, hand_brake = True);
39   %              control2 = py.carla.VehicleControl(throttle = 0, steer
     = 0, brake = 1);
40 %              control2 = static_car.get_control();
41 %              control2.brake = 1;
42 %              static_car.apply_control(control2);
43
44
45
46              % Spawn Vehicle
47
48              car_list = py.list(blueprint_library.filter("leon"));
49              car_bp = car_list{1};
50              %spawn_point = py.random.choice(world.get_map().
                    get_spawn_points());
51
52 %              spawn_point =  world.get_map().get_spawn_points();
53 %              spawn_location = spawn_point{8};
54
55
56              %%% Spawn_point transformasyon matrisini olu tur
                    sonras nda
57         %%%   istedi in konum de erleri manual olarak de i tir
58          spawn_location = spawn_points{303}
59          spawn_location.location.x = 50.33776092529297
60          spawn_location.location.y = 37.75230407714844;
61          %spawn_point.location.z = 0.6;
62          spawn_location.rotation.yaw = 0.017;
```

```
63
64                obj.car = world.spawn_actor(car_bp, spawn_location); %%
                      spawn the car
65                %obj.car.set_autopilot(false)
66
67
68            end
69
70            function [ze,x_acceleration] = stepImpl(obj,
                  steeringangle_input,throttle_input,brake_input)
71                %
72 %                   pause(0.0001);
73
74                vehicle_transform = obj.car.get_transform();
75                orientation = vehicle_transform.rotation;
76
77                x_position = obj.car.get_location().x;
78                y_position = obj.car.get_location().y;
79                x_velocity = obj.car.get_velocity().x;
80                y_velocity = obj.car.get_velocity().y;
81                w = obj.car.get_angular_velocity().z*pi/180;
82
83                x_acceleration = obj.car.get_acceleration().x;
84
85  % Simulink'ten gelen steering angle de erini carla arac na
        uygula
86      control = obj.car.get_control();
87       control.steer = rad2deg(steeringangle_input)/70;
88 %         control.steer = 0;
89 % acceleration_input = ((acceleration_input+10)*2/13)-1;
90 %     % E er accelaration_input s f rdan k     kse , fren yap
91 % if acceleration_input < 0
92 %     % Control.brake de erini 0 ile -acceleration_input aras nda
        olacak   ekilde   ayarla
93 %       control.brake = abs(acceleration_input);
94 %       control.throttle = 0; % E er fren yap yorsak , throttle'
       s f ra ayarla
95 % else
96 %     % E er accelaration_input s f rdan b y kse , throttle yap
97 %     % Control.throttle de erini 0 ile acceleration_input
       aras nda olacak   ekilde   ayarla
98 %       control.throttle = acceleration_input;
99 %       control.brake = 0; % E er gaz yap yorsak , freni s f ra
       ayarla
100 % end
101                control.throttle = throttle_input;
102                control.brake = brake_input;
103                yaw_angle = double(deg2rad(orientation.yaw));
104 %              yaw_now = yaw_angle;
105 %              w = (yaw_angle - yaw_old)/0.3 + w_old; %% angular
       velocity
106 %              w_now=w;
107                ze = [x_position,y_position,yaw_angle,x_velocity,
```

```matlab
                    y_velocity,w]';

            obj.car.apply_control(control);

        end

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

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

        end

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

        end

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

Listing 3.2: Carla Environment NMPC Class

3.5   Data Gathering

Therefore, it is essential to obtain direct access to the vehicle's data and access localization features and data are first introduced, highlighting the distinction between

the map frame and the vehicle frame (centered at the vehicle's rear axle). By using CARLA's Python API methods, including `get_location()`, and `get_transform()`, one can track both the X–Y coordinates of the vehicle blueprint in meters (particularly relevant on flat terrain) and the yaw angle, which is critical for curved routes.

Additionally, dynamic vehicle parameters—such as lateral and longitudinal velocities (in meters per second), as well as yaw rate—can be accessed using the methods `get_acceleration()` and `get_velocity()`.

Our objective here involves gathering data from the ego vehicle, which can be operated both under CARLA's autonomous mode via the traffic manager and Manual mode by a human driver. This dataset is then imported into MATLAB for analyzing vehicle dynamics and assessing the models. A high-fidelity driving setup was utilized in order to capture the corresponding manual driving data. This configuration, as shown in Figure 3.8, includes a Logitech G29 Driving Force steering wheel and pedal set combined with a Sparco sport seat, available in the department lab. This setup provided a realistic driving experience, allowing a human driver to navigate simulated highway scenarios. By recording steering control inputs as well as controlling the pedals (throttle and braking), we have collected data thorught the simulation environment manually as a benchmark for evaluating human driving behavior against both CARLA's built-in autopilot and the implemented Nonlinear Model Predictive Control (NMPC) system.

Figure 3.8: Manual Driving Setup

An essential aspect of data collection in the simulation is the sampling rate, which determines the time interval at which vehicle state information is recorded. Selecting an appropriate sampling rate for balancing data accuracy and computational efficiency, based on the scenario chosen, would be crucial. Once the scenario and ego vehicle are initialized based on predefined parameters, data recording is performed throughout the simulation. The duration of the data collection process is determined by the vehicle's motion and the constraints given in the scenario. At each time step, key vehicle state parameters are logged for subsequent analysis.

Below we present a brief snippet of our implementation to provide clearer insight into the methods and logics employed. However, for further details regarding the methodologies capturing data from the simulation, including additional Python scripts and clarification, please refer to the Appendix A.

```
steering_data = []

location_data_x = []
location_data_y = []
location_data_z = []
velocity_data_x = []
velocity_data_y = []
angular_velocity = []
yaw_data = []
acceleration_data_x = []
acceleration_data_y = []
throttle_data = []
brake_data = []
steering = []

class World(object):
    def __init__(self, carla_world, hud, actor_filter):
        self.world = carla_world
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = actor_filter
        self.restart()
        self.world.on_tick(hud.on_world_tick)

    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
```

```
39        location = self.player.get_location()
40        velocity = self.player.get_velocity()
41        transform = self.player.get_transform()
42        yaw = transform.rotation.yaw
43        yaw_rate = self.player.get_angular_velocity().z
44        yaw_rate = yaw_rate * math.pi/180
45        yaw = yaw * math.pi/180
```

Listing 3.3: Manual Data Gathering

Chapter 4

Control Design Strategies

4.1 Overview of the project

In this thesis, the primary goal is to guide a vehicle along a predetermined trajectory while optimizing key performance indicators, namely minimizing cross-track error (the lateral deviation from the reference trajectory) and reducing heading error (the angular discrepancy between the vehicle's orientation and the desired heading). At the same time, the system aims to maintain stability and ensure efficient driving performance throughout the journey. Unlike simpler controllers, NMPC can integrate and manage the trade-offs between tracking precision and actuator effort systematically.

4.2 Vehicle Model

In the dynamic single-track model, the motion of the vehicle is described using quantities like longitudinal velocity, lateral velocity, yaw rate, and side-slip angle. The inputs to the system are the steering angle and the longitudinal force, which are directly related to the driver's or controller's commands. Tire forces are modeled using simplified linear tire models, which relate the lateral forces to the slip angles at the front and rear tires. In this study, this model serves as the internal prediction model for the NMPC controller. Based on the current state of the vehicle — such as its position, speed, and orientation — the controller uses the dynamic model to simulate how the car would move in the next few steps if certain control inputs were applied. It then selects the control actions that best keep the vehicle on the desired path while respecting physical and safety constraints.

Among the various vehicle models available in the CARLA simulator, the SEAT Leon has been selected for this study. The reason for this choice is that the SEAT Leon represents a typical passenger vehicle in terms of size, dynamics, and performance characteristics.

The SEAT Leon vehicle in the CARLA simulator is modeled using a set of physical parameters that influence its motion behavior. These parameters are accessed

via the `vehicle.get_physics_control()` method and are used in the bicycle model for our control and subsequent simulation. Figure 4.1 illustrates the vehicle model employed in the simulation and control methodologies employed in this study. This vehicle serves as the primary agent in the CARLA simulator of our experiment and is equipped with the necessary sensors and dynamic properties to accurately test and evaluate the performance of manual driving, autopilot, and NMPC controllers.

Parameter	Value
Vehicle Mass, m	1318 kg
Yaw Moment of Inertia, J	2500 kg·m^2
Distance to Front Axle, L_f	1.54 m
Distance to Rear Axle, L_r	1.51 m
Front Cornering Stiffness, C_f	15000 N/rad
Rear Cornering Stiffness, C_r	15000 N/rad

Table 4.1: Seat Leon Physical Parameters in CARLA

Figure 4.1: The Seat Leon used for the Simulation

4.3 NMPC Path Tracking

Nonlinear Model Predictive Control (NMPC) is a feedback control methodology characterized by the on-line solution of moving-horizon optimal control problems

(OCPs). At each control interval, NMPC predicts future system states over a finite horizon using a nonlinear model of the process or plant, and it determines the optimal control actions (e.g., steering angle, throttle, brake) by minimizing a cost function subject to state and input constraints. This approach has gained significant traction in various fields—including automotive, aerospace, chemical processes, and robotics—due to its flexibility in handling both nonlinear dynamics and complex constraints.

One of the core strengths of NMPC is its ability to manage systematically the trade-off between performance and control effort while also respecting input/state/output constraints. Conceptually, NMPC is often viewed as a nonlinear, finite-horizon version of the Linear Quadratic Regulator (LQR), offering a more general framework for systems with pronounced nonlinearities. In automotive applications, NMPC is particularly appealing due to the nonlinear dynamics of vehicles and the need to incorporate safety-critical constraints such as road boundaries, obstacle avoidance, and actuator limits. The predictive nature of NMPC allows it to account for future trajectory deviations and proactively adjust steering, throttle, and braking commands.

The prediction model used in NMPC is the dynamic single-track model described earlier. The system can be represented in a nonlinear discrete-time state-space form as follows:

$$x_{k+1} = f(x_k, u_k)$$

where:

- $x_k \in \mathbb{R}^n$ is the state vector at time step k,

- $u_k \in \mathbb{R}^m$ is the control input vector,

- $f(\cdot)$ represents the nonlinear vehicle dynamics.

The state vector typically includes:

$$x = \begin{bmatrix} X & Y & \psi & v & r & \beta \end{bmatrix}^T$$

where X, Y are global positions, ψ is the yaw angle, v is the longitudinal velocity, r is the yaw rate, and β is the side-slip angle.

The control input vector is defined as:

$$u = \begin{bmatrix} \delta & a \end{bmatrix}^T$$

where δ is the steering angle and a is the longitudinal acceleration.

At each time step, NMPC solves the following constrained optimization problem:

$$\min_{u_0, \ldots, u_{N-1}} \sum_{k=0}^{N-1} \left[(x_k - x_k^{\text{ref}})^T Q(x_k - x_k^{\text{ref}}) + (u_k - u_k^{\text{ref}})^T R(u_k - u_k^{\text{ref}}) \right]$$
$$+ (x_N - x_N^{\text{ref}})^T Q_f(x_N - x_N^{\text{ref}})$$

subject to:

$$x_{k+1} = f(x_k, u_k), \quad k = 0, \ldots, N-1$$
$$x_k \in \mathcal{X}, \quad u_k \in \mathcal{U}, \quad \forall k$$
$$x_0 = x_{\text{current}}$$

where:

- N is the prediction horizon,

- Q, R, Q_f are positive semi-definite weighting matrices,

- $x_k^{\text{ref}}, u_k^{\text{ref}}$ are reference trajectories for state and control,

- $\mathcal{X}, \mathcal{U}$ define allowable bounds for states and controls.

In practice, the optimization problem is solved numerically at each time step using nonlinear programming (NLP) solvers. The optimization yields the optimal control input sequence, but only the first input is applied to the system. This process is repeated at the next time step in a receding horizon fashion.

The control scheme operates in closed loop, with real-time feedback from the vehicle's current state, allowing it to handle disturbances and uncertainties effectively. The prediction model is updated at each iteration using the dynamic single-track equations discretized using methods such as Euler or Runge-Kutta integration.

In the formulation of Nonlinear Model Predictive Control (NMPC), both the temporal discretization and the weighting of different components in the cost function play a central role in the behavior and performance of the controller.

