POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Design of a NMPC System for Automated Driving and Integration into the CARLA Simulation Environment

Supervisors

Prof. CARLO NOVARA

Prof. FABIO TANGO

Prof. MATTIA BOGGIO

Candidate

STEFANO CATOZZI

December 2024

Abstract

The development of autonomous vehicles is one of the most significant and promising technological challenges of the modern era, with substantial potential benefits in terms of road safety, traffic efficiency, and environmental sustainability. However, the complexity of the control systems required for fully autonomous driving demands advanced approaches capable of managing dynamic scenarios and variable operating conditions. In this context, simulators play a crucial role in testing and validating these technologies by providing a safe and controlled environment to replicate diverse driving scenarios without real-world risks. This thesis presents the design of a control system for autonomous driving implemented in MATLAB, co-simulated with the CARLA simulator. The project focuses on the implementation of a Nonlinear Model Predictive Control (NMPC) for advanced path tracking and decision-making functions in realistic and dynamic scenarios. The choice of NMPC is motivated by the need to ensure high performance in nonlinear contexts, while also considering strict constraints on vehicle states and inputs. To enable seamless integration of the NMPC controller, this thesis developed several key functionalities within the control system, including vehicle output data analysis, optimal trajectory generation based on velocity and path curvature, and the manipulation of input signals to govern the vehicle's behavior. Additionally, system identification was employed to examine the longitudinal dynamics of the vehicle under various throttle and braking conditions, enhancing the control system through a dispatching function that converts desired acceleration into appropriate input commands for the vehicle. A detailed analysis of the NMPC controller's tuning parameters was conducted to achieve a balance between tracking accuracy, robustness, and computation time—critical aspects for real-time system implementation. Moreover, the interface between Matlab and CARLA was significantly improved compared to previous studies, ensuring the controller's proper integration into the CARLA environment. The co-simulation between MATLAB and CARLA provides a realistic and modular testing environment, allowing the simulation of different traffic scenarios and variable road conditions. The results demonstrate how the NMPC controller, supported by the newly developed system functions, can effectively adapt to diverse operating conditions, optimizing the vehicle's trajectory and enhancing real-time decision-making capabilities. This work represents a significant step toward integrating advanced predictive control techniques in autonomous driving systems, highlighting both the advantages and challenges of MATLAB-CARLA co-simulation for autonomous driving applications.

Keywords: Autonomous vehicles, Control system, Path tracking, Decision making, NMPC (Nonlinear Model Predictive Control), MATLAB, CARLA simulator, Co-simulation, Output data analysis, Trajectory generation, Input

Table of Contents

List of Figures

Listings

Acronyms

AV
Autonomous Vehicle

ADS
Automated Driving System

LiDAR
Light Detection And Ranging

ECU
Electronic Control Units

AI
Artificial Intelligence

ADV
Autonomous Driving Vehicle

SAE
Society of Automotive Engineers

DDt
Dynamic Driving Task

ODD
Operational Design Domain

GPS
Global Positioning System

ADAS

Advanced Driver Assistance Systems

DQN

Deep Q Networks

RNN

Recurrent Neural Network

V2X

Vehicle To Everything

VANET

Vehicular Ad hoc Networks

ICN

Information Centric Networking

VCC

Vehicular Cloud Computing

IoV

Internet of Vehicles

XIL

Everything In the Loop

MIL

Model In the Loop

SIL

Software In the Loop

PIL

Process In the Loop

HIL

Hardware In the Loop

I/O
: Input/Output

API
: Application Programming Interface

IMU
: Inertial Measurement Unit

SLAM
: Simultaneous Localization and Mapping

PID
: Proportional–Integral–Derivative

MPC
: Model Predictive Control

NPC
: Non Player Character

RGB
: Red, Green, and Blue

GNSS
: Global Navigation Satellite System

CARLA
: Car Learning to Act

SUMO
: Simulation of Urban MObility

LGSVL
: LG Silicon Valley Lab Simulator

FMI
: Functional Mockup Interface

Sim4CV
: Simulation for Computer Vision

UE4
: Unreal Engine 4

UAV
: Unmanned Aerial Vehicle

NMPC
: Nonlinear Model Predictive Control

LQR
: Linear Quadratic Regulator

OCP
: Optimal Control Problem

VehIL
: Vehicle In the Loop

PFT
: Preview-Follower Theory

SMPC
: Stochastic Model Predictive Control

GMM
: Gaussian Mixture Model

CAV
: Connected and Automated Vehicles

FAV
: Functional Architecture View

SQP
: Sequential Quadratic Programming

QP

Quadratic Programming

NLP

Nonlinear Programming

SMC

Sliding Mode Controller

MIMO

Multiple Input Multiple Output

LQRY

Linear Quadratic Regulator with Output Weighting

TM

Traffic Manager

ALSM

Agent Lifecycle & State Management

PBVT

Path Buffers & Vehicle Tracking

DST

Dynamic Single Track

CTE

Cross Track Error

RMSE

Root Mean Square Error

Chapter 1

Introduction

1.1 Autonomous Driving

Autonomous vehicles (AVs), also known as self-driving cars and driver-less cars, are vehicles that are able to perceive the environment in which they're in and navigate through this environment without the help or intervention of a human being [1]. Since the middle 1980s, Automated Driving Systems (ADSs) have been studied and developed by many universities, research centers, car companies, and entities of other industries around the world [2] with the promise of preventing accidents, increase efficiency in transportation, reduce congestion, lower emissions, transporting the mobility-impaired and reducing driving related stress. Autonomous vehicles are categorized based on their level of automation, which refers to the degree to which they can operate without human intervention. The classification system, defined by the Society of Automotive Engineers (SAE) in 2014, provides the most widely accepted framework for understanding the various stages of automation in vehicles. They developed a harmonized system to describe six degrees of automated driving, it ranges from Level 0 (no automation) to Level 5 (full automation). This system helps differentiate respectively between vehicles that assist drivers and those capable of completely autonomous driving in all conditions. For better presenting the SAE classification system, some terms need to be totally clarified. Those terms were shaped by SAE with the best efforts to fully cover the automated system terms and his uses and do not let any lacuna for double interpretation. For instance, the Dynamic Driving Task (DDT) encompasses all functions required to operate a vehicle in on-road traffic, while the Operational Design Domain (ODD) defines the specific conditions under which the system or feature is intended to operate. The Automated Driving System refers to the combination of hardware and software capable of performing the entire DDT. The DDT Fallback, or Dynamic Driving Task Fallback, is the response by either the driver or the ADS to perform the

DDT or to bring the vehicle to a minimal risk condition. The driving automation system's manufacturer is responsible for defining the system's requirements, ODD, operating characteristics, appropriate use cases, and the system's level of driving automation. To determine the automation level of a driving system, aspects like the system's functionalities, the roles of DDT, and the DDT fallback are considered. According to SAE's six automation levels, as shown in Figure 1.1, levels 1 and 2 refer to cases in which the (human) driver continues to perform part of the DDT while the driving automation system is engaged. The upper three levels of driving automation (3-5) refer to cases in which the Automated Driving System performs the entire the DDT on a sustained basis while it is engaged [3].

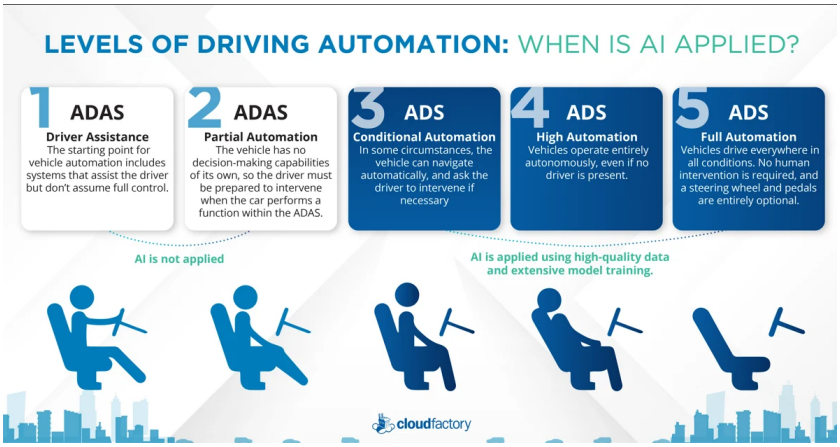


Figure 1.1: SAE Autonomous levels

- At level 0 the human driver is entirely responsible for controlling the vehicle at all times. There might be automated systems like warning alerts or emergency braking assistance, but they do not control the vehicle autonomously;
- At Level 1 (driver assistance) vehicle provide basic automation features like adaptive cruise control or lane-keeping, during a specific condition the system does either the lateral or the longitudinal motion control but never the both simultaneously, and the driver performs the remaining tasks of it;
- At Level 2 (partial driving automation) during a specific condition, the system does both the lateral and the longitudinal motion providing simultaneously steering and acceleration/deceleration, such as on highways, and the driver supervise the system and performs other tasks;
- At Level 3 (conditional driving automation) during a specific condition, the system performs the entire DDT excepting to the DDT fallback, which is done by the user if the system requires when the vehicle encounters situations it

cannot handle or if the DDT present any relevant failure. By this level the feature is capable of performing the entire DDT in low-speed, stop-and-go and freeway traffic;

- At Level 4 (high driving automation) during a specific condition, the system does the entire DDT and DDT fallback, without expecting that the user will respond to intervene if requested. At this level, the feature performs the entire DDT operation on a motorway or freeway but it may require human control in complex or undefined environments like rural roads;
- At Level 5 (full driving automation) there is no specific condition to the system operates, the feature must be capable of doing the entire DDT and DDT fallback, without expecting that the user responds a request to intervene. At this level, the system is capable of guiding the vehicle throughout complete trip, regardless of the starting and ending points or intervening road, traffic and weather condition. There is no need for steering wheels or pedals, as the system is designed to operate independently in all situations.

To be capable of navigating and operating without human intervention, an autonomous car has to rely on a combination of advanced hardware and software technologies to perceive their environment, process data in real-time, make driving decisions and control the vehicle to execute maneuvers [4]. Example of these technologies, as can be seen on a Waymo autonomous vehicle in Figure 1.2, are sensors, cameras, radars, LiDARs, electronic control units (ECUs), artificial intelligence (AI) and control systems. In particular, the system software architecture in

Figure 1.2: Waymo autonomous vehicle

Autonomous Driving Vehicles (ADVs) plays a critical role in ensuring that all the vehicle's software components work seamlessly together to achieve safe, reliable, and efficient autonomous driving. The architecture serves as the blueprint that

defines how different software modules are organized, how they communicate, and how they handle various tasks such as perception, decision-making, and control. Equally important are the software functions that operate within this architecture. These functions, such as sensor fusion, object detection, path planning, and vehicle actuation, form the core processes that enable autonomous operation. Each function must be precisely designed, implemented, and tested to ensure the vehicle can perceive its environment, make informed decisions, and execute appropriate actions in real time. Any failure in these functions could compromise the safety and reliability of the system. A well-designed software architecture not only provides a robust framework for integrating these critical functions but also ensures that they interact efficiently and reliably under diverse operating conditions. Moreover, it facilitates scalability, allowing the addition of new functionalities or improvements to existing ones, and supports rigorous testing and validation processes, which are essential for deployment and long-term maintenance of autonomous driving systems.

Finally, developing autonomous driving for everyday use faces several challenges, including ensuring safety in complex urban environments, handling unpredictable behaviors from human drivers and pedestrians, and addressing regulatory and public acceptance issues. Despite these obstacles, autonomous cars are already being used for passenger transport in cities, primarily in pilot programs or as autonomous taxis. Companies like Waymo (USA), Cruise (USA), and Baidu (China) operate in selected areas with well-mapped streets and compatible infrastructure. In cities such as Phoenix, San Francisco, and Beijing, these vehicles are now available for commercial services or ride-hailing.

1.2 AV System Architectures

The architectures of ADSs can be classified as either standalone, ego-only systems (where ego stands for the controlled vehicle), or connected multi-agent systems. Additionally, these design principles are implemented through two different approaches: modular or end-to-end driving[5].

EGO-ONLY SYSTEMS The ego-only approach up to now is the most widely used among the state-of-the-art ADSs. The philosophy is to carry all of the necessary automated driving operations on a single self-sufficient vehicle at all times, whereas a connected ADS may or may not depend on other vehicles and infrastructure elements given the situation. This lead to a more easily-manageable and practical situation in which design and development are done on a self-sufficient platform with the possibility to be opened to additional challenges of connected systems.

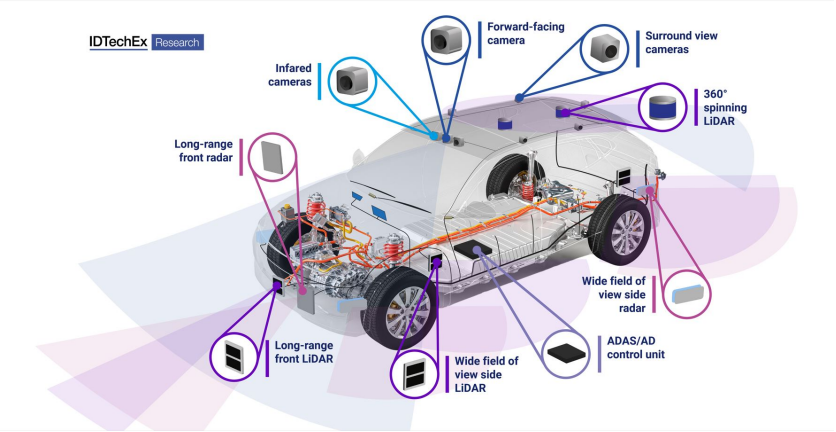


Figure 1.3: Example of AV sensors positions

MODULAR SYSTEMS Modular systems are characterized by a cascade of separate components connecting sensory inputs, as shown in Figure 1.3, to actuator outputs [6]. The main functions of a modular ADS can be synthesized as: localization and mapping, perception, assessment, planning and decision making, vehicle control, and human-machine interface. In a typical pipelines, as can be seen in Figure 1.4, raw sensor inputs are fed to localization and object detection modules, followed by scene prediction and decision making. Next, the control module generates motor commands applied at the end of the stream to the vehicle. This structure allow to develop individual modules separately splitting into an easier-to-solve set of problems the challenging task of automated driving. The major advantage of this is that these sub-tasks have their corresponding literature in robotics [7], computer vision [8] and vehicle dynamics [9], allowing direct transfer of accumulated know-how and expertise. Furthermore in a modular design, functions and algorithms can be integrated or built upon each other. E.g, a safety constraint [10] can be implemented on top of a sophisticated planning module so that, in case of emergency, can force some hard-coded rules without modifying the inner workings of the planner. As a result the design is redundant but architecture is reliable. On the other side the major disadvantages of modular systems are being prone to error propagation and over-complexity. The first ADS related fatality was caused by an error in the perception module of a Tesla in the form of a misclassification of a white trailer as sky, propagated down the pipeline until failure [11].

END-TO-END DRIVING End-to-end driving approach generates ego-motion directly from sensory inputs, without any intermediate processing, as for example in Figure 1.5. The three main techniques for end-to-end driving are: direct supervised deep learning, neuro-evolution and the more recent

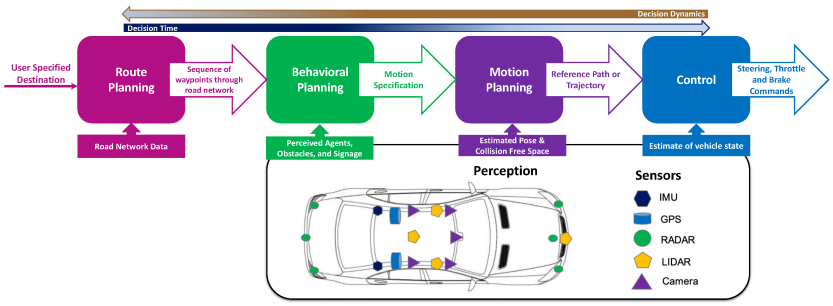


Figure 1.4: Pipeline software architecture

deep reinforcement learning. In direct supervised deep learning a connected networks model is trained typically from image inputs. In this case the ground truth, thus the real-world data used as a benchmark to evaluate the model behavior, is the ego-action sequence of an expert human driver. As consequence, since the appropriate driving actions in the perception indicators are generated by a separate module, this approach is not fully end-to-end. Artificial neural network research advances were crucial to implement deep convolutional and temporal networks in automated driving tasks. In [12] was proposed a deep convolutional neural network that takes image as input and outputs steering commands. In [13] a spatiotemporal network was developed for predicting ego-vehicle motion. Another convolutional model that tries to learn a set of discrete perception indicators from the image input is DeepDriving [14]. Different is the approach of deep reinforcement learning model, Deep Q Networks (DQN), where reinforcement learning is combined with deep learning [15]. Here actions are generated first with random initialization, then the aim of the network is to select a set of actions that maximize cumulative future rewards. Where the optimal action reward function is approximated by a deep convolutional neural network. The model adjust its parameters with experience instead of direct supervised learning. An automated driving framework using DQN was introduced in [16], and then tested in a simulation environment. While in a countryside road without traffic was achieved the first real world run with DQN [17]. DQN based systems learn the optimum way of driving instead of imitate the human driver. Neuroevolution driving approac instead is not popular as DQN and direct supervised learning. The goal in neuroevolution is use evolutionary algorithms to train artificial neural networks [18]. Real world end-to end driving with neuroevolution is not achieved yet based on our current best knowledge. However, some promising simulation results were obtained. ALVINN (Autonomous Land Vehicle In a Neural Network), that was one of the earliest successful demonstrations of using neural networks for real-time

decision-making in self-driving cars in the end of 80's [19], was trained with neuroevolution and outperformed the direct supervised learning version [20]. A Recurrent Neural Network (RNN) was trained with neuroevolution in [21] using a driving simulator. The biggest advantage of neuroevolution is that backpropagation can be removed, thus direct supervision is no more needed. End-to-end driving shows promise, but its application in real-world urban environments remains limited, with only a few controlled demonstrations. The main challenges of end-to-end driving include a lack of built-in safety mechanisms and limited interpretability [22]. Additionally, approaches like Deep Q-Networks and neuroevolution have a significant drawback compared to direct supervised learning: they require continuous online interaction with the environment and rely on trial and error to learn the desired behaviors, and failing could lead to severe incidents. In contrast, direct supervised learning networks can be trained offline using human driving data, ensuring that, once trained, the system should operate without failures.

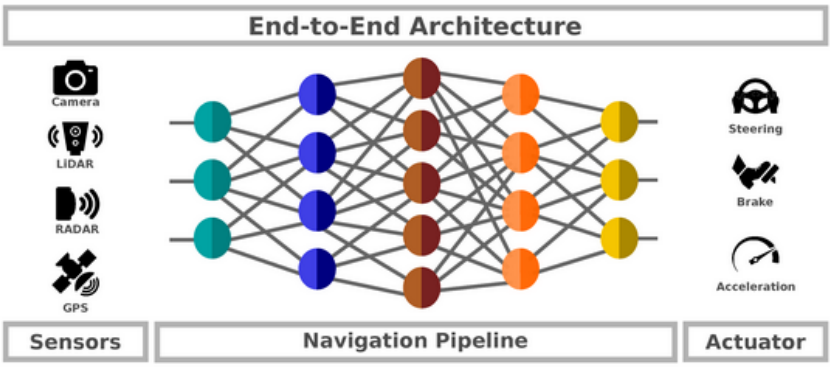


Figure 1.5: End-to-end software architecture

CONNECTED SYSTEMS Some researchers believe that the future of driving automation will be operational connected ADSs. In this emerging technology, not in use yet, the basic operations of automated driving can be distributed among agents. V2X, or "vehicle to everything", enables a vehicle to access a vast amount of data from various sources, such as pedestrians' mobile devices and stationary sensors on traffic lights [23], example shown in Figure 1.6. By sharing detailed traffic network information, V2X can help overcome the limitations of ego-only systems, including restricted sensing range, blind spots, and computational constraints. More V2X applications are anticipated to improve safety and traffic efficiency in the near future [24]. Vehicular Ad hoc Networks (VANETs) can be implemented through two main approaches: conventional IP-based networking and Information-Centric Networking (ICN)

[25]. In vehicular contexts, where data needs to be shared among agents that may experience intermittent or suboptimal connections while maintaining high mobility, conventional IP-based protocols are often insufficient [26]. Unlike IP-based networking, ICN allows vehicles to broadcast query messages to a general area rather than targeting a specific address, and they can accept relevant responses from any available source [27]. This approach is better suited to the high mobility and dispersion of vehicles on road networks, where the identity of the information source is less important. Furthermore, local data often holds greater relevance for immediate driving tasks, such as avoiding a rapidly approaching vehicle in a blind spot. Early studies, like the CarSpeak system [28], demonstrated that vehicles can leverage each other's sensors and shared information to perform dynamic driving tasks. However, to make information sharing feasible across hundreds of thousands of vehicles in a city, it is essential to reduce the vast amounts of continuous driving data being exchanged. A semiotic framework was proposed in [29] to integrate various information sources and transform raw sensor data into meaningful insights. In [30], the concept of Vehicular Cloud Computing (VCC) was introduced, highlighting its advantages over traditional Internet cloud applications. The key distinction lies in sensor data management: in VCC, sensor data remains on the vehicle and is shared only when queried by another vehicle, potentially reducing the costs of continuous data transfer to the web. Additionally, the high relevance of local data enhances VCC's feasibility. Comparisons between traditional cloud computing and vehicular cloud computing indicated that VCC is technically feasible [31]. The term "Internet of Vehicles" (IoV) was introduced to describe a connected Automated Driving System, while "vehicular fog" was presented in [27]. Creating an efficient Vehicular Ad hoc Network with thousands of vehicles in an urban setting poses considerable challenges. For Information-Centric Networking -based VANETs, key issues include security, mobility, routing, naming, caching, reliability, and multi-access computing [32]. Despite the significant potential benefits of a connected vehicular system, these challenges greatly increase the system's complexity, and no fully operational connected system has been achieved yet.

1.3 Simulation's Crucial Role in ADS Testing

Testing autonomous driving vehicles is a crucial step in ensuring the safety, reliability, and efficiency of the complex systems involved in making a self-driving vehicle. However, testing algorithms for autonomous vehicles in real world is an expensive and time consuming process. Also, in order to utilize recent advances in machine intelligence and deep learning we need to collect a large amount of annotated

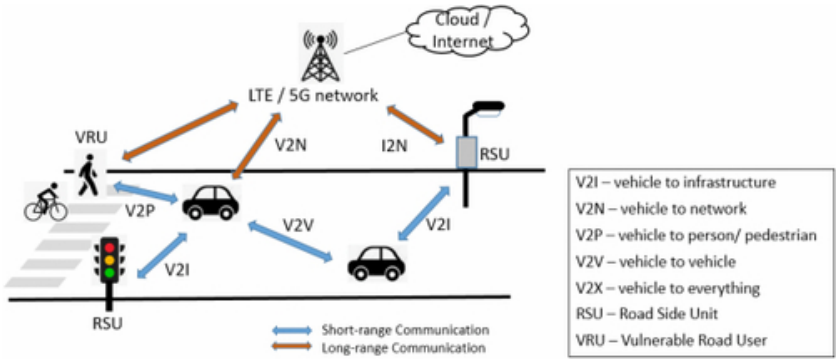


Figure 1.6: Vehicle to everything system

training data in a variety of conditions and environments [33] where certain type of infrastructure can not be present and with conditions not always reproducible. This issue is enhanced by the fact that during the training phase autonomous vehicles are often unsafe, jeopardizing the safety of public. Simulation plays a key role in supplementing and accelerating the real world testing. It allows to test scenarios that are otherwise highly regulated on public roads because of safety concerns [34]and is reproducible, scalable and cuts the development without extra cost required. Example of simulation based testing framework in Figure 1.7.