The sampling time T_s refers to the time interval at which the control algorithm is updated. At each step, the current state of the vehicle is measured, and an optimization problem is solved to compute the optimal control input. A smaller T_s allows for finer time resolution and better responsiveness to fast changes in the vehicle state or environment. However, it also increases the computational load, which may be critical in real-time implementations. The vehicle model used in the NMPC framework is discretized according to T_s, affecting the accuracy and stability of the numerical predictions.

The prediction horizon T_p defines how far into the future the controller predicts the system's behavior. It is related to the sampling time via the number of prediction steps N as:

$$T_p = N \cdot T_s$$

A longer prediction horizon enables the controller to anticipate upcoming events, such as curves or obstacles, more effectively. However, this comes at the cost of

increased computational complexity. In practice, a trade-off must be made between long-term planning and real-time feasibility.

The NMPC controller minimizes a cost function that balances the objectives of state tracking and control smoothness. The general form of the cost function is:

$$J = \sum_{k=0}^{N-1} \left[(x_k - x_k^{\text{ref}})^T Q (x_k - x_k^{\text{ref}}) + (u_k - u_k^{\text{ref}})^T R (u_k - u_k^{\text{ref}}) \right] + (x_N - x_N^{\text{ref}})^T P (x_N - x_N^{\text{ref}})$$

Here:

- $Q \in \mathbb{R}^{n \times n}$ is the state weighting matrix. It penalizes deviations of the vehicle's actual state from the reference trajectory. Higher values in Q indicate that precise tracking is a priority.

- $R \in \mathbb{R}^{m \times m}$ is the control input weighting matrix. It penalizes excessive or abrupt control actions. Higher values in R result in smoother, more conservative control.

- $P \in \mathbb{R}^{n \times n}$ is the terminal state weighting matrix. It encourages convergence of the final predicted state to the desired reference.

Proper tuning of these matrices is critical. Emphasizing Q results in accurate path following, while emphasizing R leads to comfortable, energy-efficient driving. The terminal weight P, often chosen as the solution to the discrete-time Riccati equation in the linearized case, enhances the controller's long-term behavior and ensures stability at the end of the prediction horizon.

The control section focuses on identifying appropriate parameter values for the NMPC controller once the underlying vehicle model has been established. Accurate tuning of these parameters is crucial for achieving the desired vehicle behavior with optimal precision, stability, and responsiveness. For the NMPC method implemented in this study, the initial control parameters were derived based solely on the vehicle dynamics modeled in Simulink.

To establish a reliable reference for the controller, steering angle (δ_f) and longitudinal acceleration (a_x) signals were collected from a manual driving session within the Simulink environment. These reference trajectories allowed for preliminary tuning of the NMPC parameters, such as the weighting matrices Q, R, and P, prediction horizon, and input bounds. While the Simulink-based vehicle model closely resembles the SEAT Leon model in CARLA, they are not identical due to differences in underlying dynamics, friction models, and actuator behaviors.

Through comparative analysis, it became evident that better tuning results could be obtained by integrating feedback from the CARLA simulator. This hybrid approach—tuning the controller in Simulink while validating it in CARLA—enabled

more realistic parameter identification. It also revealed discrepancies between theoretical predictions and real-time simulation behavior, particularly during aggressive maneuvers or curved path tracking.

When readjusting the parameters, the objective was to identify a configuration that generalizes well across different simulation scenarios, including both straight and curved highway paths. In addition to minimizing lateral and longitudinal tracking error, another key objective was to enable the vehicle to travel at higher average speeds while remaining within safe limits for control effort and physical constraints. This meant not only tuning the matrices but also ensuring that input constraints, such as acceleration and steering bounds, were carefully selected based on CARLA's vehicle dynamics.

Ultimately, the NMPC parameters were iteratively optimized to balance three main goals: high-fidelity path tracking, minimal control effort, and adaptability to both ideal and non-ideal road geometries. This iterative loop between Simulink-based tuning and CARLA-based validation proved essential in developing a controller that performs reliably across diverse driving conditions.

The results are based on the following weighting matrices and parameters:

Table 4.2: NMPC Parameters and Weight Matrices

Parameter	Value
T_s	0.05
T_p	3
n	6
Q	$1000 \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
P	$100 \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
R	$\begin{bmatrix} 0.1 & 0 \\ 0 & 1 \end{bmatrix}$

4.4   NMPC Model in Simulink

The following figure 4.2 presents the Simulink implementation of the Nonlinear Model Predictive Control (NMPC) framework, which is designed for autonomous vehicle path tracking and maneuver execution. The model consists of interconnected blocks that represent the various components of the control system, including the vehicle dynamics, reference trajectory generation, NMPC controller, and feedback loops.
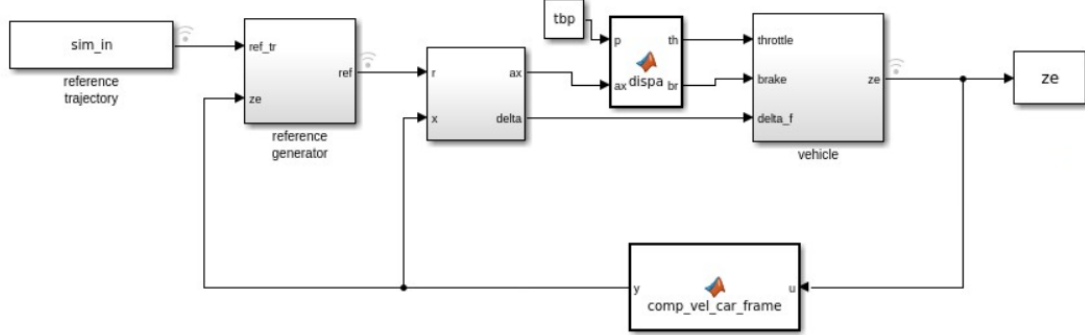
Figure 4.2: NMPC Simulink Model

The `sim_in` block provides the desired trajectory that the autonomous vehicle aims to follow. This trajectory typically includes position, speed, or other relevant reference signals necessary for trajectory tracking.

The **reference generator** block processes the input trajectory from `sim_in` and transforms it into a suitable format for the NMPC controller. It outputs structured reference data, typically positions and velocities, required for accurate path tracking. In the next chapter we examine the reference trajectory block, which generates the desired path for the vehicle and serves as a key input to the NMPC controller.

The `tbp` block translates the reference trajectory into actionable signals for the NMPC controller. It generates key dynamic references such as longitudinal acceleration (a_x) and steering angle (δ) required to guide the predictive controller's decision-making.

The `dispa` block represents the Nonlinear Model Predictive Controller, which computes the optimal control actions. It minimizes the tracking error between the predicted vehicle states and the desired reference trajectory, considering vehicle dynamics and constraints. Its outputs include throttle (`th`) and brake commands (`br`).

The `vehicle` block simulates the vehicle's physical behavior, taking throttle, brake commands, and steering inputs (δ_f) as inputs. It outputs the resultant vehicle states (`ze`), reflecting realistic vehicle response to the control inputs. `ze` consists of 6 output variables: x, y, ψ (yaw), v_x, v_y, and r (yaw rate).

The feedback loops involving the `ax` and `delta_f` blocks capture real-time vehicle states and provide continuous state updates to the NMPC controller. This feedback mechanism ensures that the controller can adapt dynamically to the ac-

tual performance of the vehicle.

4.5 Reference Generator

This function serves as a critical component within a closed-loop path tracking control system. From a systems engineering point of view, its primary role is to bridge the gap between the global reference trajectory (a pre-recorded or planned path) and the local trajectory that the vehicle must track over a finite prediction horizon. The Parameters used here are as follows:

Np (Prediction Horizon Length): Defines the number of future reference points to extract. This parameter is directly tied to the preview horizon of the control algorithm. A larger Np allows the vehicle to anticipate curves or obstacles earlier but requires more computational effort and accurate vehicle modeling.

ref_tr (Reference Trajectory): A pre-defined set of waypoints, where each row consists of the global x and y coordinates. This serves as the high-level route that the vehicle should follow.

ze (Current Vehicle Pose): Contains the current x and y position of the vehicle, as well as its heading angle (yaw). This is the real-time input from the localization system or vehicle state estimator.

One of the fundamental tasks in trajectory following is aligning the vehicle's local frame with the global reference. The function achieves this by computing the Euclidean distance between the current vehicle position and every point on the reference path, using vector norms. This ensures that the most relevant point on the path—i.e., the one closest to the vehicle—is identified.

This closest point acts as an anchor: the starting point from which future trajectory points will be selected. The use of norm-based distance ensures independence from coordinate system orientation, making the method robust and generalizable.

After identifying the closest point c, the function proceeds to extract a forward-looking segment of the path that spans Np future waypoints. The use of the `linspace` function ensures that evenly spaced samples are selected between the current index and the future index $(c + Np - 1)$. This approach guarantees temporal and spatial smoothness in the reference path, which helps in avoiding erratic control actions caused by abrupt changes in trajectory curvature.

The parameter `NS` defines the resolution of this segment and allows for interpolation-like sampling even in discretely spaced paths.

Creating the Reference Vector for the Controller

Once the relevant portion of the path is selected, the x and y coordinates are reshaped into a single column vector that serves as the control reference input. This

format is particularly suitable for control blocks in Simulink that expect a vectorized input.

The output reference vector enables the control algorithm to plan motion over the short horizon using accurate predictions of where the vehicle should be at each time step. Below is the detailed script of this implementation:

```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);
```

Listing 4.1: Reference Generator

4.6 Dispatching

In the context of control systems, "dispatching" refers to the systematic process of allocating or converting one form of control command into another set of actuator-specific commands. Fundamentally, it is about taking a generalized control output—in our case, a longitudinal acceleration command and distributing it into appropriate throttle and brake commands that the vehicle can physically execute.

The philosophy behind dispatching is rooted in the idea of resource and task allocation. Just as a dispatcher in logistics or telecommunications directs resources where they are most needed, a dispatching algorithm in vehicle control assigns the correct actuation signals based on predefined criteria. These criteria are often established from empirical reference data and reflect the vehicle's dynamic capabilities under various conditions.

In autonomous vehicle control systems, precise regulation of vehicle speed and dynamics relies significantly on accurately managing the throttle (acceleration) and braking inputs. To achieve this, a specialized dispatching function is employed to translate desired acceleration inputs into actual throttle and brake commands. This function ensures that acceleration and braking demands provided by path planners or controllers are effectively converted into actionable commands that match the

vehicle's current gear, performance capabilities, and dynamic constraints. For instance, a negative acceleration may not always require active braking; sometimes, reducing the throttle is sufficient to achieve the desired deceleration. The process, therefore, includes weighting mechanisms that decide which control input to prioritize, ensuring smooth and efficient vehicle operation.

```matlab
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


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
```

Listing 4.2: Dispatching Function

4.7 Performance Metrics

To better understand the model's performance, we rely on metrics that measure how effectively the controller maintains the vehicle's desired path and orientation in the highway scenario. Two primary indicators, namely Cross-Track Error (CTE) and Heading Error, which serve as key measures in evaluating autonomous driving performance, are here being employed:

Cross-track error (CTE) measures the lateral displacement of the vehicle from the desired path or lane centerline. In highway driving, maintaining low lateral deviation is critical for safety, as excessive drift could lead to unintended lane departures on adjacent lanes. The objective here is that NMPC, with its predictive horizon and ability to account for various constraints, excels at minimizing CTE by adjusting control inputs based on future states.

Heading error, on the other hand, quantifies the difference between the vehicle's current orientation and the reference heading. This index is especially relevant when negotiating curves or merging into different lanes, where the accuracy of the

vehicle's directional alignment affects both the smoothness of transitions and lane-keeping stability. An autonomous controller which exhibits low heading error can demonstrate quick correction capabilities, ensuring the vehicle remains properly oriented and reducing the likelihood of abrupt steering adjustments.

Simulink Function's Script for Cross Track and Heading Error

```matlab
function [e_ct,e_h]=errors(ref_pose,ze)

pose_f=ze(1:3);

% Wrap angles in the interval [0,2*pi]
ref_pose(3)=wrap_to_2pi(ref_pose(3));
pose_f(3)=wrap_to_2pi(pose_f(3));

% Cross-track error
d=ref_pose(1:2)-pose_f(1:2);
e_ct=d(2)*cos(ref_pose(3))-d(1)*sin(ref_pose(3));

% Heading error
e_h=wrap_to_pi(ref_pose(3)-pose_f(3));

end

%
   --------------------------------------------------------------------------


function lambda = wrap_to_2pi(lambda)
%   Wraps angle in radians to [0 2*pi]
%   lambdaWrapped = wrapTo2Pi(LAMBDA) wraps angles in LAMBDA, in
    radians,
%   to the interval [0 2*pi] such that zero maps to zero and 2*pi
    maps
%   to 2*pi. (In general, positive multiples of 2*pi map to 2*pi
    and
%   negative multiples of 2*pi map to zero.)
%   See also wrapToPi, wrapTo180, wrapTo360.
positiveInput = (lambda > 0);
lambda = mod(lambda, 2*pi);
lambda((lambda == 0) & positiveInput) = 2*pi;
end

function lambda = wrap_to_pi(lambda)
%   Wraps angle in radians to [-pi pi]
%   lambdaWrapped = wrapToPi(LAMBDA) wraps angles in LAMBDA, in
    radians,
%   to the interval [-pi pi] such that pi maps to pi and -pi maps
    to
%   -pi.  (In general, odd, positive multiples of pi map to pi and
    odd,
%   negative multiples of pi map to -pi.)
%   See also wrapTo2Pi, wrapTo180, wrapTo360.
```

```
39  q = (lambda < -pi) | (pi < lambda);
40  lambda(q) = wrap_to_2pi(lambda(q) + pi) - pi;
41  end
```

Listing 4.3: Carla Environment NMPC Class

Chapter 5

Simulation Results and Analysis

5.1 Path Tracking Results

To verify the effectiveness of our proposed method, simulation results were obtained using the CARLA simulator. As the model verification has been implemented earlier as the first step in the urban area, the subsequent validation step involves testing it under highway conditions. This stage ensures that the system can effectively manage higher velocities and prolonged lane-keeping maneuvers, as well as respond to demanding conditions identifying vehicles and other objects. After reviewing the available highway routes in CARLA, we have selected MAP 04, as shown in Figure 5.1, which features a highway environment, allowing a clear observation of lateral deviation and heading consistency across different paths. This selection ensures a suitable test environment for evaluating our system's performance.

Figure 5.1: Map Overview

In this simulation, the autonomous vehicle (AV) is initially positioned in the rightmost lane at a standstill (Figure 5.2), illustrated in Figure 5.3, with both velocity and acceleration set to zero. Its orientation is aligned with the highway's longitudinal axis, establishing a standardized starting point for evaluating the subsequent acceleration, trajectory tracking and its overall performance. The reference trajectory here was generated by capturing center line of the lane throught the whole path, which provides a steady, predefined path for the vehicle to follow. The primary objectives here are to validate the controller's ability to track these reference trajectories under nonlinear vehicle dynamics and constraints, emulating realistic driving scenarios. In other words, observing the vehicle maintaining its stability while navigating highway curves and straight paths together with performing smooth maneuvers.

Figure 5.2: Ego Vehicle View

Metrics serve a vital role in determining how effectively a control system adheres to its intended trajectory. These quantitative indicators not only enable objective comparisons among various control algorithms but also facilitate systematic enhancements aimed at improving overall safety, stability, and passenger comfort. Here the performance is assessed through two main key metrics, namely cross-track error (CTE) and heading error; CTE measures the vehicle's lateral deviation from the reference path, ensuring precision in lane-keeping, while heading error quantifies angular mismatches in orientation for directional accuracy. These metrics offer a comprehensive view of both lateral and angular control efficacy, complementing a broader suite of data-driven evaluations. These two metrics hold particular relevance on highways, where higher speeds and frequent lane transitions require precise

Figure 5.3: Path Reference

lateral control to mitigate drift and ensure robust navigation.

Before initiating a turn or overtaking maneuver, the controller, by assessing the conditions, applies a measured slowdown or acceleration if required, while executing a fine-tunes steering, torque, and braking to minimize path deviations and ensure stable lateral positioning. It is important that the vehicle transitions smoothly changes, demonstrating a seamless progression from controlled deceleration or acceleration to steady travel. For overtaking however the vehicle gradually accelerates and shifts lanes while maintaining safe lateral clearance, continuously adjusting torque, braking, and steering to deliver a stable lane-change experience. Once the overtake is completed, the vehicle realigns with its chosen lane and resumes its steady cruising speed. More details are explained in the following section.

5.2 Scenario Definition and Results

The main goal of the following scenarios is to observe the vehicle maintaining its stability when navigating highway curves and straight paths in addition to performing smooth maneuvers. Thereby we define two scenarios as the following:

5.2.1 Scenario 1: Highway Lane Keeping

In this scenario, the vehicle travels along a standard highway route without any overtaking maneuver, maintaining a consistent speed and following the corresponding lane. The primary objective is to evaluate the controller's ability to manage highway conditions, including slight curves and transitions between straight segments, without deviating from the defined path. As shown in 5.4, in this case before initiating a turn, the controller applies a measured slowdown or if required, the vehicle travels along a standard highway route, maintaining a consistent speed and following the designated lane. The primary objective is to evaluate the controller's ability to manage routine highway driving conditions, including slight curves and

transitions between straight segments, without deviating from the defined path, and continuously adjusting torque, braking, and steering to deliver a stable lane-change experience. More details are explained in the following section. The figure 5.4 provides a visualization of the path layout:

Figure 5.4: First Scenario Path

5.2.2   Scenario 2: Highway Overtaking

This scenario focuses on assessing the vehicle's capacity to perform overtaking maneuvers smoothly, therefore the focus shifts to evaluating the controller's ability to execute an overtaking maneuver within typical highway traffic conditions. The controller identifies an appropriate gap in the adjacent lane, transitions into it while adjusting speed and lateral positioning, and then returns to the original lane once the maneuver is complete. This setup tests both the adaptability and responsiveness of the autonomous system under typical highway passing conditions. In simpler terms, the controller, by assessesing the conditions, applies a measured acceleration while executing a fine-tune steering, torque, and braking to minimize path deviations from the trajectory and ensure stable lateral positioning. It is important that the vehicle transitions smoothly, demonstrating a seamless progression from controlled acceleration to steady travel. For more clarity, the figure 5.5 presented below shows more detail of the path layout:

Figure 5.5: Second Scenario Path

5.2.3 Highway Lane Keeping NMPC Results

Having described the scenarios and corresponding routes in detail, we now turn to the assessment of the NMPC's performance within the simulated environment. Aligning our reference drive within a shared coordinate system enables a direct comparison between the ideal driving path and the subsequent NMPC-controlled path.

As shown in Figure 5.6, the NMPC-controlled vehicle's trajectory aligns closely with the reference path, indicating that the controller can effectively predict and adjust for the required steering and speed inputs.

We analyze the route further in two distinct segments at a finer scale: the straight path and the curved path; as shown in Figure 5.7 and Figure 5.8, minor deviations from the reference path become apparent. Regarding the straight segment, the model demonstrates stable lane-keeping and appropriate steering adjustments throughout the segment, having small mismatches which reflect the inner complexities of real-time control in an autonomous driving context, mainly the predictive horizon and vehicle dynamics influence the controller's capacity to align perfectly with the reference trajectory. However, regarding the curved path, at the beginning, it shows some trajectory errors due to the sudden change in lateral acceleration and the vehicle inertia and thus the initial response lag of the control system. Specifically, the controller rapidly adjusts steering angles and speed to accommodate the increased curvature, and these rapid adjustments inherently introduce transient deviations from the ideal trajectory.
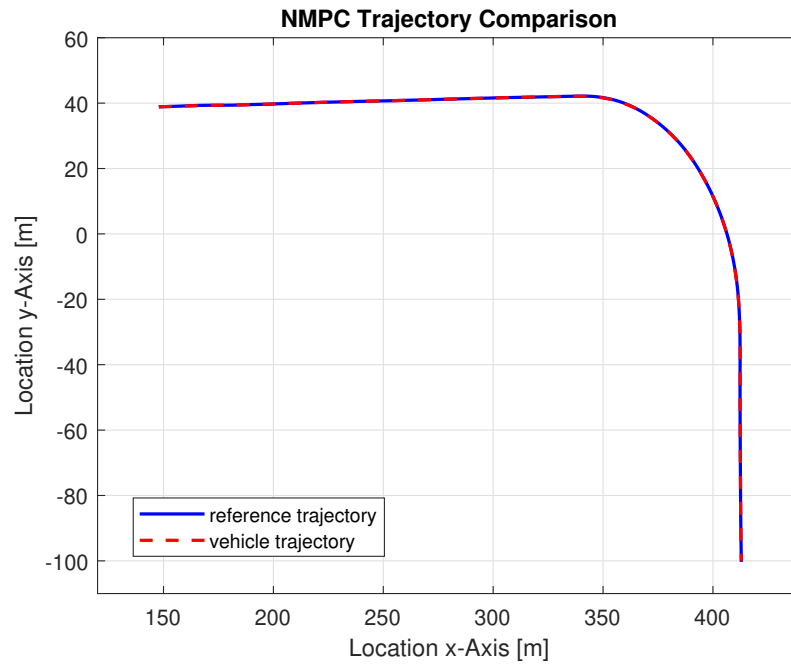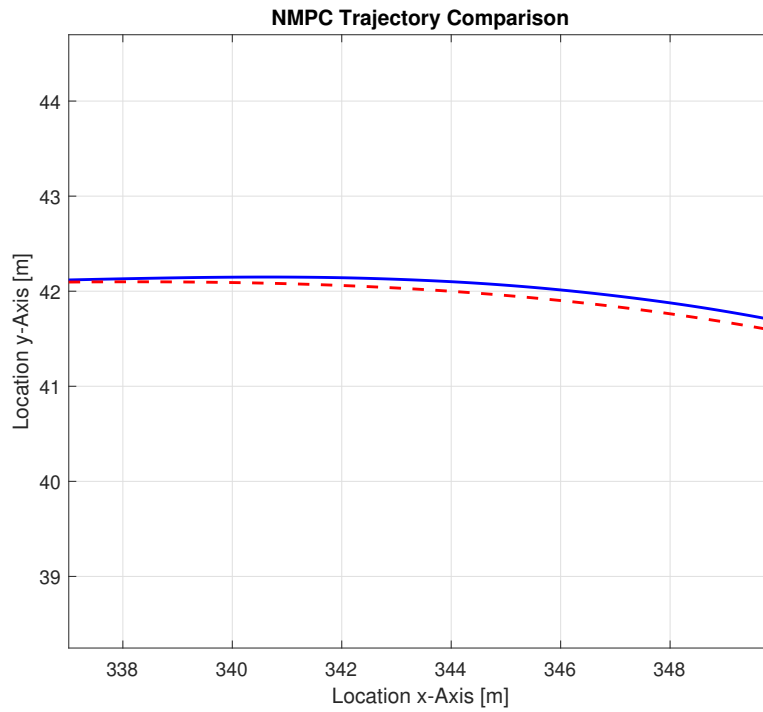
Figure 5.6: Trajectory Comparison Scenario 1

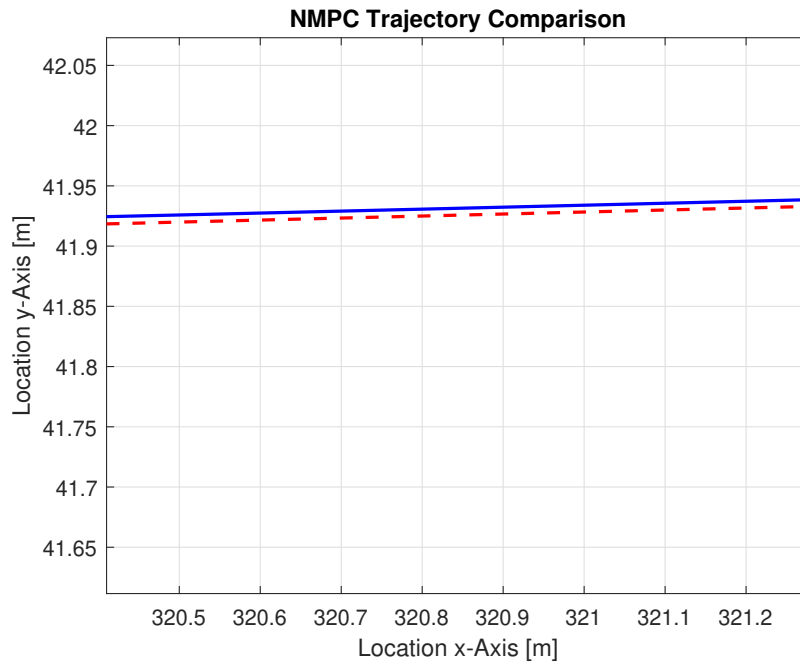Figure 5.7: Trajectory Comparison Scenario 1 Curved Path Detailed View