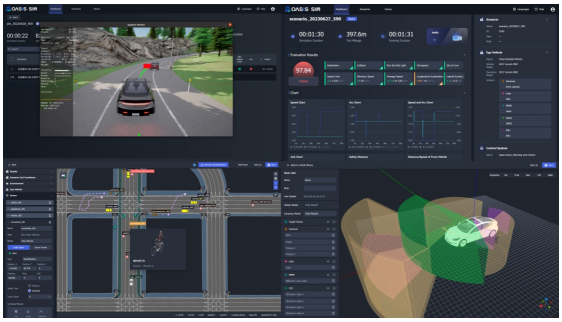


Figure 1.7: CARLA simulator advanced interface with Synkrotron Oasis

Due to the high stakes associated with allowing machines to make decisions in real-time traffic scenarios, ADV testing is multi-faceted, rigorous, and involves several phases, from virtual simulation to real-world deployment. The goal is to verify that all hardware, software, and decision-making algorithms function correctly in various conditions and edge cases. Standard ISO 26262, published in November 2011, defines the safety aspects of the development, from design to test, of electric and electronic automotive systems. In Figure 1.8 the ISO 26262 V-cycle Development Process. A significant innovative approach for system safety of ISO 26262 is that it recommends at each design stage tests and fault

injection [35]. At each step of design a specific approach using only software or

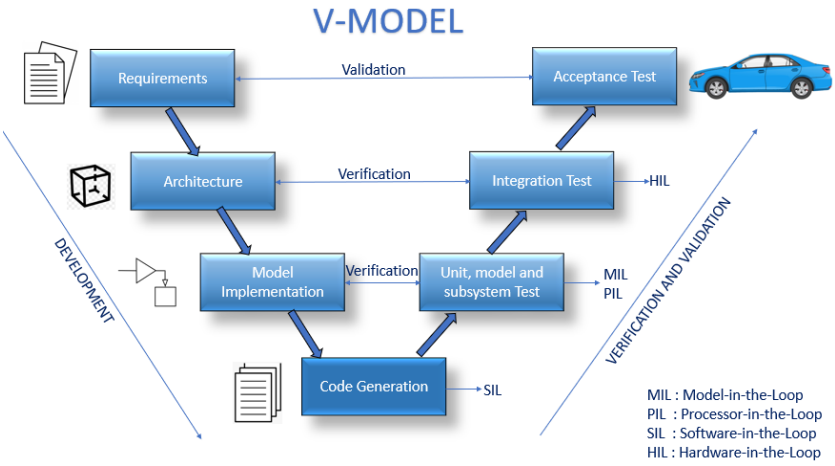


Figure 1.8: V-Cycle development process ISO 26262

both, software and hardware, is used to test the designed controller according to the predefined requirements. In addition, the tests should cover most of the potential operating conditions, and cover all the parts of the developed control logic. Tests are conducted by integrating the control logic in a loop with the plant model, known as X-In-the-Loop (XIL) testing. XIL encompasses several types of tests, including Model-In-the-Loop (MIL), Software-In-the-Loop (SIL), Processor-In-the-Loop (PIL), and Hardware-In-the-Loop (HIL) testing. In particular:

MIL MIL tests are used during control system development. In this stage, the plant is modeled and used to provide the signals required by the controller. Both controller and model are developed and run on the same computer. At this step any vital problem in the control logic can be detected .

SIL The developed control model is compiled to executable code into a system-function (S-Function) block. This S-Function block is connected to the plant and will be tested with the same tests designed for MIL. This approach increases simulation speed and it's useful to debug and analyze the generated code and controller performance. Both controller and plant are still on the same computer. SIL is the most cost-effective method for testing production code without requiring any physical hardware. However, low-level processes such as ECU communication, CAN, etc. are not modeled in a typical SIL environment [36].

PIL In PIL, the control model is compiled, and the resulting production code is loaded onto the embedded target processor. Unlike MIL and SIL, where everything is typically run on a single PC, in PIL, the controller and plant

model are hosted on two separate machines. The model running on the PC interacts with the software on the processor via a communication link, such as standard Ethernet. PIL tests do not involve any physical I/O devices like sensors or actuators.

HIL In HIL both the model and controller are compiled, and the generated code is uploaded to the respective hardware components. The controller's code is loaded onto the Electronic Control Unit, while the plant's code is uploaded to the model simulator. HIL setups enable the connection to external inputs from real physical systems, such as steering wheel commands or brake/gas pedal positions. Additionally, HIL tests include a physical component under test, which could be an actuator, sensor, or a vehicle part like the engine, transmission, or battery pack. In an HIL test, the plant is a combination of the components under test and the model running in the simulator. There are two types of input/output (I/O) in HIL: virtual I/O and physical I/O. Virtual I/O refers to the communication between the ECU and the simulator via CAN bus, while physical I/O involves CAN communication between the ECU and physical components, such as the ECU interacting with actuators, sensors, or other vehicle systems [37].

Now, it's essential to consider which types of features can be effectively tested through simulation. According to [38], five distinct categories of accident exposure factors have been identified in ADS testing:

1. Failure of components and hardware deficiencies
2. Deficiencies in sensing road, traffic and environmental conditions
3. Deficiencies in control algorithms (complex and difficult situations)
4. Behavior dependent accidents (adequate behavior and rule compliance)
5. Faulty driver and vehicle interaction (mode confusion and false commanding)

The robustness of components and hardware (category 1) must be assessed through bench testing and on proving grounds. For Hardware-In-the-Loop testing, an environment simulation is necessary. While these tests align with standard automotive component testing, they now demand lower acceptable failure rates and stricter safety integrity level requirements. The testing of category 2 environment sensors is influenced by factors like weather, environmental conditions, visibility, and lighting. Field tests are essential to assess sensor performance under diverse real-world conditions. For standardized performance assessments, specialized test areas can provide the controlled environments needed for certification tests. If suitable sensor models are developed based on field test results —simulating sensor responses under

various conditions— sensor fusion algorithms can then be evaluated in simulations. Category 3 testing focuses on control algorithms under challenging driving scenarios, with Software-in-the-Loop simulations being a central part of this process. When combined with Hardware-in-the-Loop tests, which incorporate actuator responses, the testing procedure can be completed before verifying the entire system through proving ground tests. Compliance with rules and appropriate behavior (category 4) in interactions with infrastructure and during specific scenarios, such as the presence of emergency vehicles, can be partially evaluated through Software-in-the-Loop simulations. However, full verification of behavior requires additional testing on proving grounds and in real-world field conditions. Validating the interaction between driver and vehicle (category 5) —particularly focusing on issues like mode confusion and unintended commands— is a primary application for driving simulator studies, where driver-in-the-loop simulations provide a controlled environment to observe these interactions. Following simulator studies, proving ground and field tests provide further insights and complete this phase of testing. SIL test environments and driving simulators represent distinct hardware setups, each with specific focuses and challenges. However, the core requirement for both—whether in SIL or driver-in-the-loop studies—is the simulation and precise control of the traffic environment to ensure consistent, reproducible testing conditions [39].

1.4 Simulator's Technical Requirements

A good automotive simulator must meet several critical requirements to effectively simulate autonomous vehicles, driving scenarios, and the interaction between various vehicle components [40]. These requirements ensure that the simulator can accurately model real-world driving conditions, vehicle dynamics, and system behavior. Below are some key requirements desired from any good automotive simulator:

1. Game Engine: Automotive simulators are often developed as extensions of game engines like Unreal Engine and Unity [41], which offer frameworks for rendering, physics, and scripting. Their advanced 3D graphics capabilities make them ideal for creating realistic environments that simulate vehicle interactions, road scenarios, and the behavior of other road users;

2. Perception: A crucial element of self-driving cars is perception—the system's ability to interpret its surroundings using various sensors like cameras, LiDAR, radar, GPS, and IMU [42]. These sensors provide data, which is processed by software for decision-making. To test perception systems effectively, simulators require realistic sensor models or real sensor data integration, allowing researchers to validate methods like sensor fusion [43] and optimize sensor placement in real vehicles;

3. Simultaneous Localization and Mapping (SLAM) is a key component of autonomous driving systems, responsible for creating maps of unknown environments and tracking the vehicle's location within these maps. To effectively support SLAM applications, simulators must provide camera calibration data, including intrinsic and extrinsic features. This information enables the SLAM algorithm to utilize multi-view geometry for estimating camera position and localizing the autonomous system within a global map;

4. Path Planning: Path planning involves determining a route for a mobile agent to navigate autonomously while avoiding collisions. For autonomous vehicles, this process builds on research in mobile robotics, distinguishing between global and local planning. Global planners use static maps of the environment, while local planners adapt to the agent's immediate surroundings. Various planning algorithms [44], such as A*, D*, and RRT [45], are essential for developing these planners. Therefore, simulators must provide built-in mapping functions or interfaces for importing maps, as well as the capability to program customized algorithms;

5. Vehicle control: control involves executing a planned collision-free trajectory using control inputs like throttle, brake, and steering [46], monitored by closed-loop control algorithms [47]. Common algorithms include Proportional–Integral–Derivative (PID) and Model Predictive Control (MPC) [48]. For effective implementation of these intelligent control algorithms, simulators must be able to create vehicle dynamic models and program the algorithms mathematically;

6. 3D Virtual Environment: To effectively test a vehicle's functional elements, simulators require a realistic 3D virtual environment. This includes both static objects, like buildings and trees, and dynamic ones, such as vehicles and pedestrians [49], which must behave realistically. Simulators can create these environments using game engines or HD maps of real locations. They should also support various terrains and weather conditions. The level of detail varies by application; companies like Uber and Waymo may use simpler environments [50] if they do not rely on simulations for perception model testing, while high-detail environments are essential when testing perception systems;

7. Traffic infrastructure: Simulations should include traffic infrastructure elements like traffic lights and road signs [51] to regulate traffic and ensure safety for all road users. While infrastructure may evolve to support connected vehicles in the future [52], current self-driving cars must still follow traditional traffic rules, similar to human drivers;

8. Traffic Scenarios Simulation: The ability to create diverse traffic scenarios is crucial in a simulator, enabling researchers to re-create real-world situations and explore "what-if" scenarios that may be unsafe to test otherwise. This requires support for a wide range of dynamic agents—such as pedestrians, bicycles, and various vehicle types—and a flexible API for managing complex elements like agent behaviors, crashes, weather, and traffic controls;

9. 2D/3D Ground Truth To generate training data for AI models, the simulator should supply object labels and bounding boxes for items within the scene. Each video frame from the sensors should display objects enclosed within bounding boxes;

10. Simulators require several non-functional qualities to support diverse testing needs. Comprehensive documentation is essential for usability, ensuring smooth updates with mappings for API changes to maintain compatibility. Flexibility and modularity are also important, as open-source simulators should allow developers to customize scenarios, sensors, and agents quickly. Portability across different operating systems enhances accessibility and saves time. Additionally, a scalable server-client architecture enables multi-user simulations of complex scenes, such as traffic congestion. Open-source availability further fosters collaboration, shared learning, and collective advancements within the field. Finally, Co-simulation capability in a simulator allows it to integrate with other simulation tools, such as those for traffic generation, mobility, V2X communications, or autonomous driving platforms like Autoware [53] and Baidu Apollo [54]. This integration expands the simulator's functionality, enabling support for additional features beyond its original scope.

1.5 Simulators' State of Art

Numerous simulators are available for testing self-driving car software. Many are proprietary tools developed by autonomous driving companies, such as Waymo's CarCraft and SurfelGAN, Cruise's Webviz and The Matrix, and Uber's DataViz. Other simulators are open-source, as Gazebo, CARLA, LGSVL, AirSim, AWSIM, DeepDrive, and sim4CV, they are widely utilized in both academic and industrial research. They are freely accessible, with open code that can be extended, adapted, and supported by the community, making them ideal for academic researchers and smaller teams with limited resources, as well as for industry use. Commercial simulators instead, like MATLAB/Simulink, CarSim, PreScan, NVIDIA DRIVE Sim, Cognata and IPG Automotive CarMaker, are more widely accessible and offer a broader range of functionalities, with ongoing support through patches and updates that improve their reliability. Probably one of the most widely

used is SCANeR Studio V, a modular simulation software suite developed by AVSimulation for designing, testing, and validating vehicle systems. It is widely used in the automotive industry, particularly in research and development of ADAS, autonomous driving, and vehicle dynamics. However, as paid solutions, commercial simulators can be costly, making them more feasible for industry use, where budgets are typically larger, than for academic researchers [55].

Following in this section three of the most used and developed open source simulators up to now will be presented, which were candidates for this thesis work. The feature in which we were mainly interested were ability to provide realistic 3D environments, graphic quality, accuracy of the physics engine, sensor simulation, simulation of traffic and pedestrians, weather conditions, and the ability to simulate at different times of the day.

1.5.1 CARLA

(a) *CARLA logo* **(b)** *CARLA environment*

Figure 1.9: CARLA logo and environment

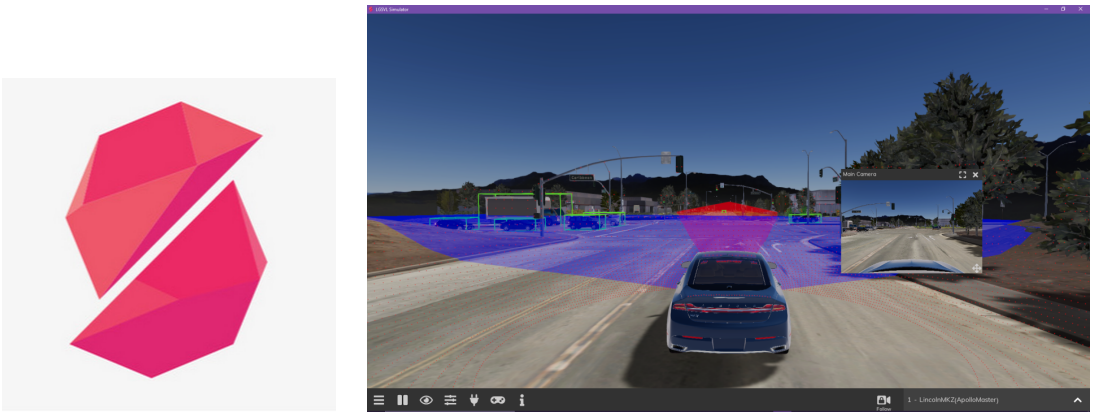
CARLA (Car Learning to Act) [56] is an open-source simulator that has been developed to support training, prototyping, and validation of autonomous driving models, including both perception and control. The simulator is developed based on the Unreal Engine [57]. The engine provides state-of-the-art rendering quality, realistic physics, basic NPC logic, and an ecosystem of interoperable plugins. CARLA simulates a dynamic environment and offers an intuitive interface for agents interacting with this world. It operates on a scalable client-server architecture, where simulation tasks are handled by the server. This includes scene rendering, world-state updates, sensor rendering, and physics calculations. To achieve realistic results, the server should be equipped with a dedicated GPU. The client API is implemented

in Python and is responsible for the interaction between the autonomous agent and the server via sockets. It is customizable by users and provides control over the simulation. The CARLA API is based on Python and C++. CARLA simulates a wide range of vehicle sensors, critical for autonomous driving research, including: Cameras (RGB, depth, semantic segmentation, infrared), LiDAR, GNSS (GPS), IMU and Ultrasonic sensors. These sensor models replicate real-world performance, allowing users to test perception, object detection, tracking, and sensor fusion algorithms. Also a variety of atmospheric conditions and illumination, as in Figure 1.9 regimes are implemented. These differ in the position and color of the sun, the intensity and color of diffuse sky radiation, as well as ambient occlusion, atmospheric fog, cloudiness, and precipitation. Therefore, CARLA tries to meet the requirements of various use cases of ADAS, for instance training the perception algorithms or learning driving policies. It leverages the OpenDRIVE(citare) standard to define roads and urban settings. The simulator supports dynamic traffic with AI-driven vehicles, cyclists, and pedestrians. These actors follow traffic rules, respond to road conditions, and can be programmed for specific behaviors (e.g., pedestrians crossing the street unpredictably or vehicles swerving). Users can design custom traffic scenarios to test edge cases like near-accidents, emergency braking, and evasive maneuvers.

CARLA is designed with artificial intelligence and machine learning in mind, allowing the training of autonomous driving models in simulated environments. It can generate large amounts of data required for training perception systems, such as object detection and semantic segmentation models. It is also possible to create maps from scratch in CARLA using the unreal engine and the predefined features in the CARLA library such as buildings, traffic lights, roads, also customizing the predefined routes within the map. CARLA supports co-simulation, i.e., it can be used with other simulators. It has native support for Simulation of Urban MObility (SUMO) [58], CarSim, Matlab amd many others. Another significant feature of CARLA is its active GitHub community, which assists users not only in resolving bugs and addressing identified issues but also in providing guidance on how to effectively use the tool.

1.5.2 LGSVL

The LGSVL (LG Silicon Valley Lab Simulator) Simulator is a high-fidelity tool designed for autonomous driving and related systems. It has been integrated with the Autoware and Apollo autonomous driving stacks for comprehensive end-to-end testing, example in Figure 1.10, and can be easily adapted for other similar ADS. As an open-source simulation engine, it promotes an open ecosystem, allowing users to apply the LGSVL Simulator to various applications and contribute to its development. The simulator is also regularly updated to meet the evolving needs



(a) *LGSVL logo* **(b)** *LGSVL environment*

Figure 1.10: LGSVL logo and environment

of the user community.

LGSVL Simulator utilizes Unity's game engine for simulation and it's capable to to simulate photo-realistic virtual environments that match the real world taking advantage of the latest technologies in Unity. The set of Functions the simulation engine can process are divided into: environment simulation, sensor simulation, and vehicle dynamics and control simulation of an ego vehicle. Also traffic simulation as well as physical environment simulation, like weather and time-of-day, are included in the Environment simulation. Aspects that are vital components for test scenario simulation. All features and settings of environment simulation can be controlled through the Python API. The source code is available publicly on GitHub since The simulation engine of LGSVL Simulator is developed as an open source project, executable files can be found and downloaded for free use. The default simulator set of sensors to choose include camera, LiDAR, Radar, GPS, and IMU as well as different virtual ground truth sensors. Furthermore, Users can additionally build their own custom sensors and add them to the simulator as sensor plugins.

LGSVL Simulator supports a basic vehicle dynamics model for physic simulation of ego vehicles. Also, the vehicle dynamics system is set up to allow integration of external third party dynamics models through a Functional Mockup Interface (FMI) [59], shared libraries that can be loaded into the simulator, or separate IPC interfaces for co-simulation. As a result, users can join together LGSVL Simulator with third party vehicle dynamics simulation tools to take advantage of both systems. LGSVL allow the creation of synthetic 3D environments to be used in simulation, however can also replicate and simulate real world locations by creating a digital twin of a real scene from logged data (images, point cloud, etc.). Map annotations (like traffic lanes, lane boundary lines, traffic signals, traffic signs,

pedestrian walking routes) can then be followed by other agents that are part of a scenario (nonego vehicles, pedestrians, controllable plugin objects). This means that vehicle agents and pedestrians in simulation will be able to obey to traffic rules, as well as traffic lights, stop signs, lanes, and turns, following a annotated route. Finally, Python API allows users to control and interact with simulated environments. With deterministic physics, scripting allows for repeatable testing in simulation [60].

1.5.3 Sim4CV

(a) *Sim4CV logo* **(b)** *Sim4CV environment*

Figure 1.11: Sim4CV logo and environment

Sim4CV (Simulation for Computer Vision) is a photo-realistic simulator designed for training and evaluation, with wide-ranging applications in computer vision. Developed using the Unreal Engine, it features physics-based vehicles, unmanned aerial vehicles (UAVs), and animated human characters set in intricate 3D urban and suburban landscapes. The simulator includes an intuitive graphical user interface that allows users to easily adjust all pertinent settings. The sim4CV simulator features immersive environments for both driving and flying, including multiple vehicles such as two passenger cars, an RC truck, and two UAVs, along with several thoughtfully crafted maps. An external map editor enables users to create their own custom maps, as shown in Figure 1.11. Additionally, the communication interface allows for the integration of external programs, enabling them to receive images and vehicle state information from the simulator, as well as send control signals, compatible with languages like C++, Python, and MATLAB.

The simulator is built on Epic Games' Unreal Engine 4 (UE4), leveraging its modern game engine architecture for real-time rendering of not just RGB images

but also enabling the generation of pixel-level segmentation, bounding boxes, class labels, and depth information with minimal effort. Users can set up multiple cameras within a scene, attach them to actors, and programmatically adjust their positions during each rendering frame. This functionality facilitates the creation of synthetic data, including simultaneous multi-view rendering, stereoscopy, structure-from-motion, and view augmentation. Additionally, UE4 features an advanced physics engine that allows for the design and measurement of complex vehicle movements, enabling realistic simulations of moving objects along with detailed physics measurements at every frame. Lastly, the extensive compatibility with flight joysticks, racing wheels, game consoles, and RGB-D sensors enables human control and input, allowing for motion capture to be synchronized seamlessly with the visually and physically rendered environment.

One application of the Sim4CV simulator is the automatic generation of datasets with free ground truth data, enabling the evaluation of cutting-edge computer vision tracking algorithms "in-the-loop" under conditions that closely resemble the real world. The simulator allows for the automatic creation of virtual driving environments, ranging from small neighborhoods to entire cities, using an overhead view editor. Users can manipulate standardized blocks of various sizes to represent objects like roads, trees, and houses, or they can opt to create the road network randomly. This approach facilitates the easy generation of diverse training and testing environments. In contrast to many other autonomous driving simulators, Sim4CV places a strong emphasis on vision-based tasks. This focus makes it particularly well-suited for research areas such as SLAM, visual odometry, and object recognition [61].

1.6 Goal of the Thesis

In the rapidly expanding field of autonomous driving, it is essential to investigate new control methods that can enhance the performance and reliability of autonomous systems. Additionally to test these algorithms well robust platforms are needed to evaluate real-time integration and behavior, automotive simulators are state of art for test these solutions before real world displacement. In this context, the goal of this thesis is to develop and integrate an advanced control system for autonomous driving that leverages MATLAB and the CARLA simulator for co-simulation. The control system aims to implement a pipeline strategy for path tracking and trajectory planning based on Nonlinear Model Predictive Control (NMPC) that takes into account velocity and path curvature, as well as demonstrate that implementation for future decision making functionalities is possible.

NMPC compared to traditional controllers, such as PID or linear model-based controllers, offers numerous advantages in the context of autonomous driving,

especially for managing complex and dynamic scenarios. An NMPC can directly account for the inherent nonlinearity of the vehicle system and its dynamics, which are often not adequately represented by linear controllers like LQR or PID. Unlike traditional controllers, which typically respond to state errors without explicitly predicting future states, NMPC can anticipate system behavior and act proactively to optimize performance. Infact, NMPC incorporates predictive optimization using a model that considers real constraints, such as the vehicle's physical limits, obstacles, and safety boundaries, to plan the optimal trajectory over a defined time horizon. This feature is crucial in autonomous driving, where decisions must be based not only on the current situation but also on predictions of the upcoming moments to ensure safety and smooth driving. It is worth underlining the fact that, while the linear model predictive control MPC is more widely used in this field also because the resolution of optimization problem is convex, the nonlinear model predictive control presents non-convex solutions, for which ad hoc algorithms must be developed and optimized. This increase the computational difficulty and integration challenge in real-time systems but allowing better performances due to accounted nonlinear behaviors. Indeed facing with nonlinear system MPC performances drops due to the necessity to have an approximated linear model, often requiring linearization of the dynamics near a specific working point or along a defined trajectory that can results non trivial especially in tracking problems. Furthermore linear MPC requires constraints convexification that can be complicated, while NMPC uses simple inequalities to describe constraints. In summary, the choice of an NMPC over traditional controllers is justified by its ability to accurately model the vehicle's nonlinear dynamics, include safety constraints, and predict the effects of control actions, enhancing the responsiveness and overall effectiveness of the autonomous driving system.

The choice to validate the control system on CARLA stems from several compelling reasons that align with the goals of developing robust and reliable autonomous vehicle algorithms. CARLA is a cutting-edge open-source simulator designed specifically for autonomous driving research, providing a versatile platform that can accurately mimic real-world driving conditions. One of the primary advantages of using CARLA is its ability to simulate realistic urban environments, dynamic traffic scenarios, and a wide range of environmental conditions, which are crucial for testing the resilience and adaptability of control algorithms. Moreover, CARLA allows for the rapid iteration of experiments without the inherent risks and costs associated with real-world testing. This is particularly important when developing novel control strategies, as it enables researchers to evaluate the performance of their algorithms in a safe and controlled setting. The flexibility of CARLA also facilitates the exploration of various scenarios, including rare or hazardous situations that are difficult to replicate in physical environments, thereby enhancing

the comprehensiveness of the testing process. Finally, significant advantage of using CARLA is its capability for real-time integration with control algorithms. This feature allows for the seamless implementation and testing of control systems in real-time scenarios, enabling developers to observe the immediate effects of their algorithms on vehicle behavior. Real-time integration is essential for validating the responsiveness and effectiveness of control strategies, as it mirrors the time-critical decision-making required in actual driving situations. By utilizing CARLA for validation, we aim to ensure that the developed control systems can effectively manage the challenges of real-world driving, ultimately contributing to safer and more efficient autonomous vehicles.

However the most critical aspect of the work is developing the control system making possible the integration of the NMPC controller with CARLA. This requires designing an architecture that enables the NMPC to operate effectively within the simulated vehicle environment, ensuring that all necessary functions are implemented and optimized to maximize performance. To achieve this integration, the control system must receive real-time data from the simulator, such as position, speed, orientation, and environmental conditions. It must also generate control commands, such as steering, acceleration, and braking, that are compatible with CARLA's interface, while addressing computational and communication delays introduced by the simulator to ensure stability and performance. Key component of this process is the tuning of the predictive model used by the NMPC to align it with the physical behavior of the simulated vehicle in CARLA. Parameters such as grip coefficients, inertia, or command response dynamics must be calibrated to accurately reflect the simulated behavior.

The integration and optimization aim to enhance trajectory tracking, ensuring the vehicle closely follows the desired path, improve responsiveness to disturbances such as obstacles or trajectory changes, and increase computational efficiency to allow the NMPC to operate in real time, showcasing the benefits of integrating MATLAB with the CARLA simulation environment for testing. Success lies in designing functions that manage the interfaces and communication between the predictive model and the simulator while fine-tuning the model to maximize overall system performance.