Figure 5.8: Trajectory Comparison Scenario 1 Straight Path Detailed View

To achieve a quantitative assessment of the model's performance, the metrics previously explained, namely cross-track error and heading error are examined, and the results are illustrated in the Figure 5.9 and the Figure 5.10, respectively.
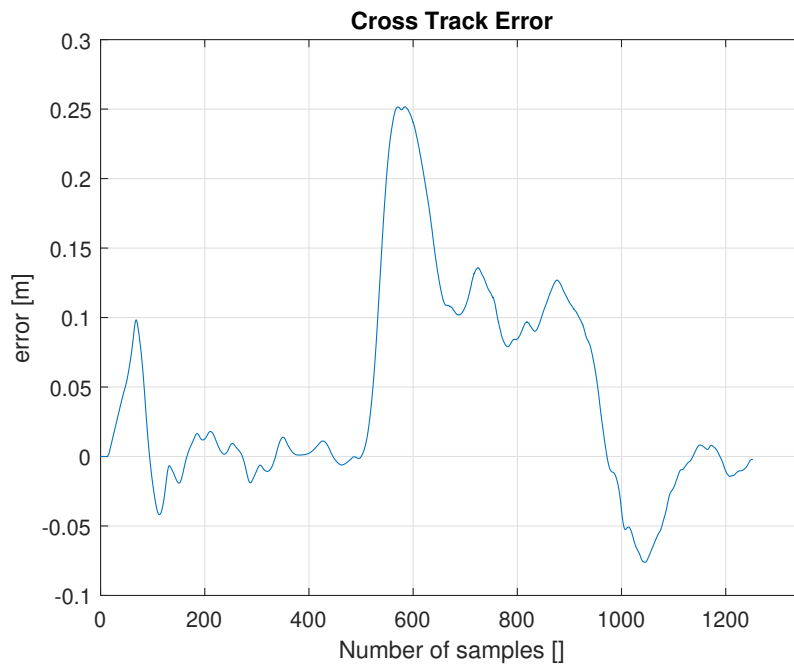

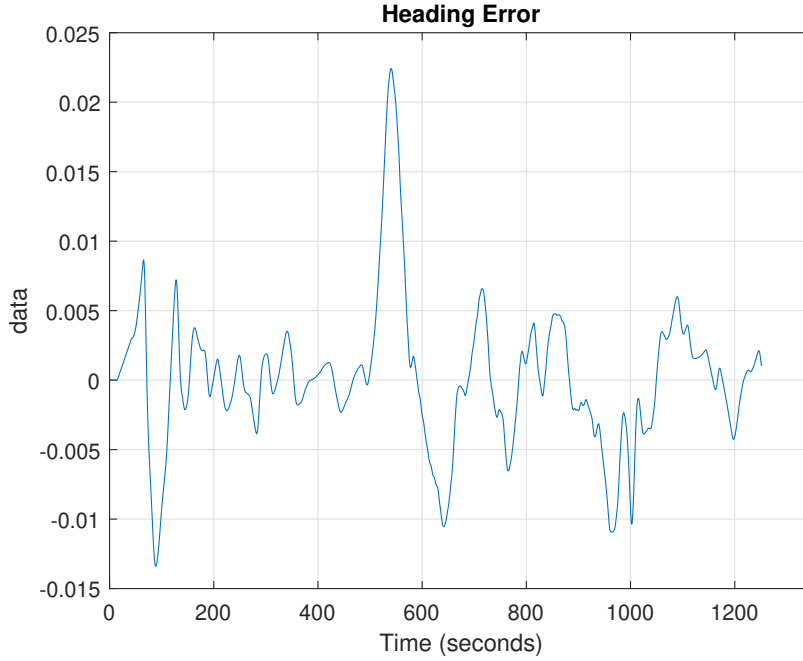Figure 5.9: Cross Track Error Scenario 1

Figure 5.10: Heading Error Scenario 1

As it was obvious through the trajectory figures above, at the beginning of the maneuver, both cross-track and heading errors are higher, indicating an initial transient state, typically caused by the vehicle's adjustment to the changing trajectory, inertia effects, and the controller's initial predictive response lag. As the vehicle progresses along the curve, the predictive part of NMPC allows it to anticipate and correct these initial errors, indicating improved lane adherence as the maneuver progresses. Over the next steps, feedback control incrementally reduces these inaccuracies by refining steering and throttle-brake inputs, realigning its orientation with the reference direction. Also two statistical indicators, namely Root Mean Square (RMS) and Maximum (Max) error values are here employed, providing better accuracy rate of the model's performance:

Root Mean Square reflects how closely and smoothly the vehicle complies to the reference trajectory throughout the scenario. A lower RMS value indicates better tracking performance as well as smoother maneuvers and greater stability

Maximum Error represents the largest deviation observed during the test. This metric highlights potential worst-case scenarios, revealing how the controller handles the most challenging conditions, such as sharp curves or rapid transitions.

	RMS Value [m]	MAX Value Error [m]
Cross Track Error	0.087	0.252
Heading Error	0.006	0.023

Table 5.1: NMPC Performance Scenario 1

5.2.4   Highway Overtaking NMPC Results

In this scenario, the vehicle follows the same highway route as in scenario 1, but the simulation now incorporates three overtaking maneuvers along with sharper turns, with the aim of challenging the model by introducing rapid changes in steering angle and acceleration variations, thereby testing its response under more demanding driving conditions. At first look in Figure 5.12, it is apparent that even with the added complexity of the route, the vehicle's behaviour is quite reliable, supporting the notion that the NMPC model here is capable of balancing aggressive and abrupt maneuvers with precise tracking path, ensuring the reliability of the model.

In the following, as the first scenario, a detailed graphical overlay of the ac-
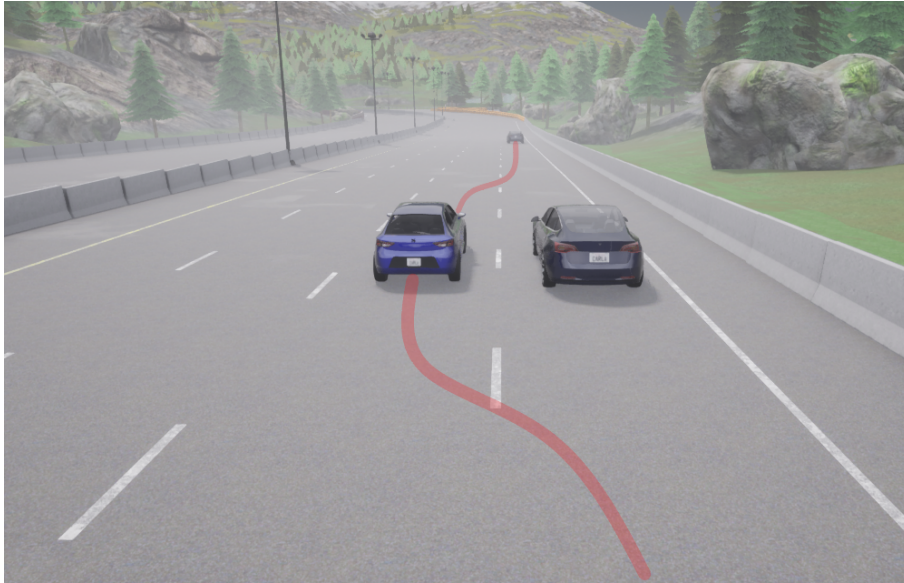
Figure 5.11: Ego Vehicle View

tual and reference trajectory. Looking at the figure 5.13, the inclusion of sharper turns regarding overtaking maneuvers introduces additional challenges, such as rapid changes in steering angle. However the close alignment between the two trajectories, as suggested by the figure, implies that the control system effectively manages rapid changes. Here we also quantify the performance of the NMPC for the chosen path using numerical error metrics, including cross-track error, heading error, and
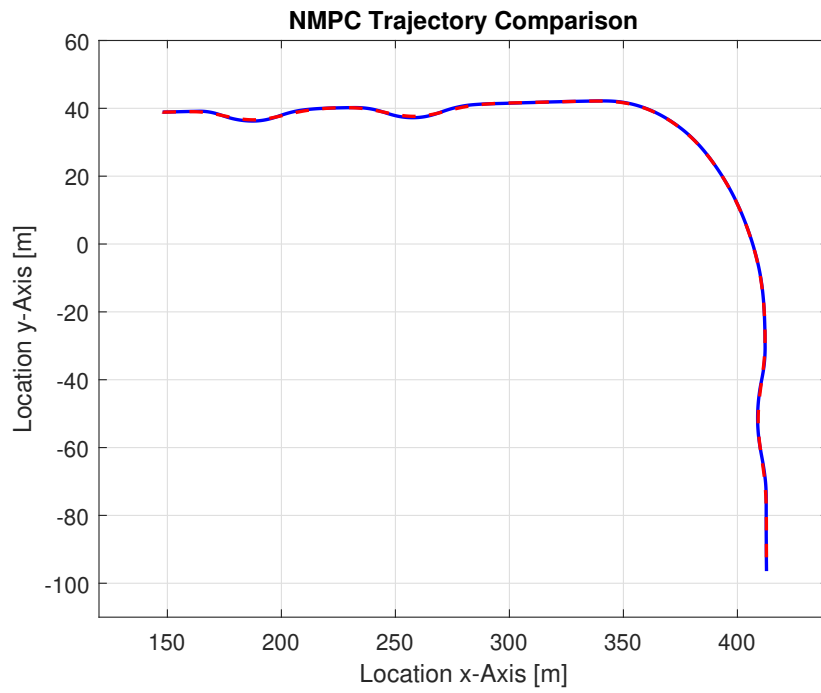
Figure 5.12: Trajectory Comparison Scenario 2
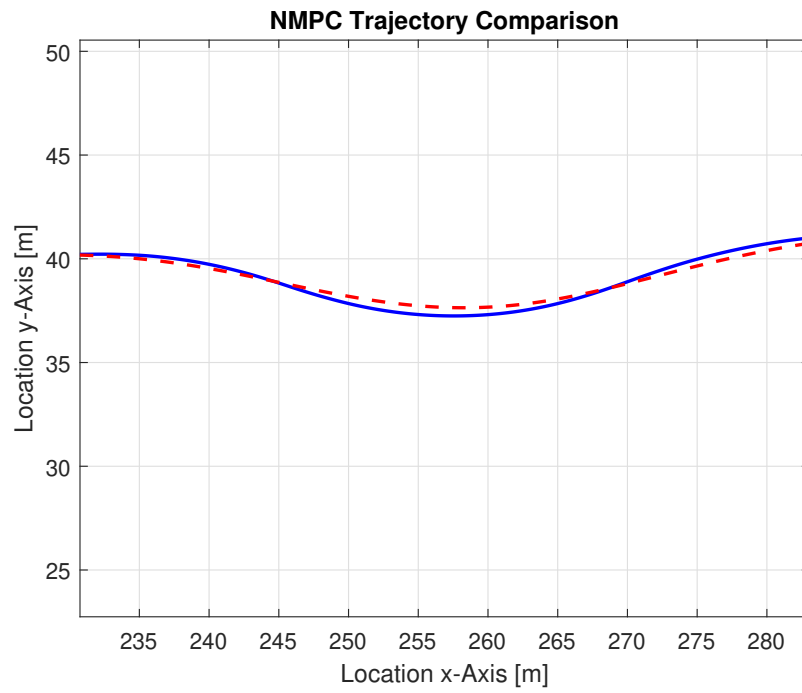
Figure 5.13: Trajectory Comparison Scenario 2 Curved Path Detailed View

their associated RMS and maximum values. The initial rises while performing the overtaking can be explained by the vehicle's sudden lateral adjustment to follow the curved path, the inertia effects inherent in its dynamics, and the brief delay in the NMPC's predictive response, which also delays a little the immediate realignment with the reference trajectory. This delay creates a temporary deviation, which is apparent as an increased cross-track error. For the same reason, transient peaks in heading error may occur during abrupt maneuvers, such as during sharp turns and lane changes, where the vehicle quickly adjusts its direction itself.