1.7 State of Art

In recent years, the implementation of nonlinear model predictive control models in automotive applications has garnered increasing interest within the research community. NMPC has proven to be one of the most advanced and efficient control techniques for handling the complexity and nonlinearity inherent in autonomous driving systems, being widely used for path tracking and trajectory

generation, demonstrating robustness in maneuvers such as obstacle avoidance and overtaking. In [62] a trajectory planner for urban autonomous driving based on a Nonlinear Model Predictive Control is presented. The algorithm leverage tools like the ACADO toolkit and the qpOASES solver. Simulation results on CarMaker demonstrated the planner's effectiveness in complex scenarios, such as overtaking slower vehicles and emergency stops to avoid unexpected obstacles. The paper [63] details the development and testing of a real-time Nonlinear Model Predictive Control system for autonomous driving, implemented on a Ford Focus vehicle. This control strategy aims to handle complex driving tasks, including collision avoidance, lane keeping, and trajectory tracking in various environments. The NMPC was validated using a sequential testing framework, progressing from simulation (MIL) to hardware-in-the-loop (HIL), and finally to full vehicle-in-the-loop (VehIL) with physical testing. In [64] a framework to design, formulate and implement a path tracker for self-driving cars based on a nonlinear model predictive control approach was proposed. It allows the designer to easily integrate multiple objective terms in the cost function either opposing or correlating. The proposed design of the controller not only targets accurate tracking but also comfortable ride and fast travel time by introducing several sub-objective terms in the main cost function to satisfy these goals. These sub-objective terms are weighted according to their contribution to the optimization problem. [65] introduces a Nonlinear Model Predictive Control approach for real-time trajectory generation in highway driving for long truck. integrating road features (e.g., curvature) and traffic information over the prediction horizon. The objective function balances tracking performance, driver comfort, and maintaining safe distances from other road users. The Optimal Control Problem (OCP) is solved using ACADO code generation, and results are compared with a feedback scheme using the IPOPT interior-point solver. Many other studies have been published demonstrating the effectiveness and performance of NMPC for obstacle avoidance and evasive maneuvers [66, 67], as well for urban traffic implementation [68]. Further studies aimed at developing NMPC in different road conditions, as in [69] where a trajectory tracking for autonomous vehicles on varying road surfaces by friction-adaptive nonlinear model predictive control was presented. Others tried to minimizing the complexity, optimizing the solver or reducing computational time, e.g., [70, 71, 72, 73].

One of the most used simulators in the literature for validation of model predictive control approach on vehicles is certainly CarSim, which offers easy integration with MATLAB/Simulink. Many articles have been released about path tracking and yaw stability using NMPC [74] and MPC [75, 76, 77, 78, 79, 80], with validation within the CarSim environment. The paper [81] presents an enhanced trajectory tracking method for autonomous vehicles by combining model predictive control with Preview-Follower Theory (PFT). This approach aims to improve tracking accuracy and lateral stability by extending the effective reference path

without increasing computational load. The system was tested through simulations using MATLAB/Simulink and CarSim, showing superior performance in terms of response speed and stability compared to standalone MPC or PFT, especially in challenging maneuvers like lane changes and slalom. In [82] collision avoidance strategy was developed using steering and braking simultaneously with nonlinear model predictive control method. In this paper, constraints on the wheel steering angle is proposed in consideration of vehicle's predicted lateral acceleration, which should be smaller than the threshold in order to maintain lateral vehicle's stability. To verify the performance of the proposed strategy, two simulation scenarios were tested in MATLAB and CarSim simulation environments.

Regarding the implementation of NMPC in CARLA Simulator we have [83], which used Simulink as the controller design environment and CARLA as the simulation environment. Both PID and NMPC were tested in co-simulation in CARLA. These two methods are applied to both lateral and longitudinal control of the vehicle, using the single-track model for PID tuning and as internal model of the NMPC.

Widely is instead the literature about implementation of different type of linear model predictive control strategy in CARLA, especially focusing on path tracking [84] and realistic maneuvers. The paper [85] introduces an event-triggered MPC strategy for autonomous vehicle path tracking to reduce computational demands while maintaining performance. Unlike time-triggered MPC, this approach only recalculates the control action when specific conditions (events) are met, using past control sequences otherwise. The validation of this method was conducted in CARLA simulator, demonstrating that event-triggered MPC significantly reduces computation with minimal performance trade-offs compared to conventional time-triggered MPC. In [86] presents a Stochastic Model Predictive Control (SMPC) approach for autonomous driving at intersections, using Gaussian Mixture Models (GMM) to capture multi-modal predictions of surrounding vehicles for collision avoidance. The main contribution is an SMPC formulation that optimizes a new feedback policy, tailored to leverage GMM structure, allowing for a less conservative and convex programming-friendly solution. This feedback-based approach addresses the uncertainties in vehicle predictions, which is especially important in intersections. The method is evaluated in CARLA simulation using a kinematic bicycle model and it is shown to improve mobility, comfort, and computational efficiency compared to two baseline SMPC approaches that rely on open-loop strategies for multi-modal collision avoidance. The paper [87] addresses real-time obstacle avoidance for connected and automated vehicles (CAVs) in complex traffic, focusing on balancing efficiency and computational feasibility. The authors propose a two-layer MPC architecture that uses a differentially flat kinematic vehicle model. A fast, quadratic programming-based MPC handles immediate, local obstacle avoidance, while an asynchronous, more computationally intensive mixed-integer MPC provides globally

optimal updates. Both layers operate in parallel, incorporating position predictions of surrounding vehicles via V2X communication. High-fidelity co-simulations in CARLA demonstrate the approach's effectiveness for real-time, collision-free navigation in urban intersections and highway scenarios. The paper [88] presents a multi-constraint predictive control algorithm with a safety layer for reliable path tracking and emergency obstacle avoidance. A controller-switching mechanism alternates between a primary nonlinear MPC and an emergency controller. The MPC's performance is validated against Stanley and PID controllers for efficiency. To ensure safety in critical situations where MPC's computational load could cause delays, an emergency braking and maneuver system is included, capable of activating independently based on the situation. The approach is tested in two emergency scenarios using the CARLA simulator and Python, demonstrating its effectiveness in rapid response situations. In [89] a mixed autonomous driving control, focusing on lateral and longitudinal navigation using Model Predictive Control and Proportional-Integral-Derivative control respectively, is presented. The MPC handles lateral control by predicting the vehicle's trajectory and adjusting steering to stay on course, while the PID controller manages longitudinal control, adjusting throttle and brakes to maintain target speed and acceleration. Additionally, the Stanley control method is employed for path tracking, enabling precise adherence to a predefined route by accounting for road curvature and vehicle dynamics. Testing in the CARLA simulator demonstrates that combining MPC, PID, and Stanley control achieves accurate motion control and reliable, safe navigation for autonomous driving. The paper [90] presents a MPC algorithm for path-following and collision avoidance in autonomous vehicles, utilizing a time-varying, non-uniformly spaced prediction horizon. This approach uses shorter time intervals for near-future predictions and longer intervals for the distant future, allowing the algorithm to extend its prediction horizon while maintaining a fixed number of prediction steps. This design enhances obstacle detection by extending the vehicle's foresight but can reduce path-following accuracy on high-curvature routes. To address this, the algorithm dynamically adjusts prediction intervals: short intervals improve path-following on curved sections, while long intervals enhance obstacle detection over greater distances. This method improves both path accuracy and obstacle avoidance range without increasing computational load, validated through tests in the CARLA simulator and real-time experiments [90].

As seen in this section the linear MPC has been more studied and integrated in automotive simulators such as CarSim and CARLA also thanks to the convex optimization problem which allows a lower computational burden in real-time integration systems, different is instead the situation for NMPC that requires a non-convex optimization, increasing the integration challenge in real-time systems. This thesis aims to contribute to this rapidly evolving field by implementing and testing an NMPC controller within the CARLA simulator, in order to evaluate the

effectiveness of the model in a realistic context and to understand the challenges related to its implementation and management in real time.

1.8 Outline and Contributions

The main objectives of this thesis focus on the development and implementation of an advanced control system based on a Nonlinear Model Predictive Control approach for autonomous driving, showing the results after being integrated in CARLA simulator. Specifically, the following chapters of the thesis will be organized as follows:

Chapter 2: Vehicle Models and Control Systems A comprehensive exploration of various dynamics models pertinent to automotive systems will be presented, providing a solid foundation for understanding vehicle behavior. In particular for this thesis work, Detailed examination of the single track model, highlighting its advantages and limitations in comparison to other models. In-depth discussion of tire modeling, emphasizing its critical role in accurately predicting vehicle dynamics. Following a tough description of typical control system architectures employed in the automotive industry, showcasing the evolution and effectiveness of each approach. Critical analysis of the most commonly used control strategies, evaluating their strengths, weaknesses, and practical applications. Special focus on the advantages of Nonlinear Model Predictive Control algorithms as well as a Detailed formulation and theoretical underpinnings of the NMPC algorithm used in this thesis is presented, illustrating its potential to enhance vehicle control and stability.

Chapter 3: CARLA Co-simulation In-depth analysis of the architecture and key features of the CARLA simulator, emphasizing its importance as a tool for automotive research and development. Illustrating how it replicates real-world scenarios to provide accurate and reliable data for control system tuning. The integration with python will be presented, as well as python functions to gather data from autonomous CARLA driving and manual driving, which will be used for system tuning. Discussion on the integration of MATLAB within the CARLA simulation environment, highlighting the steps and methods used to achieve synchronized simulations. Presentation of the MATLAB environment developed for these synchronized simulations, showcasing its effectiveness in improving control system accuracy and performance. Section 3 develops certain preliminary elements that establish the methodological and theoretical foundation of the research, providing the necessary context to understand the following sections.

Chapter 4: Control System Design Control System Description: Detailed description of the developed control system, explaining its architecture, components, and operational principles. Presentation of the model used for predictive control, highlighting its parameters and details. Explanation of the tuning process for NMPC and the design of its cost function, focusing on optimizing performance and ensuring safety. Development and implementation of new system functions for localization and trajectory tracking, based on parameters like velocity and path curvature, enhancing the accuracy and efficiency of vehicle control. Detailed analysis of the system identification process on the CARLA vehicle to create an acceleration map based on gear, throttle, and brake input commands. This map is crucial for understanding vehicle dynamics and improving control strategies. Derivation of dispatching functions to determine the appropriate input signal commands for the vehicle, ensuring smooth and precise vehicle operation. Section 4 delves into the core contributions and represents the heart of the research, as this section details the most innovative and significant developments of the work carried out.

Chapter 5: Simulation Results Comprehensive analysis of simulations conducted in the CARLA environment, providing a realistic testing ground for the developed control system. Detailed comparison between manual driving, autonomous CARLA driving, and the newly implemented NMPC control system in path tracking, highlighting the strengths and weaknesses of each approach. Simulation of an obstacle avoidance scenario, showcasing the capabilities of the NMPC control system to react swiftly and safely to unexpected obstacles. Simulation of an overtaking maneuver, demonstrating the system's ability to handle everyday realistic driving tasks with precision and efficiency.

Chapter 2

Vehicle Models and Control Systems

2.1 Dynamical Vehicles Models

Vehicle dynamics is the study of the behavior of a vehicle in motion in response to external forces such as those generated by tires, aerodynamics, acceleration, braking and gravity. The goal of vehicle dynamics is to understand and model how a vehicle responds to driver commands and road conditions, to improve vehicle performance, safety and comfort. The system of coordinates that is used to describe the vehicle motion is shown in Figure 2.1. Each axis of the car frame is subjected to forces and moments. Longitudinal forces arise from the acceleration and braking of the vehicle. These forces are mainly generated by the tires through friction between the tires and the road surface. Lateral forces appear during turns. These are the forces that act laterally on the tires and determine the vehicle's ability to steer and maintain the desired trajectory. Vertical forces are associated with the weight of the vehicle and the distribution of forces on the suspension and tires. These influence the grip of the tires on the road. Roll, Pitch and Yaw Moments are respectively rotation of the vehicle around its longitudinal axis (from side to side), inclination of the vehicle around the transverse axis (from back to front), typical during braking or acceleration, and rotation around the vertical axis, linked to changes in direction of the vehicle.

To describe the dynamics of a vehicle, different models are used, each with different degrees of complexity depending on the applications, the most important based on engineering application and usage are: the Single Track Dynamical Model, or Bicycle Model, that is a 2D model of the vehicle with 3 degrees of freedom, representing the vehicle as two wheels (one front and one rear) where forces acting on wheels on same axle are lumped together. It is used to study lateral dynamics

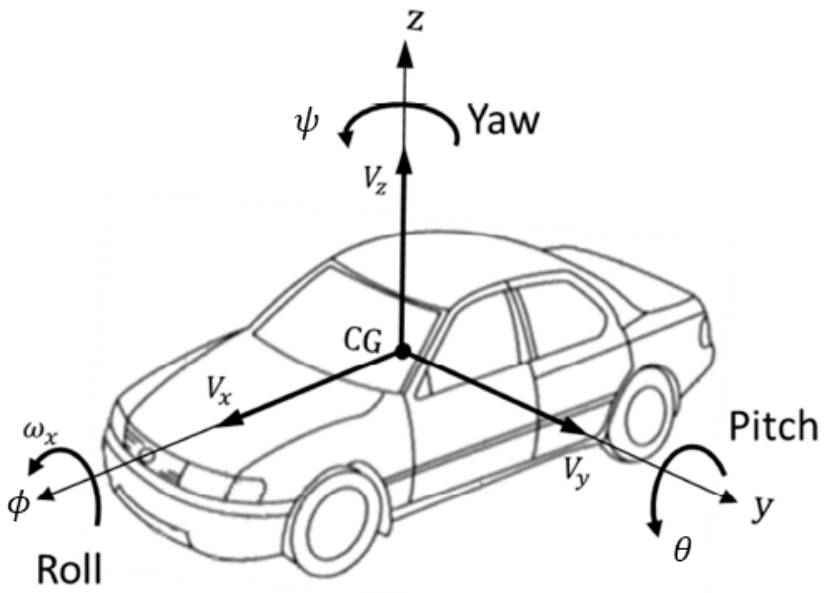


Figure 2.1: Vehicle coordinates system

in relation to steering, especially to understand how the vehicle reacts to changes in direction. It also describes the interaction between longitudinal and lateral forces and is useful for cornering stability analysis. Moreover the model can be extended to include roll and pitch effects. The Full Vehicle Dynamics Model, or Double Track Dynamical Model, instead consider the vehicle as a rigid body and take into account all six degrees of freedom: longitudinal, lateral, vertical movement, rotation around the axes (roll, pitch and yaw). They are used in advanced applications where an accurate description of the three-dimensional dynamics of the vehicle is necessary, such as for the analysis of suspension, roll, aerodynamics and overall dynamics of the vehicle. The longitudinal, lateral and vertical forces for each wheel are calculated separately, making the model much more realistic. It Allows to accurately model the behavior of the suspension, the load transfer between the wheels during cornering, braking or acceleration. Its applications are mainly the development of racing cars or high-performance vehicles where it's used near advanced simulations and crash analysis.

For performance driving, utilizing a double-track chassis model, which accounts for both lateral and longitudinal load transfer [91], can be beneficial. However, research indicates that a single-track model is adequately accurate for most of driving conditions [92], even when tire forces enter the nonlinear region, as the roll and pitch angles remain relatively small in those situations. Additionally, the single-track model is generally sufficient for most evasive maneuvers, where the primary concern is maintaining safety rather than achieving optimal performance, making

a highly precise model often unnecessary. Furthermore, the single-track model reduces computational demands, which is advantageous in automotive applications [93], especially during evasive maneuvers [69].

2.1.1 Dynamic Single Track Model

The half car dynamical model, often referred to as bicycle model, is used in applications where the vehicle's position, heading, and sideslip are of primary interest [94]. The model assumes that the vehicle moves on a two-dimensional plane, which means it does not account for roll, pitch, or vertical forces. It is primarily concerned with lateral, longitudinal, and yaw dynamics. The left and right track of the car are lumped into a single centered track assuming same forces on wheel on same axle as shown in Figure 2.2. Hence, only a single front and a single rear tire are considered, and roll and pitch dynamics are ignored, resulting in two translational and one rotational degrees of freedom [69]. The vehicle variables

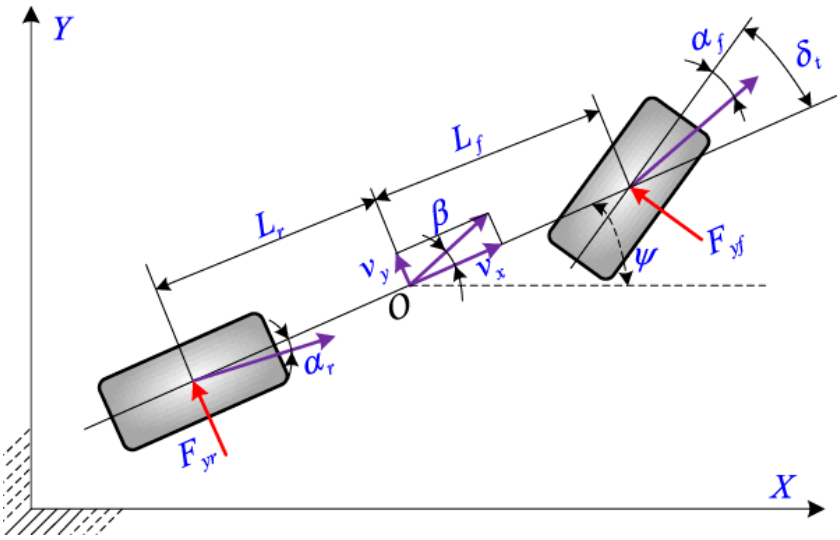


Figure 2.2: Single track bicycle model

and parameters are:

- δ_f : steering angle;
- β ; vehicle slip angle = angle between the vehicle longitudinal axis and velocity;
- α_f, α_r : tire slip angles = angles between the tire longitudinal axis and velocity;
- $O \equiv CoG$ center of gravity;
- ψ : yaw angle;

- L_f : distance CoG to front axle
- L_r : distance CoG to rear axle.

Taking the longitudinal and lateral velocities in the vehicle frame, v^X,v^Y, and the yaw rate, $\dot{\psi}$, as states, the single-track model is described by:

$$\dot{v}^X - v^Y\dot{\psi} = \frac{1}{m}(F_f^x\cos(\delta_f) + F_r^x - F_f^y\sin(\delta_f)) \tag{2.1a}$$

$$\dot{v}^Y + v^X\dot{\psi} = \frac{1}{m}(F_f^y\cos(\delta_f) + F_r^y - F_f^x\sin(\delta_f)) \tag{2.1b}$$

$$J\ddot{\psi} = L_f F_f^y\cos(\delta_f) - L_f F_r^y + L_f F_f^x\sin(\delta_f) \tag{2.1c}$$

Where F_i^x, F_i^y are the total longitudinal/lateral forces in the tire frame for the lumped left and right tires, and the subscripts $i = f, r$ indicate front and rear, respectively, m is the vehicle mass, J is the vehicle inertia about the vertical axis. The vehicle position in global coordinates $p = (p^X, p^Y)$ is obtained from the kinematic equation:

$$\begin{bmatrix}\dot{p}^X\\ \dot{p}^Y\end{bmatrix} = R(\psi)\begin{bmatrix}v^X\\ v^Y\end{bmatrix}$$

Where R is a rotation matrix dependent from the yaw angle ψ. The tire model describes how the tire forces F_i^x and F_i^y in (2.1) are generated [69].

2.1.2 Tire Models

In the field of autonomous driving, a range of tire models have been employed, spanning from simple to complex. The simplest wheel model, sufficient to describe lateral dynamics in urban scenarios, constrains the attainable force based on a geometric shape. Subsequently, certain tire models exclusively analyze lateral forces, assuming negligible tire longitudinal slip. Others tire models instead that accounts for slips in both the longitudinal and lateral directions, they're mostly used in racing application and complex scenarios.

Furthermore the relation between lateral force and the slip angle must be investigated. Where the relationship between the lateral force and the slip angle is nearly proportional, this is called *linear region*, as shown in Figure 2.3, the tire operates at small slip angles (or slip ratios). This is where the tire grip is predictable, and there is minimal sliding or saturation. As the slip angle increases beyond a certain point, the tire's behavior becomes non-linear. The lateral force no longer increases linearly with slip angle, and the tire starts to approach its peak lateral force. Once the tire exceeds its peak force, it enters the *non-linear region* where the force starts to level off or decrease.

Regarding the correlations between tire slip angle and forces, several models have been developed. These include the linear model, the simplified Pacejka model,

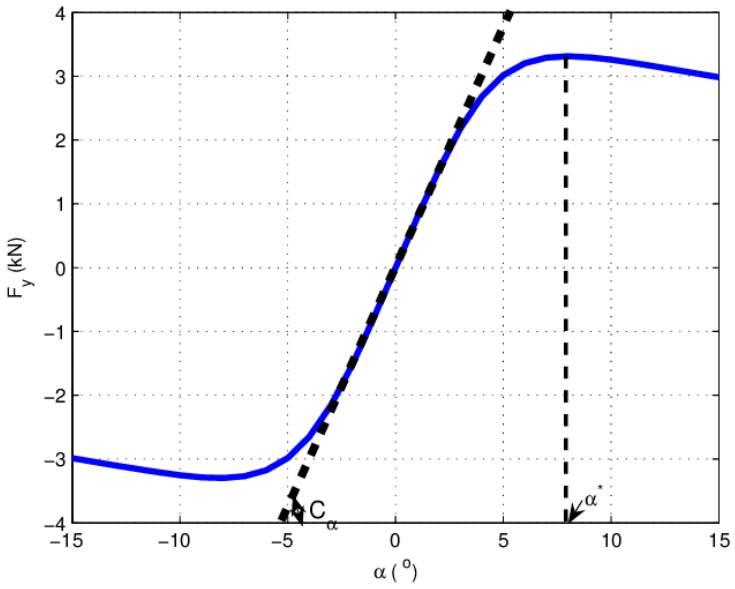


Figure 2.3: F_y lateral force vs α side slip angle

the Pacejka model, and the Fiala brush model, each with specific underlying assumptions. The linear and Pacejka models will be presented next.

Since the longitudinal slip angle is not taken into account under the assumption of small slip angles, the vehicle's longitudinal behavior is primarily determined by longitudinal models without considering tire saturation. Additionally, the camber angle of a wheel is not considered. This simplification allows for the straightforward derivation of tire slip angles. In the case of a front steering vehicle, based on a single-track model, the lateral slip angles α of its front and rear tires can be expressed as [95]:

$$\alpha_f = \arctan\left(\frac{v^Y + l_f\omega}{v^X}\right) - \delta_f$$
$$\alpha_r = \arctan\left(\frac{v^Y - l_r\omega}{v^X}\right)$$

with $w = \dot{\psi}.$, and resulting in:

$$F_f^y = -C_{\alpha,f}\alpha_f \cos(\delta_f) \quad F_r^y = -C_{\alpha,r}\alpha_r$$

where $C_{\alpha,i}$ represents the constant cornering stiffness of the respective single tire, front and rear.

Lateral forces F_i^y exerted by tires within specific lateral slip angles exhibit a nearly linear relationship with the corresponding α_i. This allows for the use of

a linear model to approximate tire behavior in this range, resulting in what is commonly referred to as a linear tire model, expressed as [96]:

$$\begin{aligned}\alpha_f &= (\frac{v^Y + l_f\omega}{v^X}) - \delta_f \\ \alpha_r &= (\frac{v^Y - l_r\omega}{v^X}) \\ F_i^y &= -C_{\alpha,i}\alpha_i \quad i = f, r\end{aligned}$$

where $C_{\alpha,i}$ represents the constant cornering stiffness of the respective single tire, front and rear.

F^y linearly increases with increasing α, corresponding to the effective range of the linear model. As the absolute value of α further increases, F^y gradually approaches saturation and remains approximately constant. The Pacejka model, widely more used with the complete car dynamical model, often referred to as the Magic Formula, is designed to fit experimental tire response curves [97] and is based on the following equation [98]:

$$\mu_{y.P} = B_1 \sin(B_2 \arctan(B_3\alpha - B_4(B_3\alpha - \arctan(B_3\alpha))))$$

where B_1 denotes the peak value, B_2 is a shape factor, B_3 represents the stiffness factor, and B_4 stands for a curvature factor. Although variations may exist in different literature sources, such as accounting for aerodynamic effects as an additional coefficient function [99], these variations are mainly focused on the derivation of F^z, the vertical force, which is crucial for lateral tire force. However, the Pacejka model is employed to estimate μ, which is subsequently used to compute lateral force F^y by multiplying it with F^z. In this context, a constant B_1 is adequate for adjusting the maximum value of μ [100].

At low slip angles α, where the lateral force F^y increases almost linearly with the slip angle, and the tire is within its elastic deformation limits. In this region, the stiffness factor B_3 primarily controls the slope of the curve. Mathematically, in the small slip angle region the Pacejka formula simplifies [101] to:

$$F_y \approx k * \alpha, \quad k = B_1B_2B_3$$

2.2 Control System Architecture for ADS

An autonomous driving controller is a system that manages and regulates the behavior of an autonomous vehicle, making real-time decisions based on data from sensors and maps. In control systems the term *functional architecture* can be used according the notion of *functional concept* expressed in the ISO26262 automotive functional safety standard. Where functional concept is defined as ”specification

of the intended functions and their interactions necessary to achieve the desired behavior". A functional architecture then refers to logical decomposition of the system into components and subcomponents, as well as the data-flows between them. However it doesn't prejudice the technical implementation of the architectural elements in terms of hardware and software. A similar term, recommended by ISO 42010 [102] for the architectural functional description of software intensive systems, is *functional view* of the architecture description. Since autonomous systems are highly software intensive, joining these two terms together we would refer to *functional architecture view* (FAV) to the functional view of the system software architecture [103]. Referring to pipeline control model of an AV Functional architecture view of control software consists typically of three layers: perception layer, decision/planning layer, and vehicle platform manipulation/trajectory control layer as shown in Figure 2.4.

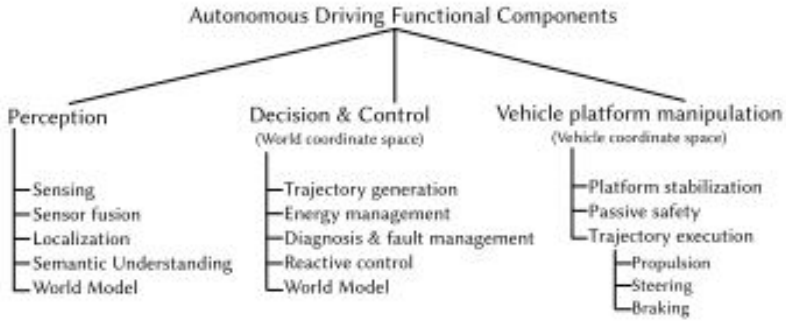


Figure 2.4: AV Software FAV

In the perception layer, the sensing components can be categorized into those sensing the states of the ego vehicle, like IMU, wheel encoders, brake pressure, throttle position and steering angle sensor, and those sensing the states of the environment, sensors like camera, radar, LiDAR, and laser etc.. Combining real-time information/data by using filtering, estimation, fusion, association and classification, activities such as localization, neighboring vehicle detection, static obstacle detection, object classification, lane detection, road detection and perception map are executed in the perception layer.

The decision and control layer refers to those functional components which are concerned by the vehicle characteristics and behavior in the context of the external environment it is operating in. In this layer energy and fault management play a crucial role in vehicle's motion and reactive control to unexpected events. However one of the core functions of the whole software architecture of AV is motion planning. There are three main features of the planning layer: route planning, behavior planning, and path planning and trajectory planning. A root can be

defined as a trip from initial position to the final destination through the road network. The route planner provides a long term planning and generates the optimal route according to traffic and distance from start point to destination point. The physical space should be transformed into a configuration space in search space planner to be the representation of environment. Another important module of the planning layer is the driving behavior planner, which is essentially a decision-making module to provide reasonable obstacle avoidance as well as safe driving actions and other types of maneuvers. Path planner and trajectory planner generates a safe, comfort and feasible trace to follow. The path is a sequence of configuration vectors (way-points) in a collision free space of independent attributes like position, orientation, linear velocity, angular velocity, acceleration, and steering angle etc. A trajectory can be defined as a sequence of spatio-temporal states in the free space (time varying way-points) which are feasible for vehicle dynamics, give the answer about how to move along the collision free path with considering mechanical limitation and kinematic constraints of vehicle [104].

Finally, in Vehicle platform manipulation layer the trajectory execution components are responsible for actually executing the trajectory generated by decision and control layer. Generation of appropriate control is achieved by a combination of longitudinal acceleration (propulsion), lateral acceleration (steering) and deceleration (braking).

A feedback control architecture is a foundational approach in control systems used to ensure that a system behaves as desired by continuously adjusting its inputs based on the difference between the system's actual performance and the desired performance (referred to as error), Figure 2.5. It is particularly important in autonomous vehicles and other dynamic systems, where real-time adjustments are critical for maintaining stability, accuracy, and safety.

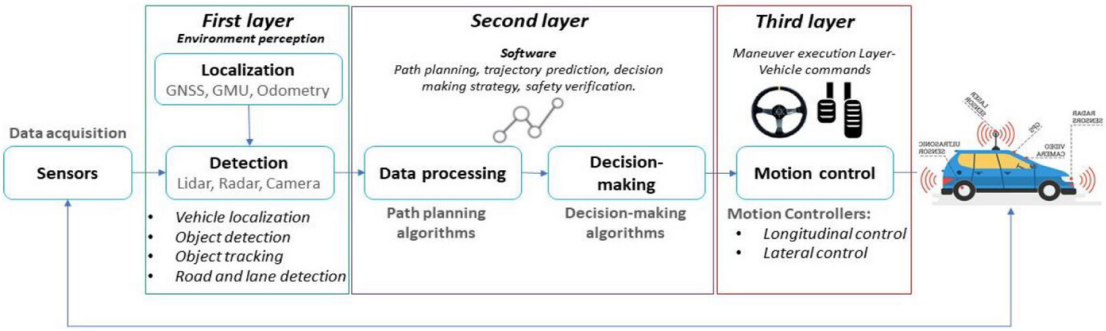


Figure 2.5: AV architecture whit control feedback

2.3 Trajectory Planning and Control Algorithms

In control systems, a feedback strategy is a crucial mechanism where the system continuously monitors its output and compares it to the desired reference or set point. If there is any discrepancy (referred to as an error), the system takes corrective action to minimize that error, adjusting its inputs accordingly. Feedback control is essential for achieving stability, accuracy, and robustness. In feedback controllers (also called closed-loop controllers) an input or reference r is compared with a measured output variable y. On the basis of the resulting error $r - y$ a suitable value is generating for the manipulated variable u, that is a plant input signal, as in Figure 2.6. Controllers can be categorized [105] by their operating

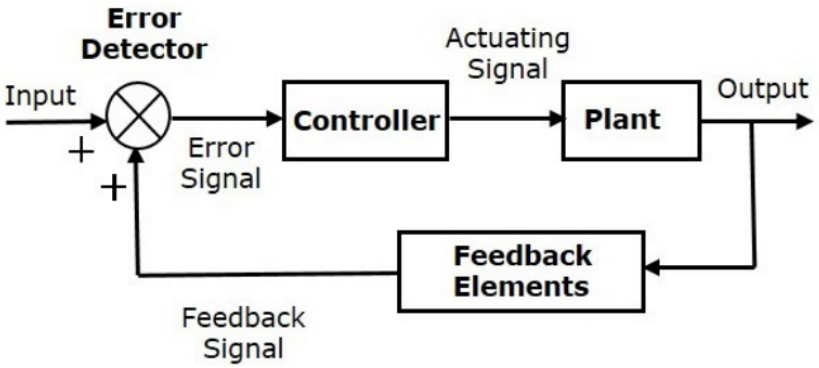


Figure 2.6: Feedback loop design

principles into classical, predictive, and repetitive types. Classical controllers—such as PID, bang-bang, and state controllers—respond reactively, adjusting only based on past and current system behavior. Predictive controllers, in contrast, utilize a model of the system to forecast future behavior and adjust in anticipation deviations w.r.t. the reference [106]. Repetitive controllers leverage the behavior from a previous cycle to compute an optimized trajectory for the following cycle, enhancing accuracy over repeated actions [107].

Two of the most widely used in control history of classical controllers are PID controllers and sliding mode controller (SMC). PID controller has simple design and theory and is a common controller usually used in industrial and vehicles applications. It consists of three terms and typically triggered by the error between actual feedback response and a desired reference. The three terms are Proportional, P, Integral, I, and Derivative, D, which corresponds, respectively, to the action that each term applied to the error signal, Figure 2.7. It is usually used in trajectory tracking to control the steering input as well as velocity input depending on a desired steering and velocity that trajectory planner imposes. While on one side PID controllers are particularly easier to implement due to its simplicity, on the

other hand tuning of its parameters can be challenging. In case of fast varying system such as vehicles, one set of parameters may perform well only for certain range of operating condition and any operating condition outside of this range will require further tuning of the controller. To address this limitation, adaptive PID controllers have been developed. These controllers adjust the PID parameters automatically in response to changing system conditions, improving robustness in fast-changing environments [108, 109].

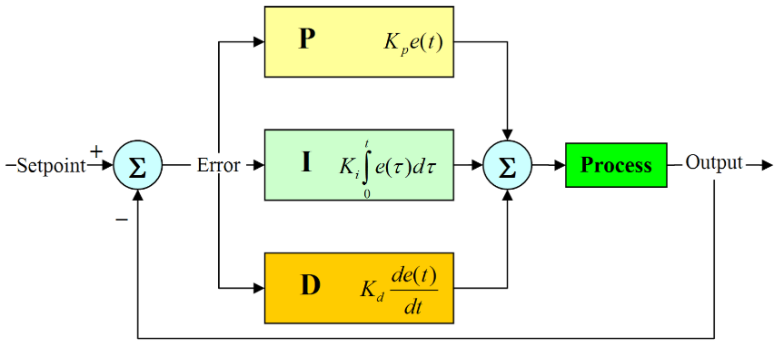


Figure 2.7: PID controller architecture

The sliding mode controller is another type of classical controller. In SMC, both state feedback and control signals are treated as discontinuous functions, making the controller unaffected to parametric uncertainties and external disturbances [110]. This robustness is a key reason why SMC is well-suited for nonlinear systems. The term "Sliding Mode" refers to the system's motion as it moves along predefined boundaries, called sliding surfaces, within the control structure. The control law employs a fast switching strategy to guide and maintain the system's state trajectory on these sliding surfaces [111] as depicted in figure 2.8. This characteristic gives SMC its strengths: it acts as a nonlinear controller with a rapid response and strong robustness under system uncertainties and disturbances. However, the fast switching mechanism used by SMC leads to an effect called chattering, which refers to high-frequency oscillations in the control signal. Chattering in control signal is inevitable and in real-world systems can cause issues such as actuator wear, plant damage, energy loss, and unintended disturbances, especially in systems with delays or imperfections in physical actuators [112]. Despite these challenges, SMC has been successfully implemented in various studies on path-tracking control for mobile robots and vehicle platforms [113, 114, 115].

The Linear Quadratic Regulator (LQR) is one of the most popular predictive controller, it uses optimal control theory where the controller gain was determined

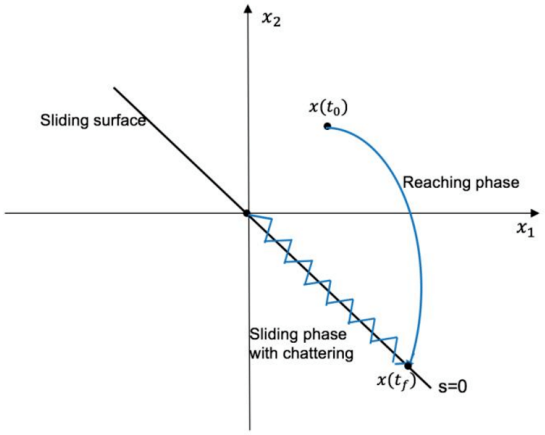


Figure 2.8: Sliding mode controller with chattering

using linear quadratic optimization approach. LQR is an optimal control technique used to determine the control inputs that drive a dynamic system to operate efficiently while minimizing a certain cost function. It is widely applied in control systems for its ability to manage stability, performance, and energy consumption in linear systems, making it particularly useful in robotics, aerospace, and autonomous vehicles. While it requires careful tuning of the weighting matrices and works best for linear systems, it remains a key tool in control engineering for efficient and stable system operation. LQR assumes a linearized model of vehicle dynamics, which may not always be accurate for highly nonlinear or time-varying systems. The goal of LQR is to minimize a cost function that reflects the trade-off between minimizing the error and the energy used for control. The LQR control law determines the optimal control input that minimizes the cost function. By adjusting the Q and R matrices, called weight, in the cost function the designer can prioritize either the minimization of state errors (making the system more responsive) or the minimization of control effort (making the system less aggressive and more energy-efficient). A higher Q will result in more precise control, while a higher R will reduce control energy but potentially allow more error in the system. Optimal controllers may provide simpler control law structure of $U = -kx$ compared to other controllers, where U is the control signal, k is the controller gains and x is the vehicle's states. Determination of k is done offline [116, 9] in the controller development stage solving the so called Riccati equation, which give the simple structure. The LQR controller guarantees stability if the system is controllable (i.e., if the system's states can be influenced by the input), as it minimizes a quadratic cost and drives the system to the desired equilibrium point [117]. However, this may not be the case for controllers that employ online tunings.

The effectiveness of any feedback design is inherently constrained by the system's

dynamics and the accuracy of its model. Therefore, even in theory, perfect tracking of time-varying reference trajectories cannot be achieved using feedback control alone, regardless of the design approach [118]. In specific situations, like the technical limitations of actuators, tailored solutions are required. These solutions are often heuristic, making them difficult to understand and maintain. Advanced control methods, such as sliding mode or back-stepping controllers, also tend to be abstract and complex in their interpretation [119]. The founders of MPC theory [120, 121] emphasized that classic control methods are effective for approximately 90% of control problems, with advanced control only needed for the remaining cases. However, MPC is a valuable approach for almost any control challenge—even those previously left unaddressed due to limited theoretical understanding or concerns about feasibility. MPC operates by continuously performing real-time optimization using a mathematical model of the system [106]. This model enables MPC to predict the future behavior of the system, incorporating these predictions into an optimization process that determines the optimal trajectory for the control variable. MPC offers an intuitive approach to parametrization by fine-tuning a process model, albeit with a higher computational demand compared to traditional controllers. Its ability to anticipate future states and incorporate hard constraints makes it especially valuable for real-world applications. With advances in computational power and the increasing availability of complex process models across various fields, MPC now facilitates the control of systems that were once considered impractical. Because MPC relies on models, which exist in nearly every discipline, it leverages established knowledge without requiring the explicit formulation of a control law—a task traditionally reserved for control specialists. Instead, MPC automatically derives the control law through model-based optimization. This implicit approach, coupled with flexibility and model integration, are key advantages of MPC and underscore its potential in engineering applications [105].

Furthermore comparing MPC to LQR, they are both expressions of optimal control, with different schemes of setting up optimization costs. The main differences between MPC and LQR are that LQR optimizes across the entire time window (horizon) whereas MPC optimizes in a receding time window [122], and that with MPC a new solution is computed often whereas LQR uses the same single (optimal) solution for the whole time horizon. Therefore, MPC typically solves the optimization problem in a smaller time window than the whole horizon and hence may obtain a suboptimal solution. However, because MPC makes no assumptions about linearity, it can handle hard constraints as well as migration of a nonlinear system away from its linearized operating point, both of which are major drawbacks to LQR. This means that LQR can become weak when operating away from stable fixed points. MPC can chart a path between these fixed points, but convergence of a solution is not guaranteed, especially if thought as to the convexity and complexity of the problem space has been neglected.

However, MPC has best performance if the plant model is linear or the plant nonlinear model can be approximate to linear model in a good way, for example industrial systems whose models are defined as 'almost linear', thus allowing good controller performance. In nonlinear systems, for example in robotics, automotive, aerospace or biomechanics, it is difficult to approximate the complex nonlinear system to a linear system. this is where nonlinear model predictive control comes into play, capable of predicting and optimizing non-linear systems ensuring better performance, but which requires higher computational resources to solve non-linear optimization problems, which are often non-convex.

2.3.1 NMPC

Nonlinear Model Predictive Control is emerging as a vital technology due to its ability to develop control algorithms for multivariable nonlinear systems while handling constraints on states, inputs, and outputs (see, e.g., [123, 124, 125, 126]). As in linear MPC, NMPC requires the iterative solution of optimal control problems on a finite prediction horizon. While these problems are convex in linear MPC, where a quadratric optimization is solved [127], in nonlinear MPC they are not necessarily convex anymore. This poses challenges for both NMPC stability theory and numerical solution[128, 129]. This is the reason why to address these complexities related to nonlinear dynamics, constraints, and non-convex performance criteria, Nonlinear MPC (NMPC) was developed[130]. Both MPC and NMPC require fast and dependable optimization algorithms to meet the stringent timing requirements of real-time closed-loop control. This is particularly challenging in NMPC, where solving nonconvex optimization problems online requires advanced algorithms with significantly higher computational demands compared to linear MPC [131]. Infact, the nonlinear nature of the problem may necessitate a substantial number of computations at each sampling moment due to the potential for multiple local minimum solutions, without ensuring the attainment of the best possible optimal solution. Consequently, NMPC requires the iterative resolution of an optimal control problem at each sampling instant in a receding horizon manner. Regrettably, there's no assurance that this receding horizon strategy of implementing a series of open-loop optimal control solutions will perform effectively or remain stable when applied to the closed-loop system. On the other hand when dealing with nonlinear systems, the performance of MPC often degrades because it relies on an approximate linear model. This typically requires linearizing the system dynamics either around a specific operating point or along a predefined trajectory, which can be particularly challenging in tracking scenarios. Moreover, linear MPC demands the convexification of constraints, a process that can be complex. In contrast, NMPC allows constraints to be represented directly as simple inequalities, simplifying their formulation and implementation. Over

the past few decades, considerable advancements have been made to reduce the computational burden associated with the NMPC approach. Nevertheless, in recent years, advancements in nonlinear optimization algorithms have led to the development of efficient NMPC implementations suitable for real-time applications.

Mathematical formulation

Model predictive control approach is also known as receding horizon control or moving horizon control. They make use of an explicit dynamic plant model to predict the effect of future reactions of the manipulated variables on the output and the control signal obtained by minimizing the cost function [132], loop example in Figure 2.9. The performance of the controller is highly influenced by how well the dynamics of the system is described by the input–output model that is used for the design of the controller [133]. Nonlinear Model Predictive Control typically incorporates three core step:

1. An explicit model is used to predict the process output over a defined future time horizon, called prediction horizon.

2. A control sequence is calculated to optimize a specific performance index.

3. A receding horizon approach is applied, shifting the time horizon forward at each step and implementing only the first control action from the calculated sequence at each interval.

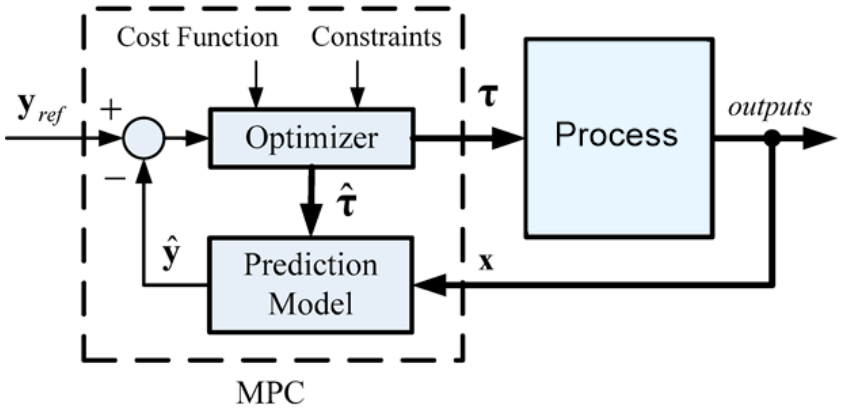


Figure 2.9: NMPC control loop

This prediction and optimization process is repeated at each time step. The core idea is that achieving optimality over a short prediction horizon leads to overall optimality over the long term [134], as the error in near-term predictions is typically smaller than in long-term predictions.

Assuming an arbitrary Multiple-Input-Multiple-Output(MIMO) nonlinear dynamic system described by the following state equations:

$$\begin{aligned} \dot{x} &= f(x,u) \\ y &= h(x,u) \end{aligned} \tag{2.2}$$

Where $x \in \mathbb{R}^{n_x}$ x is the state, $u \in \mathbb{R}^{n_u}$ is the command input and $y \in \mathbb{R}^{n_y}$ is the output; Respectively, $f\colon \mathbb{R}^{n_x}\mathrm{x}\mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$ and $h\colon \mathbb{R}^{n_x}\mathrm{x}\mathbb{R}^{n_u} \to \mathbb{R}^{n_y}$ are two functions characterizing the system dynamics and output variables. Assume that the state is measured in real-time, with a sampling time T_s, according to:

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

If the state is not measured, an observer or (2.2) in input-output form has to be employed. NMPC is based on two key operations: prediction and optimization. At each time $t = t_k$, the system state and output are predicted over the time interval $[t, t+T_p]$, where T_p is called the *prediction horizon*. The prediction horizon is the number of future time steps over which the NMPC predicts the system's behavior based on the model, in particular by integration of (2.2). For any $\tau \in [t, t+T_p]$, the predicted output $\hat{y}(\tau)$ is a function of the "initial state" $x(t)$ and the input signal:

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

where $u(t:\tau)$) denotes the input signal in the interval $[t,\tau]$. Within the time frame $[t, t+T_p]$, $u(\tau)$ acts as an open-loop input, meaning it does not rely on $x(\tau)$. The *control horizon* instead is the number of time steps within the prediction

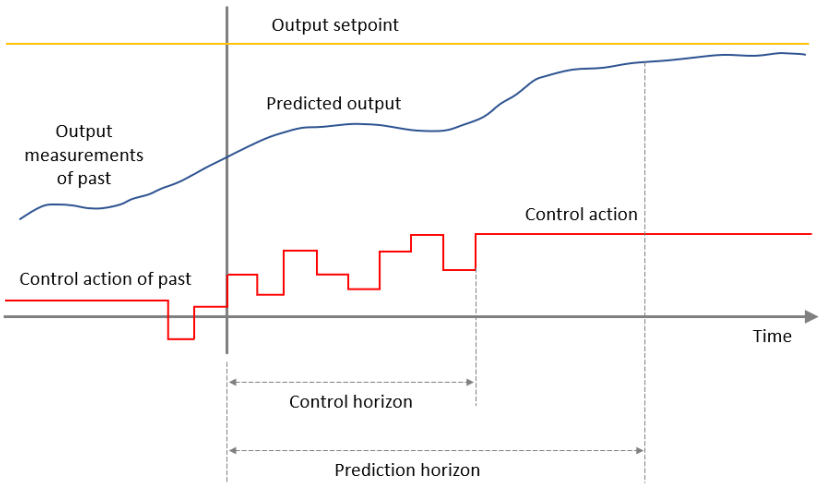


Figure 2.10: NMPC controller strategy

horizon during which the control inputs are optimized. After this horizon, the control inputs are typically held constant (or follow a predefined strategy), in our work the control horizon will be considered equal to the prediction horizon and the so called "moving blocking" strategy, explained after, will be applied. The full typical NMPC control strategy is depicted in Figure 2.10.

The basic idea of NMPC (and of the most predictive approaches) is to look for an input signal

$$u^*(t\colon \tau)$$

at each time $t = t_k$, such that the prediction

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

has a desired behavior in the time interval $[t, t + T_p]$. The concept of desired behavior is formalized by defining the *objective function*:

$$J(u(t\colon t + T_p)) \doteq \int_t^{t+T_p} (||(e_p(\tau))||_Q^2 + ||(u(\tau))||_R^2)d\tau + ||(e_p(t + T_p))||_P^2$$

where $e_p(\tau) \doteq r(\tau) - \hat{y}(\tau)$ is the predicted tracking error, $r(\tau) \in \mathbb{Y} \subset \mathbb{R}^{n_y}$ is a reference to track, $\mathbb{Y}$ is a bounded set. $||\cdot||_*$ is a weighted Euclidean norm. For example, letting Q be a positive definite weight matrix, the norm of a column vector w is defined as:

$$||w||_Q^2 = w^T Q w = \sum_{i=1}^{n} q_i w_i^2, \quad Q = diag(q_1, \ldots, q_n) \in \mathbb{R}^{n \times n}$$

The objective is to minimize, at each time t_k , the square norm of the tracking error $||e_p(\tau)||_Q^2 = ||r(\tau) - \hat{y}(\tau)||_Q^2$ over a finite time period. The term $||(e_p(t + T_p))||_P^2$ emphasizes the significance of the final tracking error. The term $||(u(\tau))||_R^2$ enables the management of the trade-off between performance and command activity.

The input signal $u^*(t\colon t + T_p)$ is chosen as one minimizing the objective function $J(u(t\colon t + T_p))$. In particular at each time $t = t_k$, for $\tau \in [t, t + T_p]$, the following nonlinear Optimal Control Problem is solved:

$$u^*(t\colon t + T_p) = \arg\min_{u(\cdot)} J(u(t\colon t + T_p)) \tag{2.3}$$

subject to:

$$\begin{aligned} &\dot{\hat{x}}(\tau) = f(\hat{x}(\tau), u(\tau)), \quad \hat{x}(\tau) = x(\tau) \\ &\hat{y}(\tau) = h(\hat{x}(\tau), u(\tau)) \\ &\hat{x}(\tau) \in X_c, \quad \hat{y}(\tau) \in Y_c, \quad u(\tau) \in U_c. \end{aligned}$$

The first two constraints in this problem ensure that the predicted state and output are consistent with the system equation (2.2). The sets X_c and Y_c account for other constraints that may hold for the predicted state/output (e.g., obstacles, barriers). The set U_c accounts for input constraints (e.g., input saturation). The optimal problem (2.3) is generally non-convex. Moreover, the decision variable $u(\cdot)$ is a signal, and optimizing a function with respect to a signal is generally a difficult task. To overcome this problem, the prediction interval $[t_k, t_k + T_p]$ can be divided into n_s sub intervals $[t_k + \tau_i, t_k + \tau_{i+1}] \subset [t_k, t_k + T_p], i \in \{1,2,\ldots,n_s\}$, where the τ_i's are called nodes, and u and r can be kept constant on each sub-interval [72]. Hence, u_{ki} and r_{ki} denote the command and the reference values at time k in the ith sub-interval, respectively. The command and reference sequences in the prediction interval are indicated with $u_k \doteq (u_{k1}, \ldots, u_{kn_s})$ and $r_k \doteq (r_{k1}, \ldots, r_{kn_s})$, respectively. In this way, the optimal problem reduces to a finite-dimensional problem, which can be solved using an efficient numerical optimization algorithm. The NMPC closed-loop command is obtained according to a so-called *receding horizon strategy*. At time $t = t_k$ the input signal $u^*(t\colon t + T_p)$ is computed by solving (2.3). Then, only the first optimal input value $u(\tau) = u^*(t_k)$ is applied to the plant, keeping it constant for $\forall \tau \in [t_k, t_{k+1}]$. The complete procedure is repeated at the next time steps $t = t_{k+1}, t_{k+2}, \ldots$.