Figure 5.14: Cross Track Error Scenario 2

Figure 5.15: Heading Error Scenario 2

Although transient deviations here are evident during sharp turns, these variations are not significant when evaluated against the quantitative measure of fluctuations sho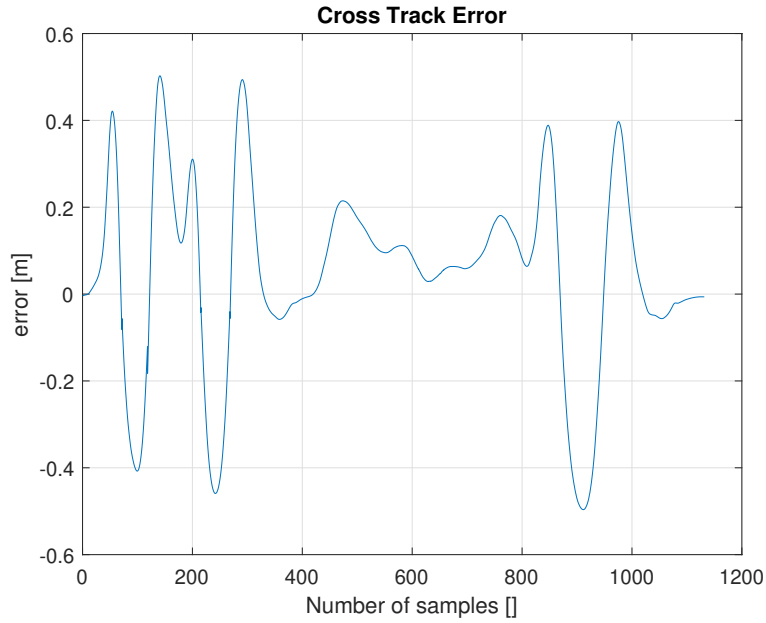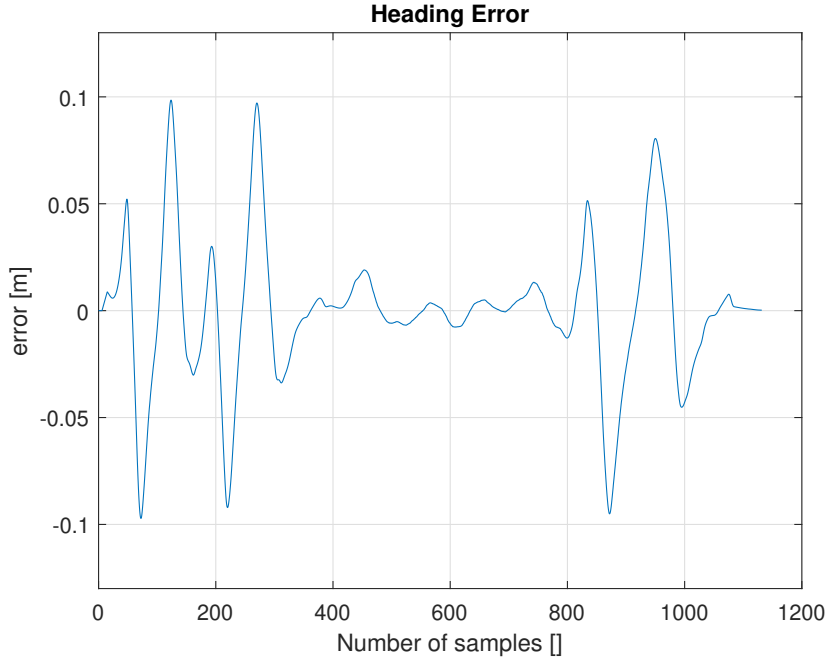wn in Table 5.6, namely the RMS and maximum error values remain sufficiently low, indicating that the model maintains an acceptable level of tracking accuracy.

	RMS Value [m]	MAX Value Error [m]
Cross Track Error	0.243	0.512
Heading Error	0.048	0.098

Table 5.2: NMPC Performance Scenario 2

5.2.5   Manual Driving mode

An alternative perspective is to perform the scenarios above in a manual driving mode, demonstrating it as a way to capture real-world human behavior within the virtual environment. By doing so, having a human driver to control the vehicle's steering and pedals, we gain insight into how a driver naturally adjusts speed and maintains lateral positioning. This real-time data provides a baseline of typical human performance. With this reference, we can later compare the automated controller's capabilities, determining how closely it mirrors or even surpasses a driver's intuitive handling of the route. Analyzing the CTE data generated during these

manual driving sessions provides a realistic baseline for lateral deviations experienced under human-controlled conditions. This empirical reference allows us to directly compare the performance of our NMPC model with the responsiveness of human drivers in terms of precision and stability, as well as quantifying the extent to which human reaction times and control inconsistencies contribute to lateral deviations. In the following the results of the simulations for the first and second scenario are shown in 1 and 2, respectively.

Figure 5.16: Manual Driving Mode Cross Track Error Scenario 1
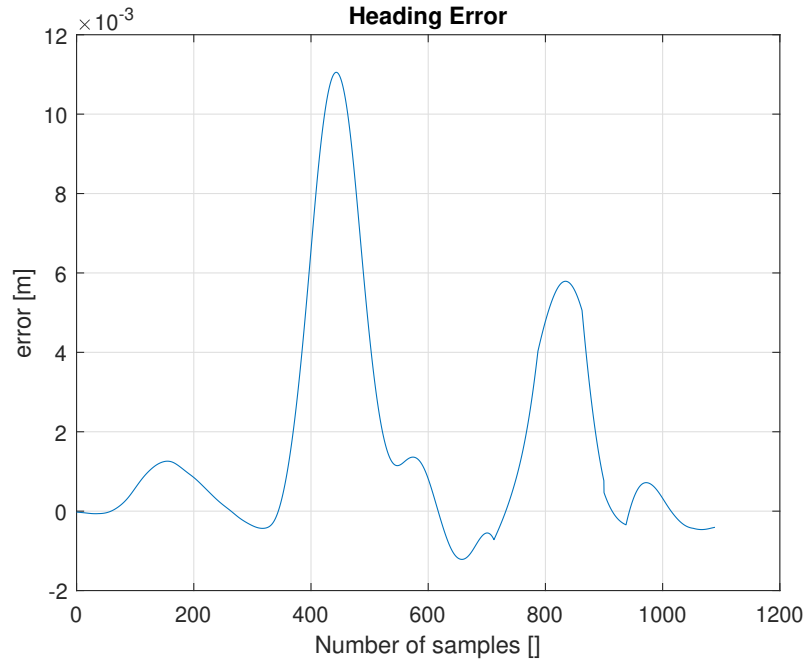
Figure 5.17: Manual Driving Heading Error Scenario 1

Here is the analysis of errors for the first scenario:

	RMS Value [m]	MAX Value Error [m]
Cross Track Error	0.143	0.264
Heading Error	4.32e-3	11.1e-3

Table 5.3: Manual Driving Performance Scenario 1
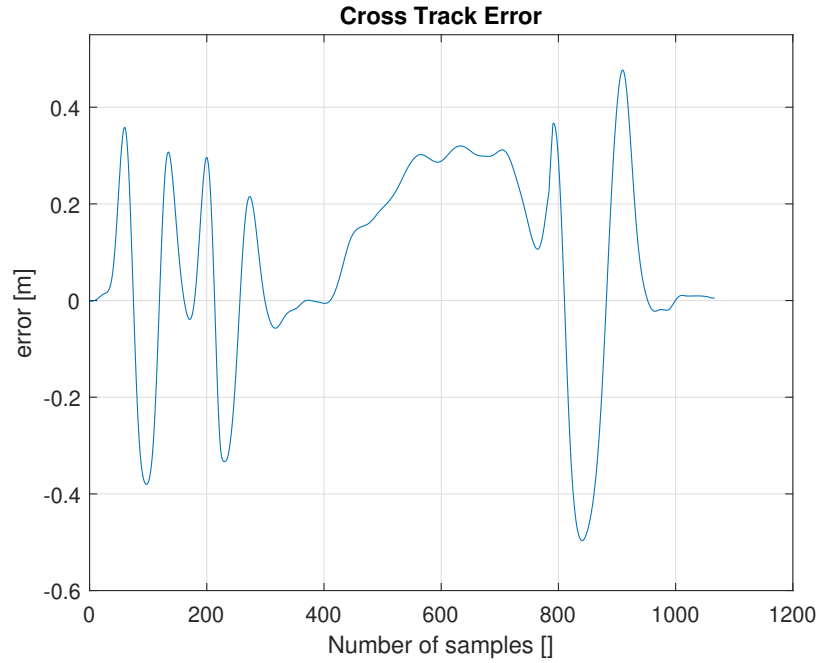
Figure 5.18: Manual Driving Mode Cross Track Error Scenario 2
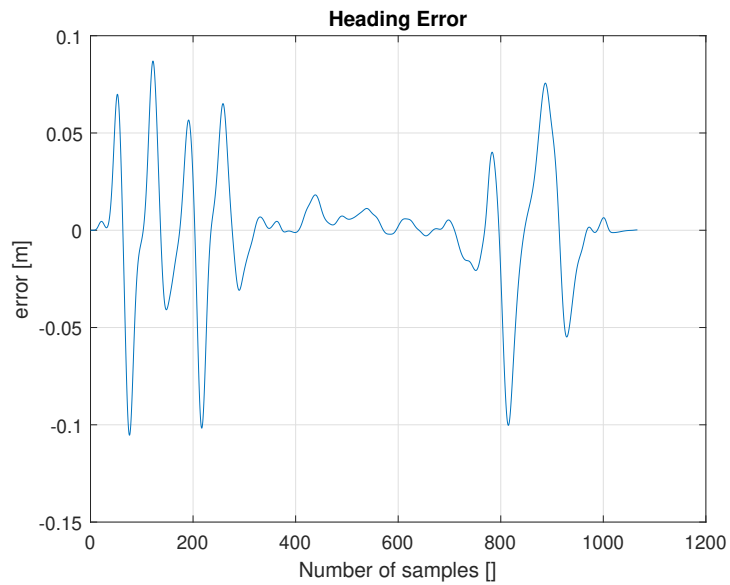
Figure 5.19: Manual Driving Mode Heading Error Scenario 2

For the second scenario the numeric values of the errors are as the following:

	RMS Value [m]	MAX Value Error [m]
Cross Track Error	0.384	0.612
Heading Error	0.048	0.098

Table 5.4: Manual Driving Performance Scenario 2

5.2.6 Carla Autonomous Driving

The CARLA's autopilot mode as explained previously in chapter 3, is here employed as an established benchmark for subsequent comparative analyses as this mode also offers predictable behavior under diverse driving conditions in addition to the reason that autopilot mode is well-suited to the highway setting, where continuous pathways and defined lane structures allow the system to demonstrate its precision in trajectory trackinga. This PID controller continuously receives feedback on the vehicle's position relative to the desired trajectory and calculates the necessary steering commands by combining the effects of the proportional, integral, and derivative terms. In the following, we present the evaluation of the autopilot mode's performance.