When examining the choosing of the parameters

The choice of control parameters is essential in the design of predictive control, as these parameters significantly influence the system's performance, stability, and computational efficiency.

- T_s: In many cases, the sampling time is fixed and cannot be adjusted. However, when adjustable, a trial-and-error approach during simulation can be employed. The sampling time should be sufficiently small to capture the system's dynamics effectively, in accordance with the Nyquist-Shannon sampling theorem. At the same time, it should not be excessively small, as this could lead to numerical instabilities and increased computational overhead.

- T_p: The prediction horizon can also be determined through a simulation-based trial-and-error process. A larger T_p generally improves the closed-loop system's stability. However, an excessively large T_p may reduce the accuracy of short-term tracking and increase the computational burden.

- Choosing Q, R, P values can be similar to the parameters in LQR/LQRY.

When examining the solution of the OCP

In the context of Nonlinear Model Predictive Control, to determine the control inputs typically minimizing (or maximizing) an objective function, optimization plays a crucial role since the problem to solve is often nonlinear, constrained, and non-convex. Optimization algorithms for NMPC need to balance accuracy, speed,

and robustness. The most used solution methods for solving the Optimal Control Problem in Nonlinear Model Predictive Control are direct methods, due to their robustness, flexibility, and ability to handle complex dynamics and constraints. In direct methods, the OCP is discretized into a finite-dimensional optimization problem, typically a Nonlinear Programming (NLP) problem. These methods focus directly on numerical optimization. In indirect methods instead, the OCP is solved by deriving and solving the necessary optimality conditions, typically through the calculus of variations or Pontryagin's Maximum Principle [135]. Indirect methods are rarely used in NMPC due to their sensitivity and lack of robustness for large-scale, nonlinear, or constrained problems.

Most used direct methods in NMPC OCP solving are [134, 72]: Sequential Quadratic Programming (SQP) and Interior-Point Method. SQP solves nonlinear optimization problems by transforming them into a sequence of Quadratic Programming (QP) problems, each easier to solve. It is highly efficient for nonlinear problems with constraints and performs well near the optimal solution. However, it may require good initialization and can have high computational costs for large problems. In NMPC, it is frequently used for real-time applications, especially when the model is highly nonlinear but well-conditioned. Another method is the Interior-Point Method that solves constrained nonlinear problems by transforming them into unconstrained ones using logarithmic barrier functions. These methods are effective for problems with many constraints and have faster convergence than other gradient-based methods. However, they require careful parameter tuning and are often applied in NMPC problems with numerous inequality constraints.

Instead for OCP formulation, the most used direct methods in NMPC are the Direct Single Shooting, Direct Multiple Shooting and Direct Collocation. Direct Single Shooting that is a method for solving Optimal Control Problems by parameterizing the control inputs over the prediction horizon. Starting from an initial state, the system dynamics are integrated forward in time using these controls, generating the state trajectory implicitly. The optimization problem minimizes a cost function, such as tracking error or energy use, while respecting constraints on the control variables. This approach is computationally efficient because it only optimizes the control inputs, resulting in smaller nonlinear programming problems. However, it is sensitive to the initial guess for the controls and handles state constraints less effectively since they are indirectly influenced through the controls. It is best suited for systems with simple dynamics and fewer constraints on the states. Direct Multiple Shooting instead focuses on the optimization of both inputs and state, it divides the prediction horizon into smaller intervals and optimizes them separately while ensuring continuity through constraints. This method is robust compared to global methods and efficient for complex nonlinear dynamic models, although it may require many iterations to converge. It is widely applied in industrial and robotic control. Another strategy used is Direct Collocation ,

it approximates the system dynamics using discrete collocation points and solves the problem as a large nonlinear optimization problem. It is suitable for systems with slow dynamics and reduces computational complexity compared to multiple shooting. However, it is sensitive to the choice of collocation points and is applied in NMPC for systems with slow dynamics or many state variables.

The choice of algorithm depends on the model complexity, the available computational resources, and the required solution quality. In this thesis work the NMPC control algorithm made use of a direct single shooting approach for OCP formulation and SQP method for solving it.

Chapter 3

CARLA Co-Simulation

3.1 CARLA Features and Architecture

Developing and testing autonomous driving technologies demands a reliable simulation environment that can realistically represent the real world, including vehicles, pedestrians, and diverse environmental conditions. This environment should also offer extensive sensor simulation capabilities and allow seamless integration with analytical tools like MATLAB. CARLA has been chose among other simulators possibilities to develop and test Autonomous driving algorithms in different scenarios and conditions. CARLA provides high-quality graphics with Unreal Engine 4 and its physics engine accurately models vehicle dynamics and environmental interaction, simulating a wide range of sensors used in autonomous vehicles. Finally, one of the most important features for us, CARLA's Python API enables smooth integration with MATLAB, allowing for an efficient workflow in data analysis and algorithm testing.

The CARLA simulator consists of a scalable client-server architecture, as represented in Figure 3.1:

Server The server manages all simulation processes, such as sensor rendering, physics calculations, world-state updates, actor management, and other related tasks. To ensure high realism, it ideally operates on a dedicated GPU, which is especially advantageous for machine learning applications.

Client The client consists of several modules responsible for controlling actor behavior and configuring environmental conditions. This is facilitated by the CARLA API (available in Python and C++), which serves as an interface between the server and client, continually evolving to incorporate new features.

The client module is the user interface for requesting information or making adjustments within the simulation. Each client connects through a designated IP

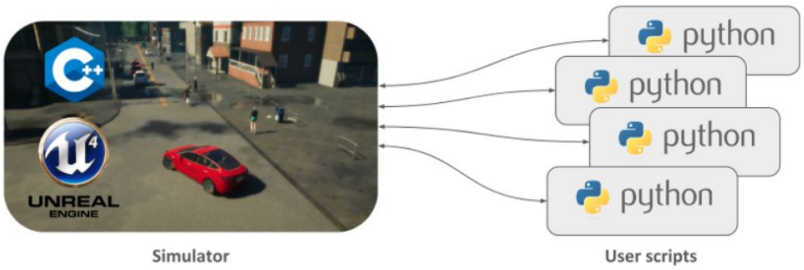


Figure 3.1: CARLA simulator server and client representation

address and port, interacting with the server via the terminal. Multiple clients can operate concurrently, but efficient multi-client management requires in-depth knowledge of CARLA and proper synchronization. The client object offers a range of functions, including loading maps, recording simulations, and initializing the traffic manager.

The world object represents the simulation environment, example in Figure 3.2, acting as an abstraction layer with essential methods for spawning actors, adjusting weather, retrieving the current world state, and more. Only one world instance exists per simulation, and it resets whenever the map changes. Through this world object, users can access various elements of the simulation—such as weather conditions, vehicles, traffic lights, buildings, and the map—via its comprehensive set of methods.

Figure 3.2: CARLA simulation with cars and pedestrians

In CARLA, actors are entities that perform actions within the simulation and interact with other elements. These include vehicles, pedestrians, sensors, traffic

signs, traffic lights, and the spectator. Effective management of actors—such as spawning, destruction, types, and control—is essential for simulation use. CARLA's flexible framework enables users to easily add new actors, each of which is a pre-designed model with animations and various attributes. Certain attributes, like vehicle color, the number of channels in a LiDAR sensor, or a pedestrian's speed, are customizable to suit specific simulation needs. The world object handles the spawning and tracking of actors throughout the simulation. The CARLA actor class offers "get()" and "set()" methods to manage actors on the map. Importantly, actors do not automatically delete themselves when a Python script terminates; they must be explicitly destroyed to free resources.

3.2 CARLA Traffic Manager

The Traffic Manager (TM) is a module that controls vehicles in autopilot mode to simulate realistic urban traffic within the simulation environment. Users can adjust various behaviors, including defining specific learning scenarios. TM operates on the client side of CARLA and follows a structured, stage-based design. Each stage has dedicated operations and objectives, enabling focused development of phase-specific features and efficient data processing. These stages run on separate threads, with synchronous messaging ensuring a one-way flow of information between stages to optimize computational performance.

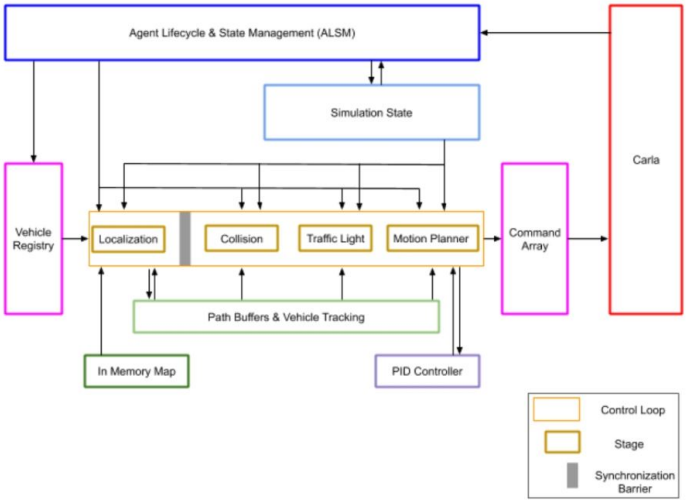


Figure 3.3: CARLA traffic manager architecture

The diagram above, in Figure 3.3, illustrates the internal architecture of the Traffic Manager. The Agent Lifecycle & State Management (ALSM) system continuously monitors the world to track all vehicles and pedestrians, removing

any entries for those that are no longer present. It collects data from the server and processes it through multiple stages, functioning as the sole component that directly communicates with the server. The vehicle registry maintains an array of autopilot vehicles (controlled by the Traffic Manager) along with a list of non-autopilot vehicles and pedestrians. The simulation state acts as a cache, storing the position, velocity, and other relevant details of all vehicles and pedestrians within the simulation.

Based on the current simulation state, the Traffic Manager generates the appropriate commands for each vehicle in the vehicle registry. The calculations for each vehicle are carried out individually and are divided into separate stages. A control loop maintains consistency across all calculations by enforcing synchronization barriers between stages, ensuring that no vehicle progresses to the next stage until the calculations for all vehicles in the current stage are finished. Each vehicle moves through the following stages:

- Localization Stage: Paths are dynamically generated by selecting a sequence of nearby waypoints from the In-Memory Map, which simplifies the simulation map into a grid of waypoints. At junctions, the vehicle's direction is chosen randomly. Each vehicle's path is stored and managed by the Path Buffers & Vehicle Tracking (PBVT) component, enabling easy access and modifications in the subsequent stages.

- Collision Stage: Bounding boxes are projected along each vehicle's path to identify and manage potential collision hazards.

- Traffic Light Stage: Similar to the Collision Stage, potential hazards affecting each vehicle's path—such as traffic lights, stop signs, and junction priorities—are identified and managed.

- Motion Planner Stage: Vehicle movement is calculated along the designated path, with a PID controller that navigate it toward the target waypoints. The PID controller determines the required throttle, brake, and steering inputs to reach the target values based on data from the Motion Planner Stage. Adjustments are made according to the controller's configuration, with modifiable parameters as needed. These inputs are then converted into a CARLA command for execution.

The commands generated in the previous step are assembled into a command array and sent to the CARLA server as a batch, ensuring they are executed simultaneously within the same frame.

Figure 3.4: Anaconda interface to run scripts made with Spyder and Python on CARLA environment

3.3 Anaconda Interface

To control and interact with Carla the Python API has been used. Thus the python version must be compatible with the Carla version installed. In order to keep under control the synchronization of Carla and Python through new possible release, it's been preferred to work in a virtual environment. To do this the program Anaconda Navigator was used to manage virtual environments, creating the one needed, install and update libraries for CARLA and python both, keeping all the desired packages under the same folder. It also provides quick access to system commands through prompt window, activating virtual environment and running scripts. Nevertheless it has a Python scripting environment program called Spyder in which scripts can be opened and edited before launching them from the Anaconda prompt, flow interface in Figure 3.4. The following steps were used to set up Carla-Anaconda communication:

- Download Carla 0.9.14;
- Download Anaconda Navigator;
- Open anaconda's prompt windows;
- Create A virtual environment as follows:
  `create -name 'name_of_environment' python=3.7`;
- Activate this environment:
  `activate 'name_of_environment'`;
- The necessary Python modules were downloaded as follows:
  `pip install Carla, pygame, numpy, jupyter, opencv-python`.

After completing these steps, the connection between Python and CARLA is successfully established. Now it's possible to navigate through change directory 'cd' command in the folder where script are stored and run them. Some tutorial examples are already present inside CARLA folder, we can run these to simulate traffic, manually control a car using the keyboard, or drive a vehicle with a steering wheel, along with many other types of simulations.

3.4 Data Gathering Autonomous Mode

As described above through python API is possible during the simulation to access to actors' parameters, such position, velocity and many else. In this project what we are interesting in is collect data from our ego car driven in autonomous mode by the CARLA traffic manager. These data will be then imported in MATLAB and used to analyses the car behavior (vehicle dynamic) under certain input, generate the reference data for the NMPC controller that will be built and understand the trajectory errors done by autopilot PID based controller in CARLA. In order to run the ego vehicle in autopilot mode is sufficient to run the script "manual control" that spawn a vehicle in the map for manual drive through keyboard and then use "P" button to enable the autopilot mode. The vehicle will start to drive in autonomous mode (it can also be set directly from a script via CARLA world method "settings" and assign autopilot variable as true).

All the data needed are collected with a specific implemented function function *"get_states"* at each game iteration in the Pygame game loop. With an if statement the function is called only if more than 50 ms are passed from previous call in order to maintain a sample time as possible near to 50 ms. Each data is saved in a Python list and at the end of the program all the data are written in in three different text files. In the first there are the dynamical states of the ego vehicle, this means x position, y position, yaw angle, x velocity, y velocity and yaw rate. In the second there are input parameters as throttle, brake and steering data, as well as the gear and the accelerations in x and y directions. Last in the third file are saved the x ad y position of the nearest waypoint (point in the center of the lane that are used as reference by the autopilot) in CARLA at the moment of the function call. In both first and second file the time instant at which each data is collected has been written in the text files. Additionally the spawn point were set in the script from time to time on the base of the desired scenario.

Following the implemented code in Python script where get() functions such as get_acceleration(), get_velocity(), get_transform(), and get_angular_velocity() will be used. The get_transform() method includes both location of the object (X, Y, Z in the map frame) in meters, and its rotation angles from which we took the yaw values. It's important to say that since a simplified 2D vehicle model will be

used the z coordinate, the pitch and roll angles will not be considered.

Listing 3.1: Python data gathering function

```
def get_state(self):
    global time_data
    time_data = pygame.time.get_ticks()
    time.append(time_data)

    location = self.player.get_location()
    velocity = self.player.get_velocity()
    transform = self.player.get_transform()
    yaw = transform.rotation.yaw* math.pi/180
    yaw_rate = self.player.get_angular_velocity().z* math.pi/180
    acceleration = self.player.get_acceleration()

    location_data_x.append(location.x)
    location_data_y.append(location.y)
    location_data_z.append(location.z)

    velocity_data.append(velocity)
    velocity_data_x.append(velocity.x)
    velocity_data_y.append(velocity.y)

    yaw_data.append(yaw)
    angular_velocity.append(yaw_rate)

    acceleration_data.append(acceleration)
    acceleration_data_x.append(acceleration.x)
    acceleration_data_y.append(acceleration.y)

    c = self.player.get_control()
    throttle.append(c.throttle)
    brake.append(c.brake)
    gear.append(c.gear)
    steering_data.append(c.steer)

    map = self.world.get_map()
    waypoint01 = map.get_waypoint(self.player.get_location(),
project_to_road=True, lane_type=(carla.LaneType.Driving | carla.
LaneType.Sidewalk))
    waypoint_list_x.append(waypoint01.transform.location.x)
    waypoint_list_y.append(waypoint01.transform.location.y)

    return acceleration_data, acceleration_data_x,
acceleration_data_y, steering_data, location_data_x,
location_data_y, yaw_data, velocity_data, velocity_data_x,
velocity_data_y, angular_velocity, waypoint_list_x,
waypoint_list_y
```

3.5 Manual Control Data Gathering

Another method used for data collection in CARLA was the manual driving mode. Using the steering wheel and pedals, Logitech G29 Driving Force setup, available in the department office as well as a Sparco sport seat it was possible to collect data and simulate driving scenario in which the human being was driving instead of the autopilot. This will be used to compare human errors with CARLA autopilot and the implemented control system.

Figure 3.5: Steering wheel and pedals setup for manual driving

The script we have to run to drive whit steering wheel and pedals is "manual_control_steeringwheel" in which the same function *get_state* as described before was implemented to collect all the data needed. Furthermore, since the design camera position in the manual driving was a 3rd person view of the car, it was necessary to change the camera position. The final set position of the camera, that provide a first person view as it can be seen in Figure 3.5, was the following:

Listing 3.2: CARLA camera view settings

```
class CameraManager(object):
    def __init__(self, parent_actor, hud, gamma_correction):
        ...
        bound_x = 0.5 + self._parent.bounding_box.extent.x
        bound_y = 0.5 + self._parent.bounding_box.extent.y
        bound_z = 0.5 + self._parent.bounding_box.extent.z
        Attachment = carla.AttachmentType
```

```
8
9           if not self._parent.type_id.startswith("walker.pedestrian"):
10              self._camera_transforms = [
11                  (carla.Transform(carla.Location(x=0.0*bound_x, y
    =-0.25*bound_y, z=1.1*bound_z), carla.Rotation(pitch=-18.0)),
    Attachment.Rigid)]
12          ...
```

This camera position lead to a more realistic driving view and to a better precision and control of the car while manual driving.

3.6 MATLAB Interface

Figure 3.6: MATLAB and Python interface with CARLA environment

Communication between Carla and Matlab is not trivial since there is no direct interface method. Two options are presented in CARLA documentation to let CARLA and MATLAB work together. The first, which setup requires numerous compatibilities and it operates more optimally on the Linux operating system, is to use ROS bridge to connect MATLAB and CARLA. However, while this option is more advantageous for processing large amounts of data, this advantage is not necessary for our project, which also relies on computers with a Windows operating system. The second option is the one that has been used: connection through Python bridge, represented in Figure 3.6. To work correctly the Matlab and Python versions have to be compatible. After have downloaded MATLAB version 2021b, which is compatible with Python version 3.7 the steps required to establish the CARLA-MATLAB connection are presented below:

To make the `easy_install` feature operational via Python, the following steps are taken in anaconda prompts in the virtual environment previously created:

- `pip install setuptools==33.1.1`;
- `Add C: \Python27 \Scripts` to your 'path' (Environment variable) - `C: \Python34 \Scripts`;
- `easy_install pip`;
- `easy_install carla-0.9.14-py3.7-win-amd64.egg`.

Then the following steps are then taken in MATLAB:

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

The last two command must be run into the MATLAB command line at the start of every MATLAB session to have access to CARLA Python API and commands. Additionally is preferable to run MATLAB application *'as administrator'*. In this way, the necessary connection is established and, making use of a virtual port, the MATLAB-CARLA communication is made ready.

3.7 CARLA Enviroment in Simulink

This subsection provides the details of integration of the CARLA simulation environment with MATLAB/Simulink, where the simulation will be run. The integration is possible using MATLAB *System block*, where with Python API it's possible to connect to Carla world and interact with the simulation, spawning and controlling CARLA actors. System block provides setup function *setupImpl* to connect with the client, change world settings and spawn actors, and a loop function *stepImpl* where until end of simulation at each time step is possible through Python API to use command such *Getters* and *Setters* to receive information about vehicle states and apply command to the ego vehicle. In the beginning part of the system we declare the external input variables as steering angle, throttle and brake. Additionally local variables are declared such the car itself as the ego vehicle and the vehicle state text file is imported to be used in the initialization function to acquire the and set the spawn position. After the loop function for each variables or set of data exiting the system block the size of these are declared, also if they're complex number, the type of the variable, single or double, and if the output signal is sampled in time. Finally the commands to be applied when the simulation finishes, in our case to destroy the ego vehicle.

One of the most important feature to pay attention at in the implementation is the synchronization between Simulink, whose engine can discrete time to have the correct works of all the elements at each time step, and CARLA server where the simulation runs. To have synchronization the CARLA world settings must be changed in the System block setup function through Python world method "settings" imposing the synchronization variable to true and the delta step size to 0.05, lower or equal than the sampling time that will be chose in the Simulink simulation. Also in the system loop the function *world.tick()* must be run to let the server simulation proceed between one step and the next of Simulink simulation. This setup is fundamental to run the co-simulation between Simulink where the controller will be designed and CARLA, being able to have the correct synchronization during the simulation.

Listing 3.3: MATLAB environment

```
classdef Carla_env_studio_sync < matlab.System
    % Carla Enviroment

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

    properties(DiscreteState)

    end

    % Pre-computed constants
    properties(Access = private)
        car;
        spawnSet = importdata("sim_out.txt");
        world;
    end

    methods(Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing
    constants

            port = int16(2000);
            coder.extrinsic('py.carla.Client');
            client = py.carla.Client('localhost', port);
            client.set_timeout(20.0);
            obj.world = client.get_world();

            settings = obj.world.get_settings()
```

```
33              settings.synchronous_mode = true;
34              settings.fixed_delta_seconds =  0.05;
35              obj.world.apply_settings(settings)
36
37              % Spawn Vehicle
38              blueprint_library = obj.world.get_blueprint_library();
39              car_list = py.list(blueprint_library.filter("leon"));
40              car_bp = car_list{1};
41              spawn_point = py.random.choice(obj.world.get_map().
    get_spawn_points());
42
43              spawnLine = obj.spawnSet.data(1,:);
44              %Convert yaw angle in degrees for vehicle spawning
45              if spawnLine(6)<0
46                  spawnLine(6) = (2*pi+spawnLine(6))*180/pi;
47              else
48                  spawnLine(6) = (spawnLine(6))*180/pi;
49              end
50              spawn_point.location.x = spa2(2);
51              spawn_point.location.y = spa2(3);
52              spawn_point.location.z = 0.6; %default setting
53              spawn_point.rotation.yaw = spawnLine(6);
54
55              obj.car = obj.world.spawn_actor(car_bp, spawn_point); %
    spawn the car
56              obj.car.set_autopilot(false)
57          end
58
59          function [ze, x_acceleration, y_acceleration, gear] =
    stepImpl(obj,steeringangle_input,throttle_input,brake_input)
60              % Perform loop calculations
61
62              obj.world.tick();
63
64              vehicle_transform = obj.car.get_transform();
65              location = vehicle_transform.location;
66              velocity = obj.car.get_velocity();
67              acceleration = obj.car.get_acceleration();
68              orientation = vehicle_transform.rotation;
69
70              x_position = location.x;
71              y_position = location.y;
72              yaw_angle = double(deg2rad(orientation.yaw));
73              x_velocity = velocity.x;
74              y_velocity = velocity.y;
75              w = obj.car.get_angular_velocity().z*pi/180;
76
77              x_acceleration = acceleration.x;
78              y_acceleration = acceleration.y;
```

```
79
80              control = obj.car.get_control();
81              gear = double(control.gear);
82              control.steer = rad2deg(steeringangle_input)/70;
83              control.throttle = throttle_input;
84              control.brake = brake_input;
85              obj.car.apply_control(control);
86
87              ze = [x_position, y_position, yaw_angle, x_velocity,
    y_velocity, w]';
88          end
89
90          function [ze, xacc, yacc, gear] = isOutputComplexImpl(~)
91              ze = false;
92              xacc = false;
93              yacc = false;
94              gear = false;
95          end
96
97          function [ze, xacc, yacc, gear] = getOutputSizeImpl(~)
98              ze = [6,1];
99              xacc = [1,1];
100             yacc = [1,1];
101             gear = [1,1];
102         end
103
104         function [ze, xacc, yacc, gear] = getOutputDataTypeImpl(~)
105             ze = 'double';
106             xacc = 'double';
107             yacc = 'double';
108             gear = 'double';
109         end
110
111         function [ze, xacc, yacc, gear] = isOutputFixedSizeImpl(~)
112             ze = true;
113             xacc = true;
114             yacc = true;
115             gear = true;
116         end
117
118         function resetImpl(~)
119             % Initialize / reset discrete-state properties
120         end
121     end
122
123     methods(Access= public)
124         function delete(obj)
125             % Delete the car from the Carla world
126             if ~isempty(obj.car)
```

```
127                     obj.car.destroy();
128                 end
129             end
130         end
131 end
```

Chapter 4

Control System Design

4.1 Project specifications

The purpose of the project is to create a control system based on NMPC controller for urban automated driving that is able to track a desired path and implement some complementary functions aimed to decision making integration. The system will be built in Simulink leveraging on the possibility to connect and interact with the CARLA environment by means of the MATLAB System block. In order to compare the NMPC system and the PID based CARLA autopilot performances the behaviors must be as similar as possible initially, so control system is designed to impose constant speed of 30 km/h as CARLA autopilot does in the absence of curves. The controller will also be able, using a function that calculates the curvature of the path, to slow down near curves and accelerate again once they are over. This ensures improved trajectory tracking on curves, as well as enhanced stability and safety in the vehicle's behavior. The reference to track will be set as the same that in the defined scenario CARLA autopilot follows in order to compare the errors and behaviors. Then the control system will be tested also in working conditions different from the initial specifications to address it's robustness and adaptivity.

Its useful to describe, for understanding of this thesis developments, that the NMPC controller function is a specific component within the control system, in our case made by a Simulink loops around at a MATLAB function, that performs the optimization task. It takes the vehicle's current states and reference points as inputs, solves the optimization problem by minimizing the cost function, and outputs the optimal control actions, such as acceleration and steering.

In contrast, the entire control system encompasses a broader set of components and functionalities. It includes not only the control function but also additional modules such as state estimation (e.g., filtering sensor data), reference generation

(e.g., defining the trajectory to be tracked), actuator interfacing (e.g., converting control commands into physical actions), and possibly a decision-making layer. The control system ensures seamless integration and communication between these components, enabling the vehicle to operate autonomously and respond dynamically to changes in the environment.

4.2 NMPC controller

One of the crucial part of this project was the choice of NMPC controller parameters. Selecting parameters is challenging, as it requires balancing control accuracy, stability, and execution speed. Poorly chosen parameters can lead to instability or ineffective system performance. A deep understanding of the system dynamics and the specific application requirements is essential. Additionally an experimental tuning phase was requested to identify the optimal parameters based on the application's specific demands. Below the parameters that best optimized the controller's operation and results will be presented.

The NMPC controller takes as input the vehicle's current states and 50 reference points for each model output to be tracked. Using these inputs, the controller performs an optimization via moving block formulation over the prediction horizon of the cost function at each step to compute the optimal model inputs required to follow the reference, as described in Chapter 2. In this case, the model inputs are acceleration and steering, which are adjusted to minimize the tracking error and achieve the desired behavior while the model output to be tracked are x and y position.

To chose the sampling time we investigates the typical sampling time in autonomous driving vehicles and control systems, these varies depending on the specific system being controlled and the application requirements. In High-level decision-making, path planning, and route navigation usually have a longer sampling time, 100–500 ms, since they don't require rapid updates. In Motion Planning and Control (Lateral and Longitudinal) layer, handling trajectory tracking and control, including steering and speed control, lower sampling times, 20–100 ms, are often preferred to maintain a smoother response and ensure safety at higher speeds. In Perception and Sensor Processing object detection, lane detection, and sensor fusion generally operate at a relatively high frequency, 10–50 ms sampling time, to capture changes in the environment and maintain situational awareness. At Low-Level Vehicle Control level, controllers that handle braking, throttle, and steering actuators operate with very low sampling times, typical 5–20 ms, to ensure fast response, critical for maintaining stability and safety in real-time. To have a good trade off between all the typical sample time in automated vehicles components, considering that we are most interested in decision making and control planning,

we chose 50 ms for the controller sampling time.

Next same procedure was followed to chose the prediction horizon, which in ADS usually varies depending on the application and specific requirements of the driving scenario. Generally, it balances between providing a sufficient foresight for safe, smooth path tracking and keeping computational demands manageable. In urban areas (complex environment) with frequent stops, pedestrian crossings, and many obstacles, a shorter prediction horizon, 2-5 seconds, is often preferred to keep computational demands feasible and quickly adapt to changes in the environment. This horizon provides enough foresight for lane-keeping, obstacle avoidance, and dealing with frequent traffic signals. In highway driving, the vehicle typically moves at higher speeds with fewer obstacles and more gradual lane changes. A longer prediction horizon, 5–10 seconds, allows the controller to plan smoother, more stable trajectories, essential for comfort and fuel efficiency. For precise maneuvers like parking or navigating through tight spaces, a short prediction horizon, 1–3 seconds, is often used due to the low speeds and high accuracy required. This short horizon allows fine-grained control without overloading the computational system. Taking into account the urban scenario in which our ego vehicle controller operates and the presence of sudden maneuvers in narrow spaces, we have chosen 3 s as the prediction horizon.

The optimization law in our case is based on the following cost function:

$$J(u(t\colon t+T_p)) \doteq \int_t^{t+T_p} (||(e_p(\tau))||_Q^2 + ||(u(\tau))||_R^2 + ||\frac{d}{dt}u(\tau)||_{Rsd}^2)d\tau + ||(e_p(t+T_p))||_P^2$$

where $e_p(\tau) \doteq r(\tau) - \hat{y}(\tau)$ is the predicted tracking error, $r(\tau) \in \mathbb{Y} \subset \mathbb{R}^{n_y}$ is a reference to track, $\mathbb{Y}$ is a bounded set.

The objective is to minimize, at each time t_k, the square norm of the tracking error $||e_p(\tau)||_Q^2 = ||r(\tau) - \hat{y}(\tau)||_Q^2$ over a finite time period. When Q is increased the tracking over all the reference points improves. The term $||(e_p(t+T_p))||_P^2$ emphasizes the significance of the final tracking error. When P is increased the tracking over the final the reference point improves. The term $||(u(\tau))||_R^2$ enables the management of the trade-off between performance and command activity. When R is increased the command effort and energy consumption of the input signals decreases. The term $||\frac{d}{dt}u(\tau)||_{Rsd}^2$ limits the difference between the input commands at the each time and those at the next instant. When Rsd is increased it limits the difference between the input signals at current instant and the one at previous instant resulting in smother input commands. Furthermore, in the cost function it is possible to insert a further term relating to any constraints on the states of the model using a dedicated function if necessary. At the moment this is not our case but if necessary it will be discussed in subsequent chapters.

Additional parameters to be included in the NMPC controller tuning are the *ncp* parameter, this indicates the position of the nodes, explained in Chapter 2,

within the prediction horizon. The position is expressed in percentage values of the prediction horizon and indicates the ranges in which the controlled input command can change and stay constant. In our case the parameter is defined as par.ncp $= [0, 10, 30, 100]$, this means that three valid input commands will be generated in the three percentage ranges defined by the parameter to best track the reference in the prediction horizon, while only the first at the initial time instant will be given to the system. These values were chosen through careful tuning as the best choice based on the resulting performances obtained in test simulations, given the speed of the vehicle and the working scenario defined for the controller application. It was seen that with 3 nodes, thus two constants commands during the prediction horizon, no matter of the values chosen the performances of the system were non satisfactory. Instead with 5 nodes the computational bargain of the controller function was too high for the system. If the vehicle were to work in a different scenario it may be necessary to redefine the parameter to optimize the precision of the controller but it could be done without difficulty.

Another important parameter to define are the upper and lower saturation limits of the input commands, in our case longitudinal acceleration and steering angle. For longitudinal acceleration the maximum and minimum values were obtained in the process of system identification (explained in next sections) analyzing data collected from manual driving. However, for the steering angle, by accessing the physics of the vehicle via Python API, it was possible to obtain the maximum and minimum steering angles of each wheel, equal to ± 70 degrees, but considered ± 50 degrees to stay away from dynamics boundaries and since in our case the range that will be used in our scenario will be less than these values. Steering boundaries values are then transformed into radians values to match dynamics measurements units.

The final parameters values found, firstly chosen as described before, then tested via trial and error, that give the best trade off in terms of accuracy and computational bargain, are:

Parameter	Value
T_s	0.05
T_p	3
n	6
R	diag([1; 1])
Q	diag([1000; 1000])
P	diag([1; 1])
Rsd	diag([50; 500])
ncp	[0, 10, 30, 100]
ub	[3.5, 0.8727]
lb	[-8, -0.8727]

Where n are the states of the vehicle model, *diag* is a diagonal matrix with diagonal entries defined in the argument vector, *ub* are the upper bounds for acceleration and steering angle, *lb* are the lower bounds for the same inputs, respectively.

4.3 Vehicle Model

4.3.1 Car Parameters

In order to describe the vehicle dynamics in the correct way and increase the level of detail of the vehicle model we must have some considerations on the chosen car and on the parameters that we will need to describe its dynamics. One of the most important parameter to know is the position of the center of gravity of the car, in our case a Seat Leon. The position of the car returned by Python API is the center of the vehicle bounding box in CARLA space, while the dynamic of the vehicle is built around the center of mass position. Using the following API command, *get_physics_control()*, is possible to get and change car parameters and wheels parameters.

Listing 4.1: CARLA get physics parameters

```
player_physics=self.player.get_physics_control()
print(f'{player_physics}')
```

From this we get the position of the center of mass with respect to the center of the bounding box, that is translated of 0.2 meters towards the front of the vehicle in x direction. Other important parameters retrieved from this command are tires lateral stiffness, $C_f, C_r = 15000N/rad$, and mass of the vehicle that is $m = 1318kg$. From wheels location, also retrieved by physics command, it was possible to determine the front to rear axle distance and furthermore, by knowing the CoG position, the CoG to front axle distance and the CoG to rear axle distances was approximated, under the hypothesis that the bounding box center of the car is in the middle of the front-rear distance. Resulting respectively $L_f = 1.16m$ and $L_r = 1.56m$. J, the vehicle inertia about the vertical axis was estimated from the mass and the dimensions of the car taken from bounding box specific command resulting about $J = 2500kgm^2$.

4.3.2 Differential Equations

To control the ego-vehicle in the simulation, it is essential for the controller to understand the vehicle's dynamics. Among various modeling approaches, the Dynamic Single-Track (DST) mode strikes a balance between simplicity and accuracy for our controller design context, which means urban driving and low speeds, making it our choice for simulating vehicle dynamics and control. This model effectively

represents a four-wheeled vehicle by simplifying it into a two-wheel model—one wheel at the front and one at the rear—under the assumption that the left and right sides of the vehicle exhibit symmetrical behavior. Furthermore, being simplified, the use of the single track model has a notable computational advantage for real-time control when the controller must predict and optimize the input commands.

Under the assumption of small steering angles, the fact that longitudinal tire slip won't be used because we'll work with a^X longitudinal acceleration. Based on other theory formulation seen in formulas 2.1 in Chapter 2 the final formulation of the nonlinear vehicle control model is:

$$\begin{aligned}
\dot{p}^X &= v^X \cos(\psi) - v^Y \sin(\psi) \\
\dot{p}^Y &= v^X \sin(\psi) + v^Y \cos(\psi) \\
\dot{\psi} &= \omega \\
\dot{v}^X &= v^Y \omega + a^X \\
\dot{v}^Y &= -v^X \omega + \frac{2}{m}(F_f^y + F_r^y) \\
\dot{\omega} &= \frac{2}{J}(L_f F_f^y - L_r F_r^y)
\end{aligned}$$

Where 2 factor was added to compensate the effect of both two wheels for each axis and F_f^y and F_r^y are the lateral forces exchanged between the wheels and the vehicle:

$$F_f^y = -C_f \alpha_f \cos(\delta_f) \qquad F_r^y = -C_r \alpha_r$$

The tire slip angles α_f and α_r are defined as:

$$\begin{aligned}
\alpha_f &= \arctan\left(\frac{v^Y + l_f \omega}{v^X}\right) - \delta_f \\
\alpha_r &= \arctan\left(\frac{v^Y - l_r \omega}{v^X}\right)
\end{aligned}$$

The longitudinal acceleration a^X and the steering angle δ_f are the control variables. The output of the system is (p^X, p^Y). Following the MATLAB function implementation of the single track model that the NMPC controller will use to predict and optimize the command inputs .

Listing 4.2: MATLAB single track model function

```
function [xdot,y] = dyn_single_track_3dof(t,x,u)

% Parameters
Lf = 1.16;      %m
Lr = 1.56;      %m
m = 1318;       % kg
```

```
J = 2500;        % kg*m^2
Cf = 1.5e4;      % N/rad
Cr = 1.5e4;      % N/rad

% State variables
X = x(1,:);
Y = x(2,:);
psi = x(3,:);
vx = x(4,:);
vy = x(5,:);
w = x(6,:);

% Input variables
ax = u(1,:);
delta = u(2,:);

% Slip angles
betaf = atan2(vy+Lf*w,vx)-delta;
betar = atan2(vy-Lr*w,vx);

% Lateral forces
Fyf = -Cf.*betaf.*cos(delta);
Fyr = -Cr*betar;

% State equations
X_dot = vx.*cos(psi)-vy.*sin(psi);
Y_dot = vx.*sin(psi)+vy.*cos(psi);
psi_dot = w;
vx_dot = vy.*w+ax;
vy_dot = -vx.*w+2/m*(Fyf+Fyr);
w_dot = 2/J*(Lf*Fyf-Lr*Fyr);

% State derivative
xdot = [X_dot;Y_dot;psi_dot;vx_dot;vy_dot;w_dot];

% Output
y = x([1  2],:);
```

4.4 MATLAB Controller Implementation

Before being able to run the simulation some steps are required in MATLAB to define all the functions, constants, controller parameters and initialize the vehicle states. Here two main scripts in MATLAB where developed to elaborate data gathered from CARLA and initialize the simulation variables. The first code developed is to clean the data collected from the simulations in CARLA and stored in text files. The name of the MATLAB file is *trajectory_cleaning*. This

step is necessary because, due to the high sampling rate in CARLA, there are many sampled points, in terms of vehicles states, that are too close to each other and become insignificant data when inserted as reference in the control system. Furthermore, they would increase the computational effort in the "for" loops in control system functions, as we will see later, where in the reference matrices there would be too many points not needed, cause too close each other. In fact the major contribution of this function is to eliminate from the data written in the text documents points on the trajectory closer than 1 mm, a distance that for a trajectory in meters is not significant and that still leaves the trajectory well detailed. All the data taken at these points, which are also practically the same in terms of speed and other data, are then deleted while the significant data is rewritten to new text files, shown in Figure 4.1.

Figure 4.1: Text files before and after trajectory cleaning function

The second implemented MATLAB code is used to initialize all the parameters needed for the simulation in Simulink as well as to display the results and signals deriving from the simulation, it is called *nmpc_init_and_results*. In the first section of code the text documents containing the cleaned data deriving from the acquisition in CARLA are loaded, furthermore the tables that will be needed for dispatching are imported, acceleration map and the vector with the respectively input command values, that will be discussed in the next sections. The text files are then broken down to obtain a vector for each collected data (time, positions, velocities, accelerations, throttle, brake etc..). Finally the reference trajectory is generated starting from the collected references of the waypoints that also the autopilot and therefore the PID based controller of CARLA follows. In the second section all parameters of the model-based predictive controller are initialized, function containing the model dynamics, function containing state constraints if present, model states number, sampling time and prediction horizon, matrices with weights related to the cost function to be optimized, lower and upper limits of the signals. By means of the function *nmpc_design_m* then these controller information is stored in the K structure in the workspace and will be used by the controller to generate the optimized inputs. In the third section parameters regarding the Simulink simulation are initialized, among these the sampling time to be inserted in the various Simulink blocks, parameters used in the Simulink functions and initialization of the vehicle states by means of collected data. Finally there is a section to be launched on its own after the simulation on Simulink

has been performed in order to print all the data and signals resulting from the simulation.

4.5 Simulink Control System

In this section the control system architecture will be presented. It is divided into three big parts from left to right, first perception and path generation, then NMPC control block, then system manipulation and feedback loop. In figure 4.2 is shown the Simulink control system design.

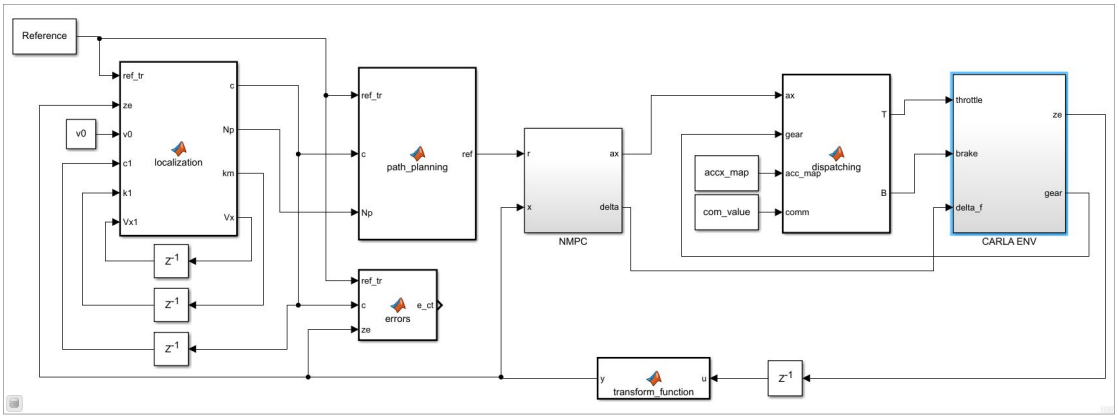


Figure 4.2: Simulink control system

All the functions developed thought out this thesis work will be presented next.

4.5.1 Transform Function

First of all, it was essential to understand what kind of data was returned by the Python API functions called inside the MATLAB system block, which communicates via client with the simulation world in CARLA, in order to communicate correctly with the NMPC controller and the whole system. It's worth to note that CARLA *get()* commands return position, location, angles, and velocities in CARLA absolute frame, while in the vehicle dynamical model velocities in x and y are in the vehicle car frame. Another point to take into account is the fact that the position returned by Python API is the center of the vehicle bounding box in CARLA, while the dynamic of the vehicle is built on the center of mass position. As discussed before in a previous section the CoG stands 0.2 m forward in the longitudinal direction towards the front axle. After a delay block of one time step needed to let the algebraic loop work in Simulink, a function in the feedback junction of the controller was added to manipulate data exiting from CARLA environment taking

into account that velocity frames are different and controller needs CoG vehicle position to work properly. Implementation is shown below.

Listing 4.3: Simulink transform function

```
function y = transform_function(u)

psi = u(3,:);
X = u(1,:);
Y = u(2,:);
xcog = X+0.2*cos(psi);
ycog = Y+0.2*sin(psi);
Vx = u(4,:);
Vy = u(5,:);
vx = vx.*cos(psi)+vy.*sin(psi);
vy = vx.*sin(psi)-vy.*cos(psi);

y = [xcog;
     ycog;
     u(3,:);
     vx;
     vy;
     u(6,:)];
```

Where u are the states data of the ego-vehicle from CARLA.

4.5.2 Localization Function

This function take care of the localization of car position inside the reference trajectory and also of the computation of the curvature of the trajectory in the following meters, it uses the vehicle states from the feedback as well as constants and other signals delayed from the same function at previous instant. Then return the position index in the trajectory reference of the estimated ego vehicle position and the value of prediction meters that will be passed to the next functions.

It starts with the computation of the closest point to the vehicle position of the reference trajectory to understand at which point of the trajectory the car is located. Then there are some lines of code to face the problem of localization when the car is intersecting same points of the trajectory as previously traveled or that would be traveled in future. In this situation there can be a sort of bug in which the localization of the ego vehicle jumps from one position to another in the trajectory. In the figure 4.3 you can see an example of a situation in which this problem could occur. To overcome this the closest point found in the trajectory w.r.t. vehicle's position is compared whit the trajectory position index in the previous time instant. In particular it should not be less than that at the instant before because otherwise we would be at an earlier point in the trajectory which would not make sense; Also, the index at the present time cannot exceed the previous one by 40 positions as it

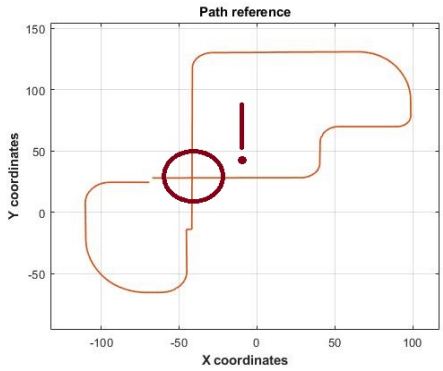


Figure 4.3: Localization bug situation

would then have identified as a current location a point in the trajectory that it will intersect in the future. If one of this two constraints are violated the position of the vehicle is approximated by the following steps. The distance the vehicle has traveled since the previous instant is estimated by knowing the velocity at the previous instant and multiplying it by the sampling time, which is the time between one acquisition instant and the next. Then with a for loop the distances of the points on the trajectory are added up from the previous position until the estimated distance traveled is exceeded, thus finding the estimated position on the trajectory at the current time. In the second part of the function the average curvature is calculated in the meters following the estimated position of the vehicle. In this case since the controller has been designed for urban speeds, therefore from 30km/h base up to 50km/h, the average curvature is calculated in the following 23 m. This parameter for future applications can be modified at will based on the cruising speed also dynamically during the simulation, for our interests 23 fixed meters are good enough. In particular a point is taken at every meter of distance along the desired distance and using the *LineCurvature2D* function (see Appendix A) the curvature values at each point taken are calculated and inserted into a vector. Once the null values in the vector are eliminated they are averaged to obtain the average curvature. This is then compared to the value obtained at the previous instant to ensure that there are no changes in curvature that are too sudden, in this case greater than a value of 0.05 m^{-1} which in meters of radius of curvature are equivalent to 20 meters. It is possible to apply these constraints in our case since the trajectory to follow is well known a priori as well as the scenarios in which we operate. The average curvature value is then used in an equation created specifically for our needs in terms of desired speed and behavior that returns the desired speed value for that stretch of path, where *v0* is a constant corresponding to the desired speed at zero curvature, in our case 23 to obtain 30km/h. This is one of the key equations that determines the desired speed of the car based on the

calculated curvature, in this case average, of the path. This allows the vehicle to slow down more or less in the case of more or less sharp curves. This equation in future scenario studies is also fully modifiable based on the case studied in order to obtain different types of behavior. Finally Np value is computed from desired velocity ad given to the next function that will calculated the path in the following Np meters.

Listing 4.4: Simulink localization function

```
function [ref_pos,c,Np,km,Vx] = Np_gen(ref_tr, ze, v0, c1, Vx1, k1)

N = size(ref_tr,1);
pose = ze(1:6);
Vx = ze(4);
Tp = 3;
Ts = 0.05;

% 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_pos = ref_tr(c,:)';
if c>c1+40 || c<c1 %avoid localization bug in intersection
    dist_prevision = (Vx1/Tp)*Ts;
    if dist_prevision<0.2
        dist_prevision = 0.2;
    end
    for k = c1:N
       gap = 0;
       if c1==0
           d = 2;
       else
           d = c1;
       end
       for j = d:k
           gap = gap+norm(ref_tr(j,:)'-ref_tr(j-1,:)');
       end
       if gap>dist_prevision
           c = k;
           ref_pos = ref_tr(c,:)';
           break
       end
    end
end

% Curvature
CD = 23;
index1 = zeros(CD,1);
index1(1) = c;

```

```
dist = 1;
for i = 1:length(index1)-1 %find points of 1 m distance between each other and place the trajectory index in the vector
    for k = c:N
        gap = 0;
        for j = c+1:k
            gap = gap+norm(ref_tr(j,:)'-ref_tr(j-1,:)');
        end
        if gap>dist
            index1(i+1) = k;
            dist = dist+ 1;
            break
        end
    end
end

Vertices = zeros(CD,2); %build matrix with all points found previously
for i = 1:length(index1)
    if index1(i)==0
        index1(i) = N;
    end
    Vertices(i,:) = ref_tr(index1(i),:);
end

Lines = [(1:size(Vertices,1))' (2:size(Vertices,1)+1)']; Lines(end,2) = 1;
k = LineCurvature2D(Vertices,Lines,CD);
k = k(~isnan(k)); %delete not a number values
km = mean(k);
if km<k1-0.05
    km = k1-0.05;
elseif km>k1+0.05
    km = k1+0.05;
end

v = v0/(abs(km)*10+1); %scenario dedicated, set desired velocity based on path curvature

if v<10
    v = 10; %set minimum velocity, scenario dedicated
end

Np = v/3.6*Tp;
```

4.5.3 Path Planning Function

Once the desired velocity and thus the meters on which the prediction will be made have been defined in previous function, this function has the purpose of selecting the points that will be passed to the NMPC controller itself for optimization and the calculation of the optimal inputs to be provided to the vehicle. In our case the optimization logic works with 50 reference points. Using multiple concatenated for loops and vectors for the position indices in the reference matrices, the prediction meters are divided into 50 equidistant points and for each of these, starting from the current position of the vehicle, a point is taken in the reference trajectory with the desired distance. In this way the 50 points on the reference trajectory are obtained, which will be reworked through the MATLAB 'reshape' function to give as output from the function a row vector of length 50 times number of references passed, in this case x and y values of the position points and therefore a row vector of length 100.