Figure 5.20: Autopilot Mode Cross Track Error Scenario 1
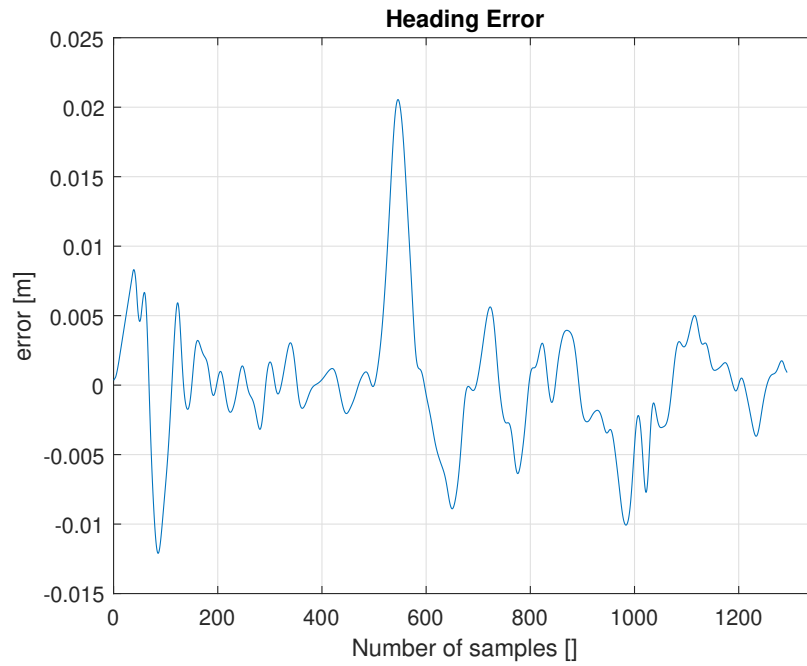
Figure 5.21: Autopilot Mode Heading Error Scenario 1

	RMS Value [m]	MAX Value Error [m]
Cross Track Error	0.165	0.322
Heading Error	0.072	0.020

Table 5.5: Autopilot Mode Performance Scenario 1

and for the second scenario the results are:
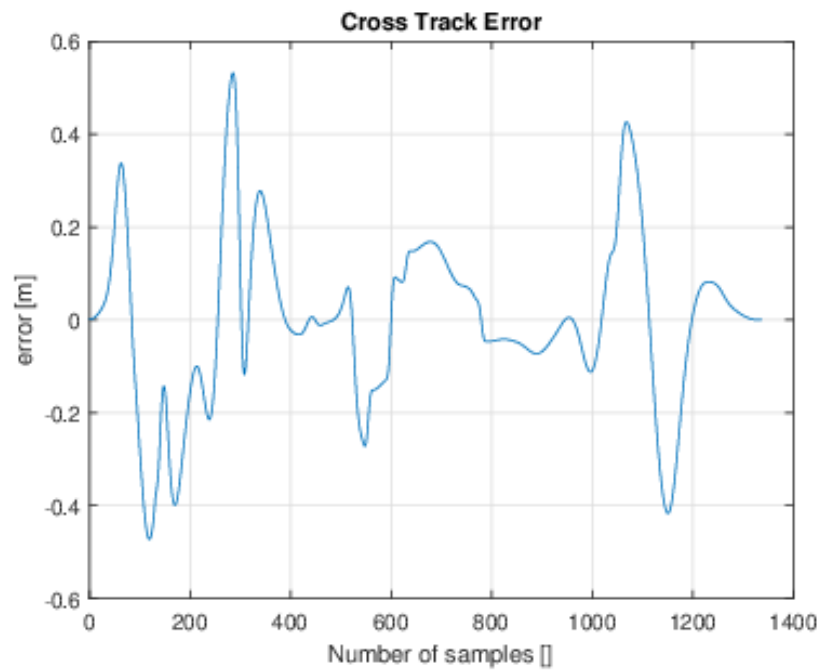
Figure 5.22: Autopilot Mode Cross Track Error Scenario 2
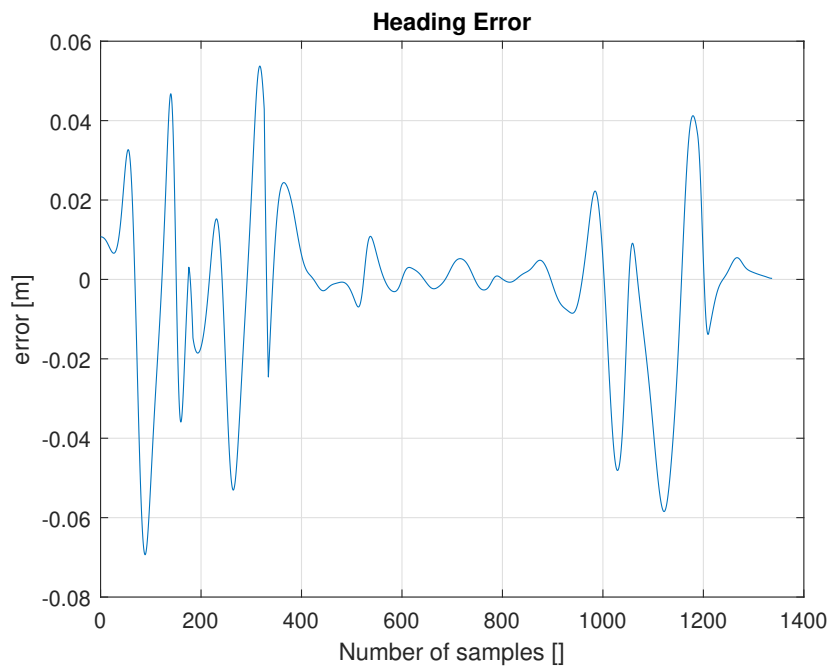
Figure 5.23: Autopilot Mode Heading Error Scenario 2

The errors remain relatively quite low throughout the straight path, indicating that the vehicle consistently adheres to the planned path. When the vehicle approaches the curved segment, a small spike in error occurs due to the abrupt change in path geometry. However, this time, it has been slightly higher than the previous control method (NMPC). This spike is a natural response to the dynamic transition. As the vehicle continues through the curve, the error gradually diminishes and returns to low levels for the remainder of the route, demonstrating the autopilot's capability to recover and re-establish control over the vehicle's lateral position. As detailed in the previous section, the RMS and maximum errors have been computed for both cross-track and heading metrics to quantitatively assess the autopilot's trajectory tracking performance. The table below presents these numeric results:

|	RMS Value [m]	MAX Value Error [m]
Cross Track Error	0.257	0.572
Heading Error	0.033	0.066

Table 5.6: Autopilot Mode Performance Scenario 2

In the takeover scenario, the control system experiences a temporary but noticeable degradation in tracking performance, primarily observable through spikes in both cross-track error (CTE) and heading error; This behavior is expected in dynamic driving situations where rapid adaptation is required, as here there are 3 maneuvers added to the scenario. Hence the increase in the RMS values and Heading errors reflects a general rise in the overall deviation from the reference trajectory during the maneuver phase before the autopilot re-stabilizes the vehicle. Although the RMS and maximum errors remain elevated compared to straight-line or nominal driving, they remain within acceptable bounds for highway-level autonomous operation, indicating the PID-based autopilot mode demonstrates a reliable performance, effectively managing the overtaking tasks while maintaining its stability.

5.2.7   Performance Comparison

From an academic standpoint, comparing NMPC with manual driving and AUtopilot Mode in Carla provides critical insights into the advantages of model-based predictive control in managing the inherent complexities of vehicle dynamics. The NMPC controller's ability to predict future states and optimize control inputs contributes to lower RMS and maximum error values, even in scenarios demanding rapid adjustments such as overtaking. This not only validates the controller's design but also underscores its potential to reduce human error in high-risk maneuvers. In contrast, manual driving performance, while flexible, is limited by human reaction times and variability, which often result in higher transient errors during sharp maneuvers. By providing a benchmark based on quantitative analysis, this comparative study contributes to strengthen the idea that automated strategies can deliver superior consistency and reliability.

Table 5.7: Comparative Analysis of Various Control Strategies Scenario 1

Control Method	RMS e_{ct}	RMS e_h	Max e_{ct}	Max e_h
NMPC Control	0.087	0.048	0.252	0.098
Autopilot Control	0.165	0.072	0.322	0.020
Manual Driving	0.143	4.32e-3	0.264	11.1e-3

Table 5.8: Comparative Analysis of Various Control Strategies Scenario 2

Control Method	RMS e_{ct}	RMS e_h	Max e_{ct}	Max e_h
NMPC Control	0.243	0.0174	0.512	0.0450
Autopilot Control	0.257	0.033	0.572	0.066
Manual Driving	0.384	0.048	0.612	0.098

However, the predictive nature of NMPC requires solving an optimization problem in real time, which can be computationally intensive in more unpredictable conditions. Regarding the advanced PID controller used in CARLA's autopilot as seen in the table of results, it has a good balance of performance with simpler implementation and lower computational demands compared to NMPC, demonstrating reliable performance on normal highway trajectories, and also maintaining low deviations during straight sections and gradual curves as it was in our scenario case. However sometimes this controller may experience brief unfavourable spikes in its behaviour during rapid or unexpected changes in the trajectory, leading to higher maximum errors in certain scenarios. Unlike NMPC, the PID approach might not predict future trajectory changes as effectively, potentially limiting performance in highly dynamic environments.

5.3 Obstacle Avoidance

Object avoidance in autonomous highway driving involves integrating safety constraints into the path-following algorithm to ensure that the vehicle maintains a safe distance from obstacles while following its intended route. Essentially, the controller not only tracks a predefined reference trajectory but also continuously monitors for obstacles—such as other vehicles, debris, or road hazards—and dynamically adjusts the planned path to avoid collisions. This is achieved by formulating additional constraints within the control problem that define safe zones around the detected objects.

The constraints typically specify a minimum separation distance between the vehicle and any obstacle, ensuring that the vehicle does not come too close. These can be spatial constraints, such as maintaining a safe lateral distance, and temporal constraints, like ensuring the vehicle has enough time to slow down or steer away when an obstacle is detected. By putting these constraints together into a Nonlinear Model Predictive Control framework, the controller can optimize the vehicle's trajectory in real time, balancing the need to follow the reference path with the imperative to avoid obstacles. In practice, this approach allows the vehicle to integrate object avoidance into its overall navigation strategy. When an object is detected, the controller adjusts the trajectory within the bounds of constraints, ensuring that deviations from the reference path are minimized while still providing a buffer to prevent collisions.
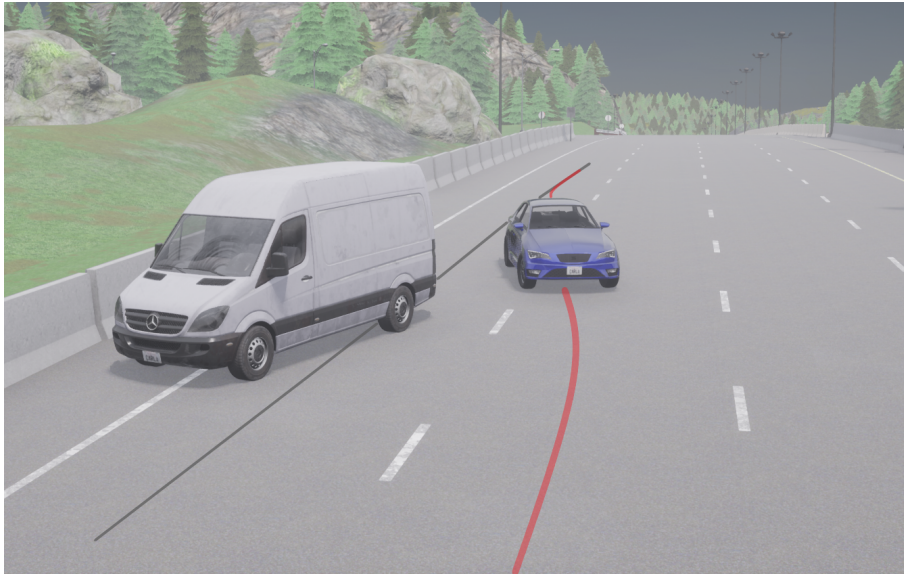
Figure 5.24: Path changing of the Vehicle with Object

Therefore, following the successful behaviour of the model for path-following scenarios, the next phase of evaluation of our model would involves an object avoidance; In this case, the model is required to autonomously detect an object, execute a lane change to safely bypass it, and then return to the original lane once the object has been passed. The objective is to perform these maneuvers smoothly, without introducing abrupt control responses that could compromise safety or passenger comfort. The methodology used here is a state-driven system that adapts its behavior based on eventual occurrences. In other words, under normal driving conditions, the controller maintains a baseline state focused solely on tracking the predefined reference. However, when the vehicle's sensors detect an obstacle—based on the localization of the object and the trajectory of the ego vehicle-it transitions into an obstacle avoidance state.