Listing 4.5: Simulink path generation function

```
function ref  = path_planning(ref_tr, c, Np)

N = size(ref_tr,1);

Ns = 50;
index = zeros(Ns,1);
target_dist = linspace(2,Np,Ns);

for i = 1:length(index)
    for k = c:N
        gap = 0;
        for j = c+1:k
            gap = gap+norm(ref_tr(j,:)'-ref_tr(j-1,:)');
        end
        if gap>target_dist(i)
            index(i) = k;
            break
        end
    end
end

ref_init = zeros(2,Ns);

for i = 1:length(index)
    if index(i)==0
        index(i) = N;
    end
    ref_init(:,i) = ref_tr(index(i),:)';
end

```

```
ref = reshape((ref_init),Ns*2,1);
end
```

4.5.4 Error function

Error function is crucial to understand controller's performance. In particular, to measure the tracking error we refer to the cross track error (CTE). To calculate the deviation of the current position from the ideal trajectory this function calculates the point-to-line distance between the current position of the vehicle and the line passing through the two points closest to it on the reference trajectory. By knowing the index c of the vehicle position in the reference matrix, calculated in the localization function, this identifies the first of the two points, to find the second one the distances between the current position and the point before and after the first point identified by c index are calculated and the one that is closest to current location is chosen. Finally as can be seen below using the implemented formulas the distance between the current position of the vehicle and the line passing through the two chosen points is calculated.

Listing 4.6: Simulink errors function

```
function [e_ct] = errors (ref_tr, c, ze)
N = length(ref_tr(:,1));

% Cross-track error
%chosing the two points nearest to current position
if c<N
    e1 = vecnorm(ref_tr(c-1,:)'-ze(1:2));
    e2 = vecnorm(ref_tr(c+1,:)'-ze(1:2));
    if e1>e2
        e = c+1;
        d = c;
    else
        e = c;
        d = c-1;
    end
else
    d = c-1;
    e = c;
end

x = ref_tr(d:e,1)';
y = ref_tr(d:e,2)';
m = (y(1)-y(2))/(x(1)-x(2));
q = (x(1)*y(2)-x(2)*y(1))/(x(1)-x(2));

zex = ze(1);
```

```
zey = ze(2);

e_ct = abs(zey-(m*zex+q))/sqrt(1+m^2); %Point to line distance
```

4.6 Identification of the Vehicle Model

To control the vehicle in the Carla simulator, we need to convert the optimal acceleration data ($\tilde{a^X}$) generated by our nonlinear model predictive control algorithm into throttle and brake commands that the client can apply to the ego vehicle in simulation. Since a^X represents the acceleration (if positive) or deceleration (if negative) of the vehicle, the challenge is to map this data to the appropriate controls to apply to Carla vehicle. The commands that can be applied to the vehicle in CARLA through Python API are throttle, brake and steering angle in degrees. For steering the optimal command calculated by the controller is applied directly. Instead calculating the accelerator and brake commands is more complex and requires studying the longitudinal dynamics of the vehicle. In particular it is necessary to identify the accelerations corresponding to each accelerator and brake input command. The values applicable to the accelerator and brake commands range from 0 to 1 for each of the two, where zero means respectively no accelerator and no brake while 1 means respectively full throttle and full brake. To identify the longitudinal dynamics, simulations were launched in manual driving where specific accelerator and brake commands were imposed, through the study of the acquired data it was possible to notice how the acceleration values were mainly dependent on the throttle and brake command and the gear in which the vehicle was rather than speed. These statements can be considered valid in our scenario as the car has an automatic transmission that has been seen to always behave in the same way in gear upshifts and downshifts, changing the gear always at the same speed and engine rpm consequently. In Figure 4.4 it is possible to find the plot of acceleration values for the throttle and brake commands specified respectively, and related to the corresponding gear. We thus calculated the average acceleration values for a set of discerned throttle and brake commands. Where to calculate the deceleration obviously we started from higher speeds until arriving at vehicle stopped. Values are only taken in gear one to four since this permits to cover speeds from 0 km/h to 125 km/h when the fifth gear is inserted. Resulting average accelerations for second gear are plotted in figure 4.5, where throttle inputs goes from 0 to 1 and brake values goes from -1 to 0, considering that throttle and brake are never used together.

As can be noted for throttle and brake values equal to 0 both negative acceleration is obtained while no acceleration is obtained with throttle values around 0.5 throttle, this demonstrates the importance of the system analysis in order to obtain a more

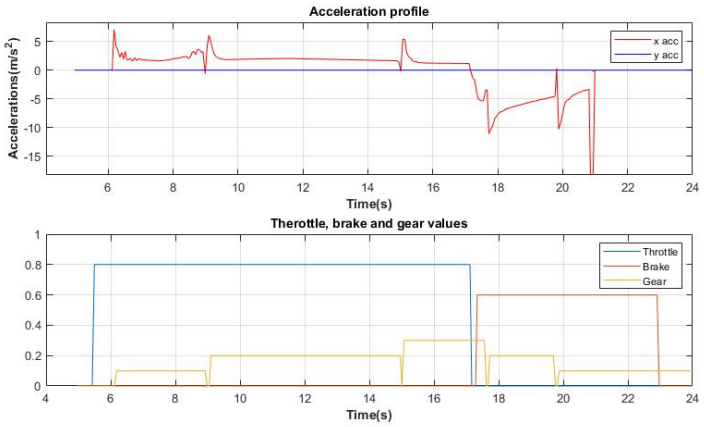


Figure 4.4: Study of longitudinal acceleration under specific throttle and brake input, 0.8 and 0.6 respectively

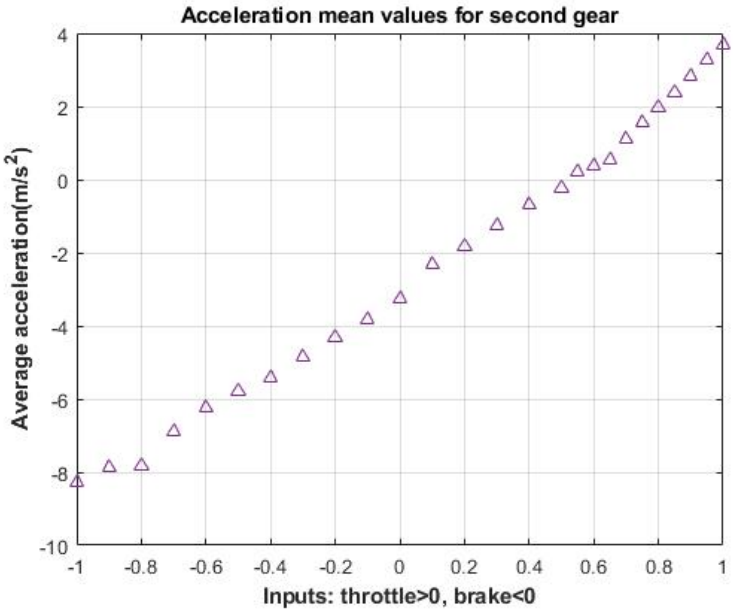
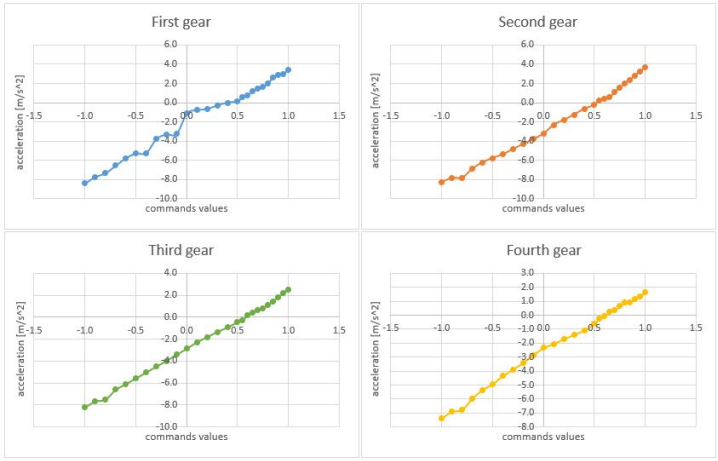


Figure 4.5: Average acceleration values in second gear

complete dispatching that provides the right inputs to have the accelerations desired by the controller. Even in vehicles that we drive every day, if we pay attention to it, due to friction, with little throttle pressed in while moving the car decelerates. In figure 4.6 you can see all the mapped acceleration values related to the various commands, where the brake command has changed its sign to negative for ease of representation.

		1st gear [m/s^2]	2nd gear [m/s^2]	3rd gear [m/s^2]	4th gear [m/s^2]
Brake	-1.00	-8.397	-8.298	-8.205	-7.392
	-0.90	-7.730	-7.875	-7.652	-6.897
	-0.80	-7.339	-7.849	-7.488	-6.768
	-0.70	-6.566	-6.905	-6.607	-5.957
	-0.60	-5.783	-6.246	-6.083	-5.339
	-0.50	-5.283	-5.790	-5.556	-4.931
	-0.40	-5.255	-5.405	-5.023	-4.336
	-0.30	-3.796	-4.850	-4.489	-3.889
	-0.20	-3.312	-4.339	-3.970	-3.397
	-0.10	-3.268	-3.814	-3.392	-2.844
None	0.00	-1.149	-3.239	-2.869	-2.333
Throttle	0.10	-0.787	-2.318	-2.263	-2.051
	0.20	-0.670	-1.828	-1.810	-1.703
	0.30	-0.287	-1.245	-1.333	-1.381
	0.40	0.000	-0.672	-0.921	-1.081
	0.50	0.136	-0.239	-0.442	-0.602
	0.55	0.552	0.212	-0.251	-0.246
	0.60	0.769	0.365	0.217	-0.056
	0.65	1.167	0.546	0.458	0.220
	0.70	1.463	1.089	0.655	0.384
	0.75	1.650	1.562	0.821	0.656
	0.80	2.013	1.986	1.121	0.923
	0.85	2.590	2.391	1.471	0.937
	0.90	2.902	2.821	1.820	1.146
	0.95	2.953	3.281	2.183	1.375
	1.00	3.435	3.690	2.553	1.650

(a) *Acceleration values for first to fourth gear*



(b) *Plotted acceleration values for each gear and commands*

Figure 4.6: Complete acceleration map for first to fourth gear

Once this sort of engine map, that can be seen in Figure 4.6, has been created, which we will call acceleration map, it must be implemented in the dispatching function in order to provide the vehicle with the right commands. Certainly in the case of more complex scenarios or studies the detail of the map or the methods for identifying the system can vary, in our case the complexity and the level of definition used for the identification has proven to be optimal for obtaining the desired vehicle acceleration values.

4.6.1 Dispatching Function

The dispatching function is then needed in order to compute the correct acceleration and brake values based on the received requested acceleration, the gear in which the vehicle is, the matrix containing the acceleration map and the vector containing the corresponding acceleration and brake values. If the car has to start from a standstill and positive acceleration is requested, a throttle command of 0.75 is given to allow the vehicle to engage first gear. Subsequently if the requested acceleration or deceleration values are greater than those obtainable with the maximum command for the current gear, the accelerator or brake output command is set to maximum. If we are not in one of the previous cases the function searches for the command values among which the desired acceleration or deceleration is present and through linear interpolation between the two commands and their respective acceleration values the precise input command is calculated. It is necessary to calculate the

value and subsequently if it is positive apply throttle with that value and brake at zero, while if negative set the brake with that value changed in sign and throttle equal to zero. Throttle and brake are never used together and therefore one of the two commands is always zero.

Listing 4.7: Simulink dispatching function

```
function [T,B,ind] = fcn(ax, gear, acc_map, comm)
gear = round(gear);
T = 0;
B = 0;
C = 0;

if gear==0 && ax>0.1
    C = 0.75;
elseif gear==0 && ax<0.1
    C = 0;
elseif ax>acc_map(end,gear)
    C = comm(end);
elseif ax<acc_map(1,gear)
    C = comm(1);
else
    for i = 2:26
        if ax>acc_map(i-1,gear) && ax<acc_map(i,gear)
            C = comm(i-1)+(ax-acc_map(i-1,gear))*(comm(i)-comm(i-1))
    /(acc_map(i,gear)-acc_map(i-1,gear));
            break
        end
    end
end

if C==0
    B = 0;
    T = 0;
elseif C>0
    T = C;
elseif C<0
    B = -C;
end
```

Chapter 5

Simulation Results

5.1 Path Tracking in Urban Scenario

This section presents the results of simulations conducted in the CARLA environment to evaluate the performance of the developed control system. Through a series of tests in varied driving scenarios, the goal is to assess the system's responsiveness, stability, and adaptability under conditions that closely mimic real-world testings. The scenarios examined will range from the simplest to verify performance in terms of path tracking and response to commands to more difficult ones such as obstacle avoidance and overtaking to verify the functioning of the control with decision making functions in the presence of other vehicles.

The results are analyzed to determine how effectively the control algorithms manage vehicle behavior in urban settings, respond to dynamic obstacles, and adapt to sudden changes in control factors. Specific scenario performance will be discussed, offering insights into the strengths of the system and identifying areas for potential improvement. By demonstrating the system's functionality and resilience within CARLA, these simulations provide a critical validation step, supporting the readiness of the control system for real-world deployment.

In this first scenario, we examined the control system's ability to accurately follow a predefined path within the CARLA simulation environment. The Town10HD map was chosen for this initial scenario as it represents an urban environment with a significant variety of road features, including both tight and wide curves, as well as single-lane and two-lane roads. The path reference chosen is the one as illustrated in Figure 5.1. The path is composed by two medium curves to the left and two narrow curves to the right. In between there are two straight way.

This scenario focuses on evaluating speed behavior of the ego vehicle and path tracking performance, specifically by measuring how closely the vehicle adheres to the intended trajectory. A key performance metric in this assessment is the cross

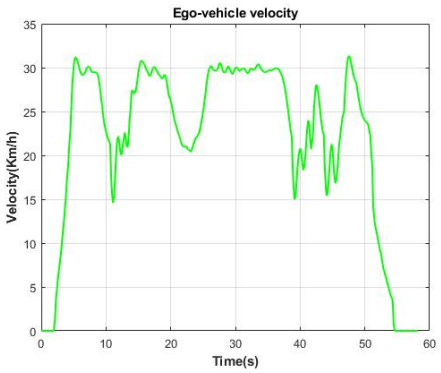
Figure 5.1: Path reference in CARLA

track error, which represents the lateral deviation of the vehicle from the desired path. Lower CTE values indicate a more precise adherence to the path, reflecting the system's effectiveness in maintaining the desired trajectory.

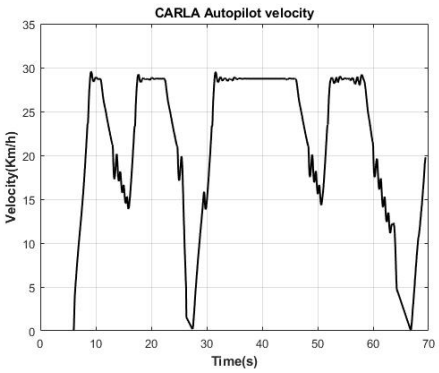
For this analysis, we compare the performance of two control strategies: the Nonlinear Model Predictive Control implemented within our system, and CARLA's built-in autopilot, which relies on a PID-based control approach. By comparing the CTE values achieved by each method, we aim to gain insights into the relative strengths of NMPC in path tracking and assess its advantages over the traditional PID-based control in CARLA's autopilot. Nevertheless a further comparison between developed control system and manual driving will be presented to understand similar and different aspect of vehicle behavior and performance under either NMPC system control and human driving.

5.1.1 NMPC vs CARLA Autopilot

In the performance comparison between NMPC system and CARLA PID based autopilot firstly we want to check if the ego-vehicle velocity performances are aligned with the design specification. In figure 5.2 is shown the plot with the ego vehicle velocity along the path and also the velocity on the same path of the vehicle on autopilot mode in CARLA. As for design implementation the vehicle maintains a stable speed around 30 km/h on the straight road. Additionally, is well notable the speed adjustments when approaching and during curves allowing deceleration until around 15 km/h in curves and the accelerating again up to 30 km/h on straight. This indicates that NMPC system responds to the design requirements adjusting the vehicle's speed in response to changes in path curvature, which is



(a) *Ego-vehicle velocity during path tracking scenario*



(b) *CARLA Autopilot velocity during path tracking scenario*

Figure 5.2: Comparison between NMPC and CARLA Autopilot velocity

shown in Figure 5.3, optimizing for stability and control. Toward the end of the

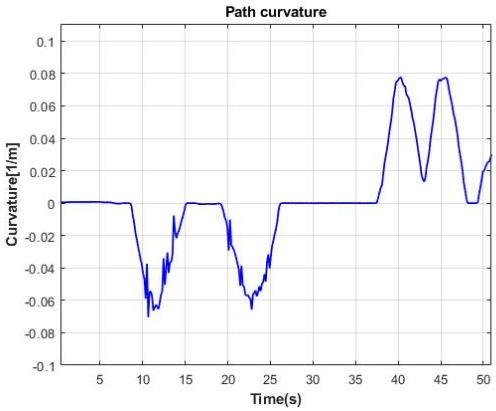
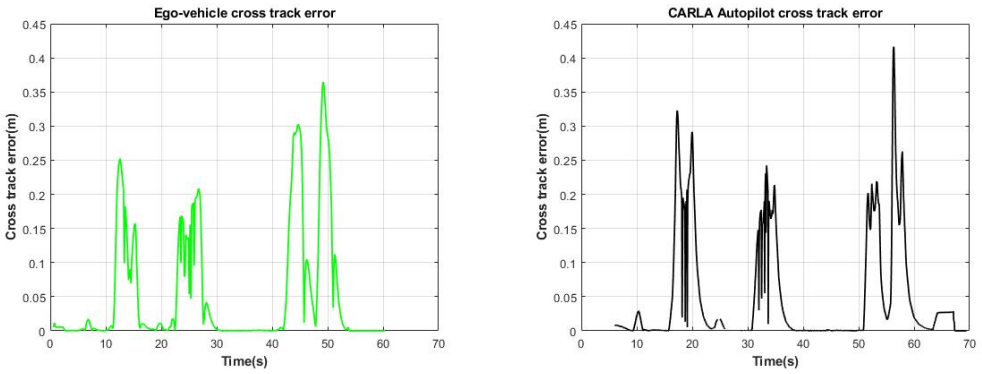


Figure 5.3: Path curvature calculated by system *localization* function

trajectory (around 50 seconds), the vehicle decelerates significantly, approaching the end of the path. This velocity profile highlights NMPC's ability to adapt speed dynamically in response to path characteristics, contributing to smoother and safer path tracking. It can also be noted, with regards to the speed comparison, that the CARLA autopilot has a speed profile very similar to that obtained by the control system, confirming the excellent performance obtained by integrating the developed system with CARLA.

Moving into the analysis in term of Cross Track Error over time for two control systems, the Nonlinear Model Predictive Control implemented system and CARLA's PID-based autopilot, the results are shown in Figure 5.4. The NMPC control system



(a) *NMPC cross track error* **(b)** *CARLA Autopilot cross track error*

Figure 5.4: Comparison between NMPC and CARLA Autopilot cross track error

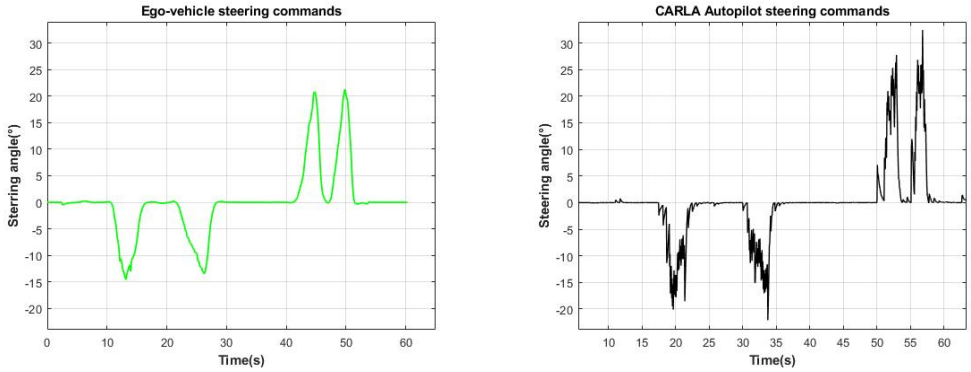
exhibits generally low CTE values, with only a few peaks reaching approximately 0.35 meters. The error remains close to zero for most of the trajectory, indicating precise path tracking. The few peaks suggest slight deviations, occurring in areas where the path include curves. Overall, NMPC demonstrates good accuracy, effectively minimizing lateral deviations. CARLA's PID based autopilot shows a similar trend but with slightly more pronounced peaks, with some CTE values exceeding 0.4 meters. While the PID based CARLA controller performs reasonably well, its error is slightly higher and less consistent than NMPC's. The increased fluctuations indicate that the PID based system may have struggled more to adapt to sudden changes in path direction, showing its limitations in complex tracking scenarios compared to NMPC. By exploiting commonly used performance metrics such as the Root Mean Square Error (RMSE), an indicator that measures the root mean square error between the observed values and the desired values, and the Max Value Error, which indicates the maximum absolute error committed by the observed values, we confirm the better results obtained by the NMPC-based system compared to the CARLA PID-based autopilot, as shown in Table 5.1.

	RMSE Value [m]	MAX Value Error [m]
NMPC system	0.200	0.361
CARLA autopilot	0.261	0.415

Table 5.1: NMPC system vs CARLA autopilot performance parameters in first scenario

As shown in Figure 5.5, the steering commands generated by the Autopilot PID-based controller and applied to the vehicle during autonomous driving in

CARLA are less smooth compared to those generated by the NMPC controller. The NMPC commands are much more linear and exhibit fewer oscillations. This happens because the NMPC is able to predict the behavior and the trajectory of the vehicle by generating command inputs that are less marked and more long-lasting over time, while the autopilot must correct the inputs at any moment based only on the direction of the vehicle and the waypoint to be reached, resulting in less discrete commands. This improved behavior is a significant advantage even in a real vehicle in terms of actuation. It reduces stress on the steering components, minimizes wear and tear, and avoids situations where excessively abrupt command changes might damage the vehicle or lead to suboptimal responses. Here NMPC



(a) *NMPC steering commands* **(b)** *CARLA Autopilot steering commands*

Figure 5.5: Comparison between NMPC and CARLA Autopilot steering commands

shows his strength in predicting vehicle behavior and optimizing path tracking with robust trajectory generation.

Figure 5.6 presents a comparison between the acceleration outputs requested by the NMPC controller and the actual accelerations achieved by the vehicle after applying the corresponding throttle and brake commands calculated in the dispatching function. This comparison highlights the effectiveness of the system identification process, where a comprehensive mapping was created to correlate acceleration values with specific throttle and brake inputs. By using this mapping, the vehicle can achieve the desired accelerations accurately, closely following the intended behavior dictated by the NMPC controller. This high fidelity in replicating the requested accelerations enhances the controller's overall performance in every point of the path, in fact a correct dispatching permits to maintain the constant speed when needed with low acceleration or deceleration applied, while during strong acceleration and deceleration phases it allows to correctly have the desired behavior. This is also more crucial during turns when have a different acceleration

from the one calculated by NMPC algorithm could lead to control loosing caused by the sum of already present lateral forces and moments and wrong acceleration or deceleration forces.

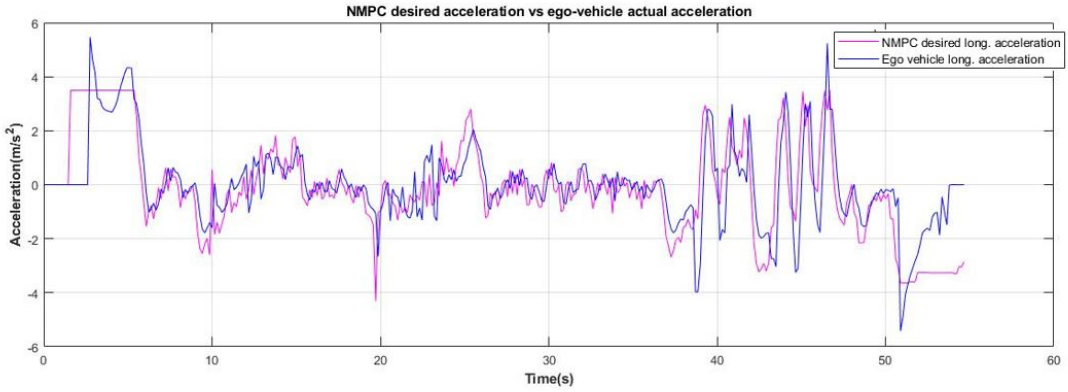


Figure 5.6: Ego-vehicle acceleration compared to NMPC desired acceleration during path tracking scenario

The minor delay observed in the vehicle's response is due solely to the inherent delay block in the control loop, where the desired accelerations are realized one sampling period after the initial request. Despite this delay, the response time is very short with respect to the vehicle's dynamic changes, ensuring that the NMPC operates effectively even during rapid acceleration or deceleration events. This precise and timely response allows the control system to function optimally, ensuring smooth and stable vehicle behavior.