In this object avoidance state, the controller dynamically recalculates the reference trajectory to create a safe bypass; once the obstacle reaches a threshold condition defined in advance, the system then adjusts the trajectory so that the vehicle can safely navigate around the obstacle, taking into account the vehicle's dimensions and the obstacle's position relative to the original path. Once the vehicle successfully bypasses the obstacle and re-enters its original lane with a proper safety margin regarding the obstacle, the system automatically transforms back to its normal state. The results of this scenario are shown in the following:
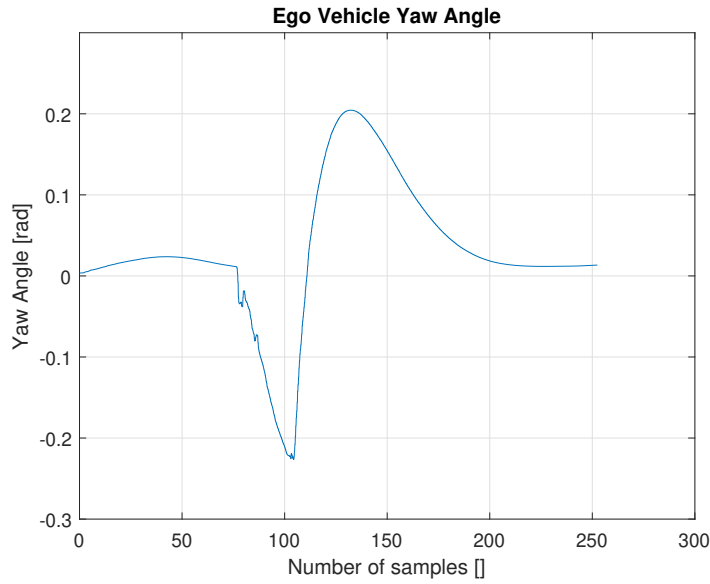
Figure 5.25: Yaw Angle of the Ego Vehicle

Regarding the results above we can say that a smoothly varying yaw angle typically indicates stable steering control, whereas sharp or abrupt changes may point to sudden evasive maneuvers—often necessary in object avoidance scenarios. The close alignment of actual and reference trajectories—except where deviation is required to circumvent obstacles—implies that the system can be both safe and efficient in real-world scenarios, where the model demonstrates a reliable approach to obstacle avoidance, balancing the need for accurate path following with timely, smooth evasive actions.
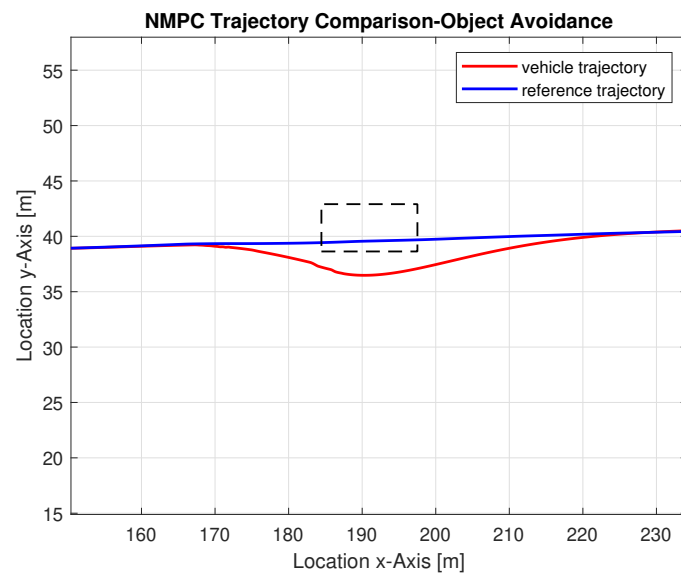
Figure 5.26: Trajectory of the Ego Vehicle

Chapter 6

Conclusion

6.1 Summary of Findings

To evaluate the controller's performance, a simulation-based experiment is carried out. This thesis has demonstrated the effectiveness of Nonlinear Model Predictive Control in enhancing path tracking, vehicle stability, and control precision in autonomous driving applications. The research focused on integrating NMPC into a MATLAB/Simulink and CARLA co-simulation framework, where a comprehensive system identification process and dispatching functions were implemented to transmit real-time control commands for vehicle operation in the CARLA simulator. Simulations were carried out in two distinct scenarios, featuring different maneuvers and speeds. Key evaluation metrics, Cross-Track Error (CTE) and Heading Error (HE), were analyzed to quantify tracking accuracy.

Additionally, manual driving data was incorporated to provide a realistic human driving benchmark, allowing for a direct comparison between NMPC, CARLA autopilot, and human driving behavior. Carla's built-in PID-based Autopilot is discussed as a benchmark for comparison, illustrating the typical strengths and limitations of a system that is presumably fine-tuned for a specific vehicle model. This sets the stage for exploring the NMPC approach, which employs a predictive model to anticipate future vehicle states. Even though it uses a simplified, more general vehicle model, NMPC can optimize control inputs in real time to achieve accurate and stable performance across varying speeds and driving conditions. The final results of this study reveal that the NMPC-based system can outperform or match the simulator's native solution, despite the latter likely being tailored to a single vehicle model. The findings underscore the generalizability and reliability of NMPC, emphasizing its potential as a robust solution for future autonomous driving applications.

6.2   Future Work

Future work could focus on refining the vehicle model accuracy, incorporating more precise road condition data, and extending NMPC's capabilities to additional ADAS functionalities such as lane changes and adaptive speed control, further advancing the field of automated driving systems. We anticipate that CARLA will serve as a valuable platform for researchers by offering a setting for testing a wide range of perception, planning, and control strategies under realistic driving conditions. It provides the capability to generate large-scale, diverse datasets, which are crucial for the development of learning-based autonomous driving algorithms, including deep learning models for perception, behavior prediction, and decision-making.

While the Nonlinear Model Predictive Control (NMPC) framework developed in this project demonstrates promising results for path tracking within the CARLA simulator, there remain several exciting avenues to enhance its robustness and applicability to real-world autonomous driving scenarios. A first extension would involve integrating richer perception capabilities. In its current form, the NMPC primarily relies on high-level trajectory information. However, in real driving environments, diverse sensor data—such as cameras, LiDAR, radar, or GPS/IMU—would be necessary to detect obstacles, lane boundaries, and other key features. By fusing multiple sensor inputs, one can refine the NMPC cost function and constraints to account for uncertain or dynamic elements (e.g., moving vehicles, pedestrians). This sensor-fusion approach would likely involve advanced filtering techniques to robustly estimate the vehicle's state and its surroundings in real time.

A second related extension would be the development of adaptive or robust NMPC schemes. In practical deployments, vehicle dynamics may vary due to tire wear, changing weather, or aerodynamic effects. Consequently, incorporating parameter adaptation mechanisms—such as online identification via Extended Kalman Filters or machine learning estimators—can help the NMPC remain effective under a broader range of conditions.

Moreover, there is significant potential in combining classical NMPC with learning-based methods. Reinforcement learning or imitation learning could generate high-level policies or reference trajectories, which the NMPC controller would then refine, ensuring adherence to stability and safety constraints. Such a hybrid approach balances the adaptability of data-driven models with the reliability of model-based control.

Lastly, moving beyond single-vehicle scenarios to multi-agent or cooperative driving holds considerable promise. Extending the NMPC framework to account for interactions with other road users, traffic rules, and communication protocols would mark a critical step toward fully autonomous driving solutions. Ultimately, validating these enhancements in more realistic conditions—via hardware-in-the-loop testing or scaled vehicle prototypes—would demonstrate their effectiveness and pave the way for real-world deployment of NMPC-based autonomous driving systems.

Appendix A

Appendix

As discussed in Chapter 3, which outlines the data collection methodology in CARLA using Python, this section presents a more complete script that was implemented for data gathering:

```
steering_data = []

location_data_x = []
location_data_y = []
location_data_z = []
velocity_data_x = []
velocity_data_y = []
angular_velocity = []
yaw_data = []
acceleration_data_x = []
acceleration_data_y = []
throttle_data = []
brake_data = []
steering = []

start_time = time.time()
duration = 300 #based on methodology being in use
class World(object):
    def __init__(self, carla_world, hud, actor_filter):
        self.world = carla_world
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = actor_filter
        self.restart()
        self.world.on_tick(hud.on_world_tick)

    def get_state(self,):

```

```
35
36          acceleration = self.player.get_acceleration()
37          steer_value = self.player.get_control().steer
38          throttle_value = self.player.get_control().throttle
39          brake_value = self.player.get_control().brake
40          location = self.player.get_location()
41          velocity = self.player.get_velocity()
42          transform = self.player.get_transform()
43          yaw = transform.rotation.yaw
44          yaw_rate = self.player.get_angular_velocity().z
45          yaw_rate = yaw_rate * math.pi/180
46          yaw = yaw * math.pi/180
47          # print(acceleration)
48          # print(location)
49          # print(f'yaw:{yaw}')
50          # print(f'acceleration_x:{acceleration.x}')
51          # print(f'acceleration_y:{acceleration.y}')
52          print(f'velocity_x:{velocity.x}')
53          print(f'position_x:{location.x}')
54          print(f'position_y:{location.y}')
55          print(f'position_z:{location.z}')
56          #print(f'velocity_y:{velocity.y}')
57          #print(yaw)
58          #-1.9616636037826538
59          #report the data into  a list, called state
60          global acceleration_x
61          global acceleration_data_y
62          global steering_data
63          global location_data_x
64          global location_data_y
65          global velocity_data_x
66          global velocity_data_y
67          global angular_velocity
68          global yaw_data
69          global location_data_z
70          global throttle_data
71          global brake_data
72          global steering_data
73
74          acceleration_data_x.append(acceleration.x)
75          acceleration_data_y.append(acceleration.y)
76          steering_data.append(steer_value)
77
78          location_data_x.append(location.x)
79          location_data_y.append(location.y)
80          location_data_z.append(location.z)
81          yaw_data.append(yaw)
82          velocity_data_x.append(velocity.x)
83          velocity_data_y.append(velocity.y)
84          angular_velocity.append(yaw_rate)
85          brake_data.append(brake_value)
86          throttle_data.append(throttle_value)
87          steering_data.append(steer_value)
```

```
88
89
90          return acceleration_data_x , steering_data , location_data_x ,
                location_data_y , yaw_data , velocity_data_x ,
                velocity_data_y , angular_velocity , throttle_data ,
                brake_data
91
92      def restart ( self ):
93          # Keep same camera config if the camera manager exists.
94          cam_index = self . camera_manager . index if self .
                camera_manager is not None else 0
95          cam_pos_index = self . camera_manager . transform_index if self
                . camera_manager is not None else 0
96          # Get a random blueprint.
97          #blueprint = random.choice ( self . world . get_blueprint_library
                (). filter ( self ._actor_filter ))
98          blueprint = self . world . get_blueprint_library (). filter ('leon
                ')[0]
99          blueprint . set_attribute ('role_name', 'hero')
100         if blueprint . has_attribute ('color'):
101             color = random . choice ( blueprint . get_attribute ('color').
                    recommended_values )
102             blueprint . set_attribute ('color', color )
103
104          spawn_points = self . world . get_map (). get_spawn_points ()
105
106             #23_10_24 primary and original location for the main
                    simulation
107             spawn_location = spawn_points [302]
108             #edited
109             spawn_location . location . x = 50.33776092529297
110             spawn_location . location . y = 37.75230407714844
111         self . player = self . world . try_spawn_actor ( blueprint ,
                spawn_location )
112
113 def game_loop ( args ):
114     pygame . init ()
115     pygame . font . init ()
116     world = None
117
118     try :
119         client = carla . Client ( args . host , args . port )
120         client . set_timeout (2.0)
121
122         display = pygame . display . set_mode (
123             ( args . width , args . height ),
124             pygame . HWSURFACE | pygame . DOUBLEBUF )
125
126         hud = HUD ( args . width , args . height )
127         world = World ( client . get_world () , hud , args . filter )
128         controller = DualControl ( world , args . autopilot )
129
130         # while True:
```

```
131          #       global start_time
132          #       global duration
133          #       current_time = time.time()
134
135          #       if current_time - start_time > duration:
136          #           break
137          #       world.get_state()
138
139          #       time.sleep(0.3)
140
141
142
143          clock = pygame.time.Clock()
144          while True:
145              clock.tick_busy_loop(60)
146              if controller.parse_events(world, clock):
147                  return
148              world.tick(clock)
149              world.render(display)
150              pygame.display.flip()
151
152              global start_time
153              global duration
154              current_time = time.time()
155
156              if current_time - start_time > duration:
157                  break
158              world.get_state()
159
160              # time.sleep(0.3)
161              time.sleep(0.05)
162
163
164      finally:
165          global location_data_x
166          global location_data_y
167          global acceleration_data_x
168          global acceleration_data_y
169          global steering_data
170          global velocity_data_x
171          global velocity_data_y
172          global yaw_rate
173          global yaw_data
174          global brake_data
175          global throttle_data
176          global angular_velocity
177
178
179          #simin ax, delta
180          #simout positionx,y,t,yaw, velocx,veloxy, yaw rate
181          # with open('sim_in.txt', 'w') as file:
182          #     file.write('acceleration_x, steering_data\n')
183          #     for i, j in zip(acceleration_data_x, steering_data):
```

```python
184            #            file.write(f'{i}, {j}\n')
185         with open('sim_in.txt', 'w') as file:
186             file.write('location_x, , location_y, yaw_data,
                   velocity_x, velocity_y, angular_velocity,
                   acceleration_x, acceleration_y, brake_value,
                   throttle_value\n')
187             for a, s, d, f, g, h, j, k, l, m  in zip(
                   location_data_x, location_data_y, yaw_data,
                   velocity_data_x, velocity_data_y, angular_velocity,
                   acceleration_data_x, acceleration_data_y, brake_data
                   , throttle_data):
188                 file.write(f'{a}, {s}, {d}, {f}, {g}, {h}, {j}, {k
                       }, {l}, {m}\n')
189         with open('sim_in_simulink.txt', 'w') as file:
190             file.write('acceleration_x, steering_value\n')
191             for z, x in zip( acceleration_data_x, steering_data):
192                 file.write(f'{z}, {x}\n')
193     if world is not None:
194         world.destroy()
195
196     pygame.quit()
197
198     while True:
199         clock.tick_busy_loop(60)
200         if controller.parse_events(world, clock):
201             return
202         world.tick(clock)
203         world.render(display)
204         pygame.display.flip()
205
206         global start_time
207         global duration
208         current_time = time.time()
209
210         if current_time - start_time > duration:
211             break
212         world.get_state()
213
214         # time.sleep(0.3)
215         time.sleep(0.05)
```

Listing A.1: Carla Environment Data Gathering

Bibliography

[1] Ryan J. Harrington, Carmine Senatore, John M. Scanlon, and Ryan M. Yee. *The Role of Infrastructure in an Automated Vehicle Future*, June 2018.

[2] Nidhi Kalra and Susan M. Paddock. *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?*, RAND Corporation, 2016.

[3] National Highway Traffic Safety Administration (NHTSA). *Automated Vehicles for Safety*, U.S. Department of Transportation, 2020.

[4] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. *On a Formal Model of Safe and Scalable Self-driving Cars*, arXiv preprint arXiv:1708.06374, 2017.

[5] Christopher Hart. *Understanding SAE Automated Driving Levels: Opportunities and Challenges*, The National Academies of Sciences, Engineering, and Medicine Workshop, 2019.

[6] Philip Koopman and Michael Wagner. *Challenges in Autonomous Vehicle Testing and Validation*, SAE Int. J. Trans. Safety 4(1):15–24, 2016.

[7] Bryant Walker Smith. *Automated Driving and Product Liability*, Michigan State Law Review, 2017.

[8] International Transport Forum (OECD). *Autonomous Driving: Technical, Legal and Social Aspects*, Corporate Partnership Board Report, May 2015.

[9] Erik Coelingh, et al. *Advanced Active Safety and Autonomous Drive*, Volvo Car Group, White Paper, 2013.

[10] Waymo Team. *Waymo Safety Report: On the Road to Fully Self-Driving*, October 2020.

[11] Steven E. Shladover. *Connected and Automated Vehicle Systems: Introduction and Overview*, Journal of Intelligent Transportation Systems, 22(3):190–200, 2018.

[12] J. Ziegler, P. Bender, M. Schreiber, et al. *Making Bertha Drive—An Autonomous Journey on a Historic Route*, IEEE Intelligent Transportation Systems Magazine, 6(2):8–20, 2014.

[13] Raj Rajkumar, Charles Chan, et al. *A Comprehensive Overview of Autonomous Vehicles*, Handbook of Transportation, Springer, 2016.

[14] D. Gonzalez, J. Perez, V. Milanés, and F. Nashashibi. *A Review of Motion Planning Techniques for Automated Vehicles*, IEEE Transactions on Intelligent Transportation Systems, 17(4):1135–1145, 2016. (Section on sensor fusion and perception preprocessing)

[15] A. Broggi, M. Bertozzi, A. Fascioli, and M. Sechi. *A Modular System for Vision-Based Driver Assistance*, IEEE Transactions on Intelligent Transportation Systems, 6(1):62–72, 2005.

[16] B. Paden, S. Z. Yong, D. Yershov, and E. Frazzoli. *Motion Planning and Decision Making for Autonomous Vehicles*, Annual Review of Control, Robotics, and Autonomous Systems, Vol. 1, pp. 277–304, 2018.

[17] Heechul Yun, Kyungtae Kang, Hyunsoo Park, and Kang G. Shin. *Automated Vehicle Safety: Multi-Modal V2X-Based Decision Making Architecture*, ACM/IEEE Design Automation Conference (DAC), 2020.

[18] S. Chen, J. Hu, Y. Shi, L. Zhao, and X. Wang. *Vehicle-to-Everything (V2X) Services Supported by LTE-Based Systems and 5G*, IEEE Communications Standards Magazine, 1(2):70–76, 2017.

[19] Liang Liu, Fei Zheng, Qiang Chen, and Shunqing Zhang. *Edge Computing for Autonomous Driving: Opportunities and Challenges*, IEEE Network, 33(6):112–117, 2019.

[20] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella. *On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration*, IEEE Communications Surveys.

[21] L. Nkenyereye, D. Kim, and Y. Han. *Secure and Efficient Firmware Update Scheme for Internet-Connected Vehicles*, IEEE Internet of Things Journal, 6(3):5384–5395, 2019.

[22] A. Vahidi and A. Eskandarian. *Research Advances in Intelligent Collision Avoidance and Adaptive Cruise Control*, IEEE Transactions on Intelligent Transportation Systems, 4(3):143–153, 2003.

[23] Hyunggi Cho, Y. W. Seo, B. V. Kumar, and R. Rajkumar. *A Multi-Sensor Fusion System for Moving Object Detection and Tracking in Urban Driving Environments*, IEEE International Conference on Robotics and Automation (ICRA), 2014. (Relevant to lane detection and object-aware LKA)

[24] Ali Alizadeh and M. B. Menhaj. *Blind Spot Detection Using a Multi-Sensor Fusion Technique*, IEEE Transactions on Intelligent Vehicles, 2(1):22–32, 2017.

[25] S. Tsugawa. *Vision-Based Blind Spot Detection and Warning System*, IEEE Intelligent Vehicles Symposium, pp. 157–162, 2005.

[26] Noah J. Goodall. *Machine Ethics and Automated Vehicles*, In *Road Vehicle Automation*, Springer, pp. 93–102, 2014.

[27] Bonnefon, J.-F., Shariff, A., and Rahwan, I. *The Social Dilemma of Autonomous Vehicles*, Science, 352(6293):1573–1576, 2016.

[28] Javier B. Bernal, et al. *Cybersecurity for Connected and Autonomous Vehicles: Threats and Challenges*, IEEE Transactions on Intelligent Transportation Systems, 23(11):18514–18527, 2022.

[29] Nidhi Kalra and Susan M. Paddock. *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?*, RAND Corporation Report RR1478, 2016.

[30] S. Geyer, B. Klamann, M. Gresser, et al. *Model-Based Verification and Validation of Advanced Driver Assistance Systems Using Simulation*, SAE Technical Paper 2014-01-0200, 2014.

[31] Javier B. Bernal, et al. *Cybersecurity for Connected and Autonomous Vehicles: Threats and Challenges*, IEEE Transactions on Intelligent Transportation Systems, 23(11):18514–18527, 2022.

[32] A. Loos, T. Kuhnt, and C. Stiller. *Validation of Automated Driving using Simulation and Model-Based Testing*, In *Handbook of Intelligent Vehicles*, Springer, pp. 1315–1339, 2020.

[33] T. T. Nguyen, T. Debus, J. Stein, and R. Isermann. *Hardware-in-the-Loop Simulation for Control Algorithm Development in Driver Assistance Systems*, SAE Technical Paper 2008-01-0595, 2008.

[34] D. Schäfer, M. Lauer, and C. Stiller. *Simulative Validation of Environment Perception Algorithms Using Real Sensor Data Reprojection*, IEEE Intelligent Vehicles Symposium (IV), 2019.

[35] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. *CARLA: An Open Urban Driving Simulator*, Proceedings of the 1st Annual Conference on Robot Learning (CoRL), PMLR, 2017.

[36] P. Bhattacharyya, S. Sankaranarayanan, and S. M. Mohan. *A Modular Deep Reinforcement Learning Framework for Autonomous Driving Using CARLA*, IEEE Transactions on Intelligent Vehicles, 6(4):696–709, 2021.

[37] C. Choi, M. Jeong, H. Kim, and Y. H. Lee. *Simulink-Based Controller Design and Validation for Autonomous Vehicles Using CARLA*, International Conference on Control, Automation and Systems (ICCAS), 2021.

[38] A. Mueller, A. Dosovitskiy, B. Ghanem, and V. Koltun. *Sim4CV: A Photo-Realistic Simulator for Computer Vision Applications*, International Journal of Computer Vision (IJCV), 126(9): 861–879, 2018.

[39] LG Electronics America Research Center. *LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving*, https://www.lgsvlsimulator.com, Accessed 2025.

[40] Jason Kong, Mark Pfeiffer, Georg Schildbach, and Francesco Borrelli. *Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design*, IEEE Intelligent Vehicles Symposium (IV), pp. 1094–1099, 2015.

[41] H. Peng, B. Zhang, and M. Tomizuka. *A Review of Modeling Approaches for Autonomous Driving*, Annual Reviews in Control, 50:64–83, 2020.

[42] Karl J. Åström and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*, Princeton University Press, 2nd Edition, 2021.

[43] Rajesh Rajamani. *Vehicle Dynamics and Control*, Springer, 2nd Edition, 2011. (Chapter on bicycle model and lateral dynamics)

[44] Klaus Bogenberger, Bernd Lenz, et al. *Fundamentals of Vehicle Dynamics and Control: Modeling and Simulation of Road Vehicles*, Springer Vieweg, 2021.

[45] Thomas D. Gillespie. *Fundamentals of Vehicle Dynamics*, SAE International, 1992.

[46] T. L. Temple, J. A. Christian, and T. L. Martin. *Accuracy of the Bicycle Model for Lateral Control of Autonomous Vehicles*, IEEE Transactions on Intelligent Vehicles, 3(3):280–292, 2018.

[47] Mattia Boggio, Carlo Novara, and Matteo Taragna. *Trajectory Planning and Control for Autonomous Vehicles: A "Fast" Data-Aided NMPC Approach*, European Journal of Control, 74:100–110, 2023.

[48] Jason Kong, Mark Pfeiffer, Georg Schildbach, and Francesco Borrelli. *Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design*, IEEE Intelligent Vehicles Symposium (IV), pp. 1094–1099, 2015. (Compares NMPC with other controllers)

[49] Yassine Hammoudeh, Ahmad Al-Sabbagh, and Mohammed Abu-Faraj. *Autonomous Car Lateral Control Using Classical PID and Robust PID Controllers*, International Journal of Advanced Computer Science and Applications (IJACSA), 11(2): 2020.

[50] A. A. Aly, A. A. Hatab, and H. F. F. Mahmoud. *Design of a PID Controller for Autonomous Vehicle Path Tracking Using Simulink*, IEEE Conference on Intelligent Transportation Systems (ITSC), 2017.

[51] Tim Wenzel, Matthias Wulfmeier, and Ingmar Posner. *Nonlinear Model Predictive Control for Robust Path Tracking of Autonomous Vehicles*, IEEE Transactions on Intelligent Vehicles, 5(4):571–580, 2020.

[52] M. Diehl, H. J. Ferreau, and N. Haverbeke. *Efficient Numerical Methods for Nonlinear MPC and Moving Horizon Estimation*, In *Nonlinear Model Predictive Control*, Springer, pp. 391–417, 2009.

[53] James B. Rawlings, David Q. Mayne, and Moritz M. Diehl. *Model Predictive Control: Theory, Computation, and Design*, Nob Hill Publishing, 2nd Edition, 2017. (Chapter 7: Nonlinear MPC)

[54] Iman Askari, Babak Badnava, Thomas Woodruff, Shen Zeng, and Huazhen Fang. *Sampling-Based Nonlinear MPC of Neural Network Dynamics with Application to Autonomous Vehicle Motion Planning*, arXiv preprint arXiv:2205.04506, 2022.

[55] Jun-Ting Li, Chih-Keng Chen, and Hongbin Ren. *Time-Optimal Trajectory Planning and Tracking for Autonomous Vehicles*, Sensors, 24(11):3281, 2024. DOI: 10.3390/s24113281

[56] Mattia Boggio, Carlo Novara, and Michele Taragna. *Nonlinear Model Predictive Control: an Optimal Search Domain Reduction*, IFAC-PapersOnLine, 56(2):145-150, 2023.