The yaw rate, expressed in radians per second, measures the angular velocity around the vehicle's vertical axis. It is a key indicator of a vehicle's dynamic stability, as it reflects how the vehicle behaves during maneuvers such as turning, braking, or accelerating. In stable conditions, the yaw rate should remain proportional to the steering input, indicating that the vehicle is following the intended trajectory. Excessive or fluctuating yaw rates may signal instability, such as oversteer, understeer, or loss of lateral grip. Yaw rate data for path tracking scenario are shown in Figure 5.7. Early in the timeline (0–10 seconds), the yaw rate remains close to zero, indicating minimal angular movement and stable driving conditions. Between 10 and 30 seconds, the yaw rate dips negatively, reaching a minimum of around -0.4 rad/s, likely corresponding to controlled leftward maneuvers. These fluctuations, although pronounced, remain within a range that does not suggest any loss of stability. From 40 to 50 seconds, the yaw rate increases significantly, with peaks approaching 0.7 rad/s. These positive values correspond to rightward strict turns, the symmetry and smooth nature of the peaks suggest

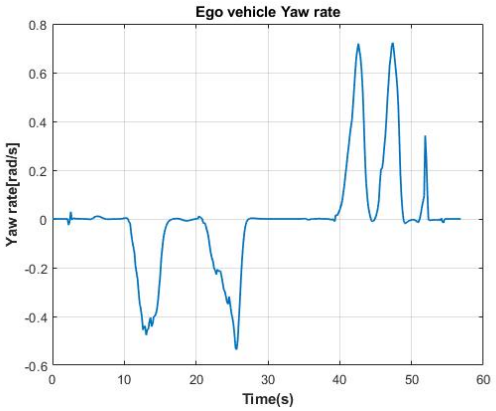


Figure 5.7: Ego vehicle yaw rate during path tracking scenario

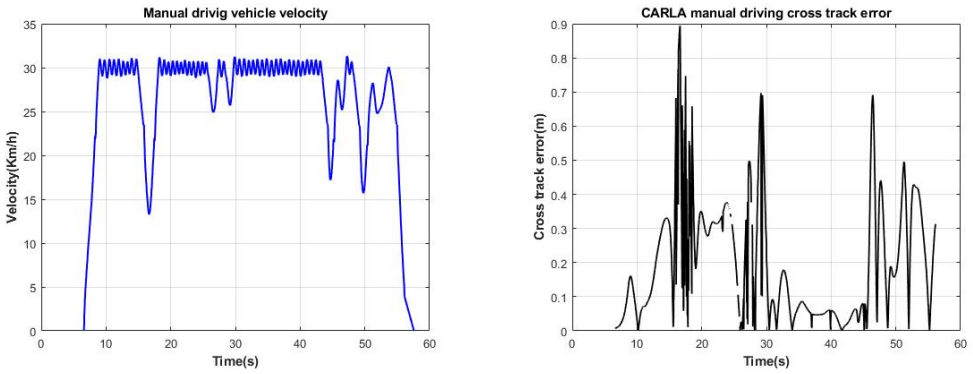
deliberate and controlled vehicle movements. Notably, the yaw rate stabilizes back near zero after 50 seconds, confirming that the vehicle returned to a steady state. The absence of extreme oscillations or abrupt discontinuities indicates that the vehicle maintained stability throughout the test. The recorded yaw rate values, while reflecting dynamic maneuvering, remain well within a range consistent with safe and stable handling. This suggests that the vehicle control system effectively managed all lateral forces without losing traction or control and ensuring optimal path tracking.

5.1.2 NMPC vs Manual Driving

To further understand the vehicle's behavior and validate if the target speeds align with the design specifications, a series of manual driving with steering wheel and pedals tests were conducted within the CARLA environment. These manual trials serve as a benchmark, allowing us to observe the range of vehicle speeds achievable under human control and to examine typical Cross Track Error values when navigating the designated path without automated control.

By analyzing the CTE generated during manual driving, we can establish a reference for the lateral deviations that occur in real-time, driver-controlled conditions. This comparison provides a valuable context for evaluating the automated control systems' performance and helps identify the baseline level of path accuracy achievable through manual intervention.

The results in Figure 5.8 indicate that the speeds achieved by the developed control system are highly consistent with those observed during manual driving, where speed was limited at 30 km/h maximum for better comparison, and in particular in cornering scenarios. In curves, the NMPC system results demonstrates deceleration patterns closely matching those of human drivers, with comparable



(a) *Manual driving velocity* **(b)** *Manual driving cross track error*

Figure 5.8: Manual driving with steering wheel in CARLA results

target speeds reached in these sections.

In terms of Cross Track Error, the autonomous NMPC control system shows a marked improvement over manual driving. While manual driving produced CTE values exceeding 0.5 meters and reaching up to 0.9 meters, the autonomous system significantly reduces this lateral deviation, especially in curves. This improvement is particularly notable in challenging turns where visual references for manual guidance are limited, underscoring the system's ability to maintain a more accurate path in complex sections of the route. Comparing manual driving to NMPC also in terms of RMSE and Max value error as we expected performances of the NMPC system are much better, as demonstrated in Table 5.2.

	RMSE Value [m]	MAX Value Error [m]
NMPC system	0.200	0.361
Manual Driving	0.565	0.903

Table 5.2: NMPC system vs manual driving performance parameters in first scenario

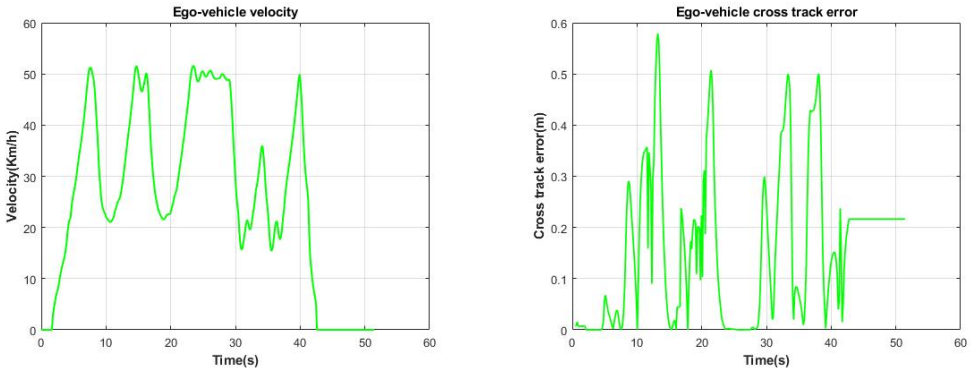
5.2 Path tracking with augmented velocity

This section describes the testing of the developed control system under operating conditions with speeds higher than the design specifications. Specifically, the system was tested at a cruising speed of 50 km/h. The route used for this test is the same as in the first scenario, characterized by a mixed track with medium and

tight curves in an urban environment. This type of route allows for evaluating the controller's behavior robustness at higher speed on both straight and curved section. To enable the controller to operate under the new specifications, the following changes were made into the *localization* function block in Simulink model at line 74 where is generated the desired speed:

- Modification of the *v0* parameter: The constant parameter *v0* was adjusted to a value of 43, ensuring that in the absence of curvature in the path, the vehicle travels at a speed of 50 km/h.
- Increased curvature weighting: The weighting assigned to curvature was increased from 10 to 30. This adjustment ensures more pronounced deceleration in tight turns, enhancing safety and maintaining vehicle control in the most demanding conditions of the route.

The resulting speed profiles, as shown in Figure 5.9, demonstrate that the controller can seamlessly adapt to higher speeds by adjusting the system's constants. As intended the speed decreases near curves, enhancing vehicle safety and stability. On straight sections, the speed remains consistent with the desired value. This highlights the system's effectiveness in maintaining cruising speed under low curvature conditions and adequately decelerating near tighter curves. In terms of cross-track error, the errors remain very low, even considering that the curves are tackled at higher speeds compared to the previous scenario, and that the curves are among the tightest present on the CARLA map. The error does not exceed 0.6 meters, confirming the excellent performance of the controller. This test allows for assessing the robustness and adaptability of the controller



(a) *Ego-vehicle velocity* **(b)** *Ego-vehicle cross track error*

Figure 5.9: Comparison between NMPC and CARLA Autopilot cross track error in augmented velocity scenario

in more complex operating scenarios, confirming the system's ability to manage

higher speeds while maintaining stable and safe behavior. While there is space for improving system performance at 50 km/h, comparing RMSE and Max value errors, system performances at higher velocity are slightly worse than those obtained at design speed of 30 km/h as shown in Table 5.3

Working velocity	RMSE Value [m]	MAX Value Error [m]
30 km/h	0.200	0.361
50 km/h	0.342	0.585

Table 5.3: NMPC system performance parameters comparison between first and augmented velocity scenario

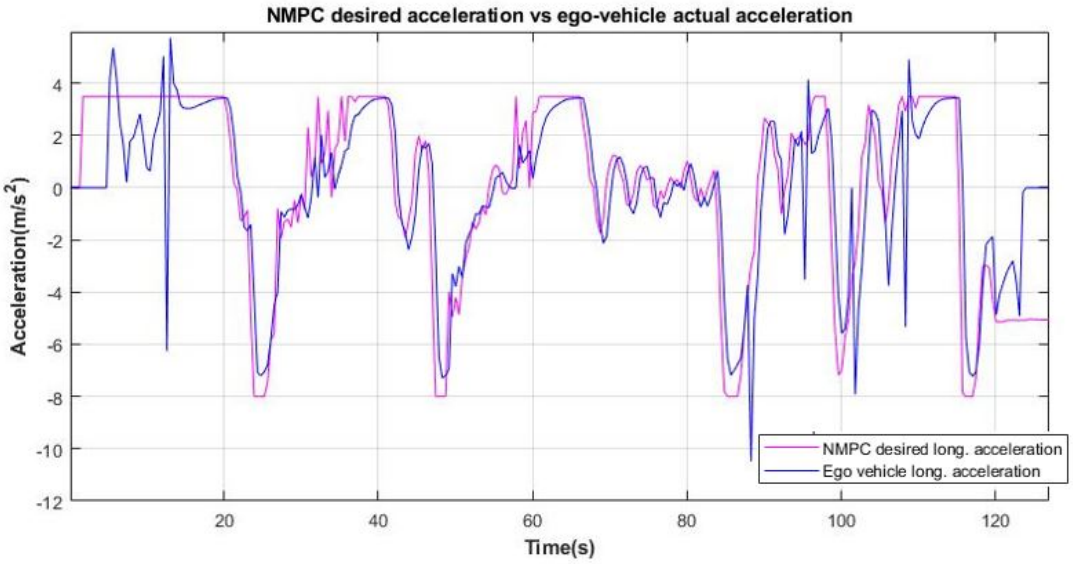


Figure 5.10: Ego-vehicle acceleration compared to NMPC desired acceleration in augmented velocity scenario

Figure 5.10 compares the acceleration outputs requested by the NMPC controller with the actual accelerations achieved by the vehicle, following the throttle and brake commands generated by the dispatching function. The results are even more significant at 50 km/h compared to 30 km/h scenario, as not only the first two gears but also the third gear is used. This further confirms the success of the system identification process. The spikes in the graph correspond to gear shifts, where the transmission momentarily enters neutral, causing brief negative accelerations.

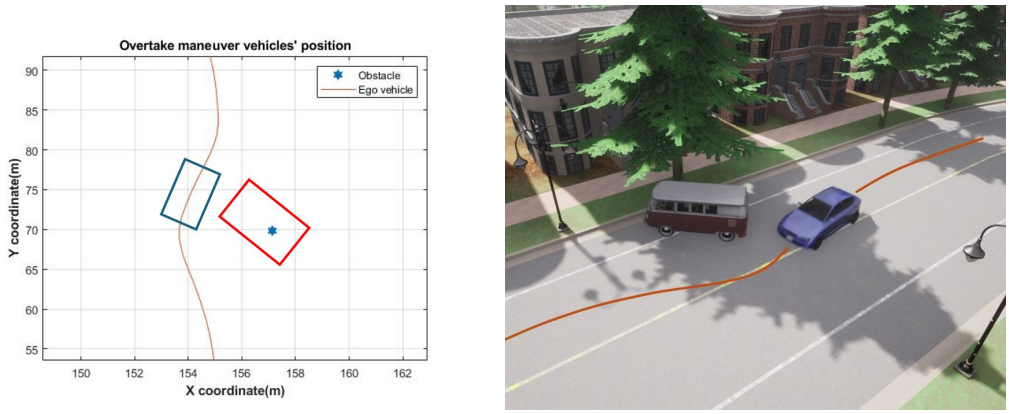
In conclusion, the results demonstrate that the control system is capable of

effectively operating at speeds different from the design speed. Specifically, the system performs well not only at the target speed of 30 km/h but also at higher speeds, such as 50 km/h, where additional gears are engaged. This adaptability confirms the robustness and flexibility of the NMPC controller, ensuring it can maintain the desired performance across a range of operating conditions. The system's ability to handle varying speeds adjusting constant variables in the loop without compromising trajectory tracking or stability highlights its effectiveness and readiness for real-world applications with diverse driving scenarios.

5.3 Obstacle avoidance

Obstacle avoidance is a fundamental capability in autonomous driving systems, critical for ensuring safety and robustness in dynamic environments. Implementing an effective obstacle avoidance mechanism is essential to allow the ego vehicle to navigate seamlessly around static or dynamic obstacles while maintaining a predefined trajectory. This scenario demonstrates the integration of a trigger function within the control system to detect obstacles and execute avoidance maneuvers.

The scenario is run in a narrow rural road in Town05. To create the obstacle avoidance scenario, a vehicle obstacle, in particular a Volkswagen Transporter T2 van was placed stationary stopped with half the vehicle in the middle of the road simulating what could be an emergency parking. A supervisory control function was implemented to continuously monitor the ego vehicle's and the obstacle position. When the obstacle was detected within a specific threshold, distance relative to the ego vehicle, the avoidance maneuver was triggered by setting a trigger variable to 1. In response, the path generation function dynamically adjusted the reference path. The control system shifted the reference trajectory laterally by the distance needed to avoid the obstacle and not invade the opposite lane too much. In fact the function calculates how much the obstacle is near or overcome the reference trajectory and shifts the reference taking care of the dimensions of the ego vehicle. Once the ego vehicle cleared the obstacle and re-entered its original lane with a predefined safety margin, at least 5 meters ahead of the obstacle, the control variable was reset to 0, allowing the ego vehicle to resume its original trajectory, as can be seen in Figure 5.11. While in Figure 5.12 the steering angle imposed by the controller is shown, at the beginning to the left (negative values indicate turning the steering to the left) because initially the vehicle is coming from a previous curve and after about 2.5 seconds again to the left to avoid the vehicle and then to the right to return to the lane again. The yaw angle and yaw rate are also shown in the same figure, highlighting the changes in direction of the vehicle and its angular velocity which, however, remains contained in absolute value below 0.5 radians per

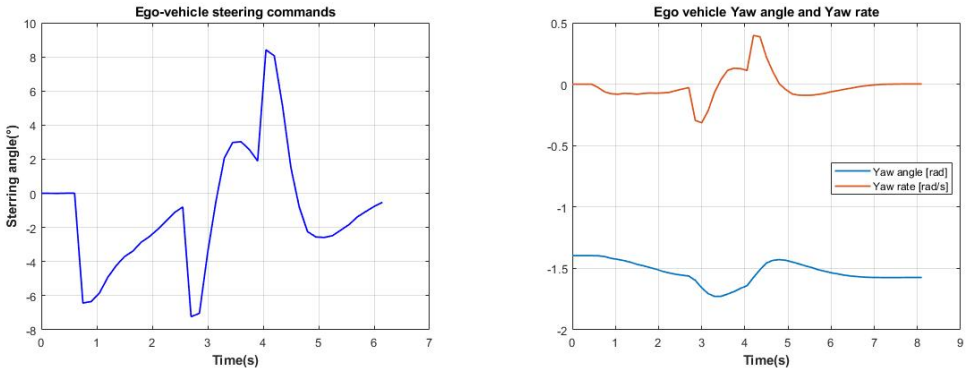


(a) *Ego-vehicle and obstacle vehicle positions*

(b) *Ego vehicle trajectory in CARLA environment*

Figure 5.11: Obstacle avoidance maneuver

second, values which highlight the correct stability of the vehicle when cornering.



(a) *Ego-vehicle steering commands*

(b) *Ego vehicle yaw angle and yaw rate*

Figure 5.12: Obstacle avoidance maneuver steering commands, yaw angle and yaw rate

This dynamic adaptation of the trajectory, combined with real-time supervisory control, enabled the ego vehicle to avoid the obstacle smoothly and efficiently, showcasing the adaptability of the Nonlinear Model Predictive Control system to real-world scenarios.

In addition, the NMPC controller also offers the possibility of managing obstacle avoidance scenarios by using the *fcon* function in which it is possible to express limits to be imposed on the vehicle states. Similar results to those obtained were

in fact achieved by inserting the aforementioned function inside the controller in which an ellipse of approximately the size and position of the obstacle to be avoided was identified.

5.4 Overtaking maneuver

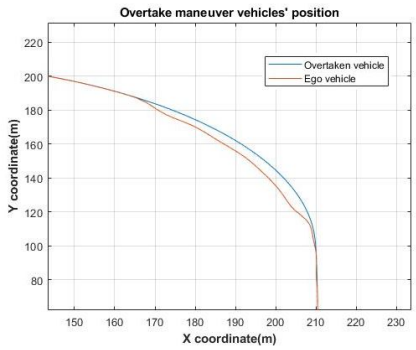
Overtaking is a critical behavior in autonomous driving, essential for ensuring efficient traffic flow and avoiding delays caused by slower vehicles. Implementing a robust overtaking mechanism in the control system is vital for achieving seamless navigation, particularly in dynamic multi-vehicle environments. This scenario highlights the importance of integrating a control/decision function to initiate and manage the overtaking process within the control system.

Town05 was chosen due to presence of extra-urban roads with multiple lane where a overtake maneuver can be executed. Then to create this overtaking scenario, a second vehicle was spawned approximately 20 meters ahead of the ego vehicle within the simulation environment, using MATLAB system for scenario configuration. The second vehicle was set to operate autonomously with CARLA autopilot at a 30 km/h speed. A supervisory control function was implemented to monitor the relative positions and speeds of the two vehicles. This function calculated the distance between them and triggered the overtaking behavior under specific conditions:

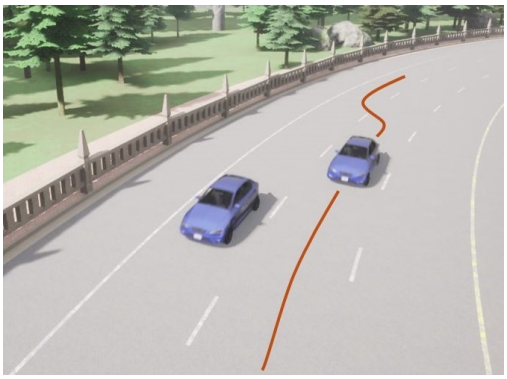
- The distance between the vehicles dropped below 15 meters.
- The leading vehicle's speed was slower than that of the ego vehicle.

When these conditions were met, a control variable was set to 1, signaling the need for an overtaking maneuver. Within the path tracking function, if the variable is set to 1, the reference path is adjusted by 3.5 meters to the left, corresponding to the next lane center offset, to enable the ego vehicle to shift lanes and initiate the overtaking maneuver. After overtaking, once the ego vehicle was at least 15 meters ahead of the previously leading vehicle, the trigger variable was reset to 0. At this point, the ego vehicle returned to its original predefined path, completing the overtaking process, as shown in Figure 5.13.

As seen in Figure 5.14, the ego vehicle maintains a speed of approximately 50 km/h throughout the scenario. The overtaking maneuver occurs on a curve, where the ego vehicle naturally decelerates in response to the curvature of the path. Despite the reduced speed in the curve, the ego vehicle remains faster than the leading vehicle, allowing it to successfully complete the overtaking. Once the maneuver is completed and the curve ends, the ego vehicle accelerates back to its cruising speed of around 50 km/h. In terms of Cross Track Error, the results align well with expectations for a lane-change maneuver. Remarkably, even at speeds



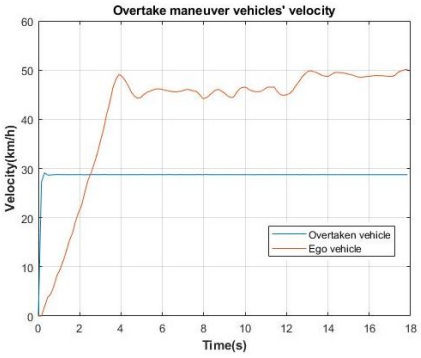
(a) *Vehicles path*



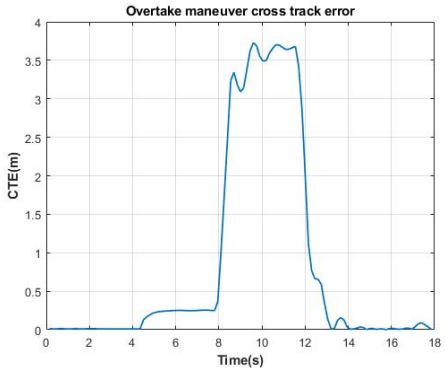
(b) *Ego vehicle trajectory in CARLA environment*

Figure 5.13: Overtaking maneuver

slightly higher than the design specifications, and despite the curve's geometry, the CTE remains very limited during the pre-overtaking phase, staying below 0.3 meters. While during the overtake it tracks correctly the center of the left lane at 3.5 m. This demonstrates the control system's ability to maintain precise lateral positioning lasting over time, even under challenging conditions such as higher speeds and road curvature.



(a) *Overtake vehicles velocity*



(b) *Ego-vehicle overtake cross track error*

Figure 5.14: Overtaking maneuver vehicles velocity and ego-vehicle CTE

This dynamic adjustment of the reference path, combined with the supervisory control function, ensures a smooth and controlled overtaking operation, demonstrating the system's adaptability to real-world driving scenarios and the capability of NMPC to perform well in a complex system.

Chapter 6

Conclusion

6.1 Thesis Results

This thesis focuses on the development of a control system based on an NMPC controller implemented in MATLAB/Simulink and co-simulated with the CARLA simulator. A variety of supporting functions were developed, including localization, path planning, and a dispatching function, all underpinned by a comprehensive system identification effort to enhance performance. Code to support variable velocity control based on path curvature was also developed. Significant improvements were made to the CARLA-MATLAB interface, particularly through the synchronization between Simulink and CARLA's world environment, ensuring seamless integration and real-time operation. The primary objective of this work was to demonstrate how an NMPC controller, integrated into an autonomous driving system, can improve path tracking, enhance vehicle stability, optimize the handling of input commands, and operate effectively across diverse scenarios. This was achieved through the implementation of additional decision-making functions, highlighting the potential of NMPC as a robust and flexible solution for advanced vehicle control systems.

During test simulation the developed control system demonstrates excellent performance across multiple aspects, confirming its robustness and versatility for autonomous driving applications. The system handles variable speeds effectively, it excels in speed management relative to curvature dynamically adjusting vehicle velocity based on road geometry to maintain stability and safety. In particular, the path tracking performance is outstanding, ensuring precise adherence to the generated trajectory and showcasing a better behavior comparable to that of Carla's built-in Autopilot. The vehicle also exhibits excellent stability during maneuvers, maintaining control and smooth behavior even under challenging conditions. Additionally, the input signals to the system are tunable by adjusting the

cost function weights and their smoothness outperformed the CARLA's autopilot, allowing flexible adaptation of the vehicle's dynamics to suit specific behaviors.

While the exact implementation of Carla's PID-based Autopilot remains unclear —whether it uses adaptive strategies, fixed logic or other approaches— it is undoubtedly tailored specifically to the known vehicle model, giving it an inherent advantage. In contrast, the proposed NMPC-based approach achieves better results despite operating with generalizable logic and simplified car model, demonstrating its effectiveness.

The modular structure of the system allows for future expansion and integration of additional features. Its tunable design with easily adjustable parameters and functions, offers the flexibility to modify the vehicle's behavior as desired. Furthermore, decision-making mechanisms can be seamlessly incorporated without requiring significant changes to the existing structure. This was demonstrating by the adaptability of the system to realistic scenario as obstacle avoidance and overtaking maneuver where the integration between the control logic and the NMPC controller handled the situations with good performances.

The system has been implemented and tested in real-time within the Carla simulation environment, achieving excellent results in terms of responsiveness and stability. Overall, the proposed control system proves to be a robust and effective solution for path tracking and trajectory generation with significant potential for future developments and enhancements in autonomous driving technologies.

6.2 Limitations of the Work

While the proposed control system has demonstrated excellent performance in various scenarios, several limitations should be acknowledged. The speed imposed by the controller, as it aims to reach the furthest point given within T_p, depends on the length of the prediction horizon and by the N_p variable that determine the most distant reference point given to the NMPC controller. This dependency to the prediction horizon could limit the system's adaptability to different path characteristics or driving conditions, speed should be a variable chosen depending on the situation and tracked by the controller. Additionally some of the implemented functions in Simulink blocks were specifically designed for this application. While effective, their ad hoc nature may reduce generalization and require rework or adjustments when applied to other systems or environments. Additionally, the system's performance relies heavily on specific parameter tuning, such as weights in the NMPC cost function, which were tailored to the particular scenario and vehicle model used in this study. This may necessitate significant effort for adaptation in other contexts.

Furthermore the exact vehicle model was not fully known, requiring approximations and empirical methods for system identification. While these methods achieved good results, they may not perfectly capture all aspects of the vehicle's dynamics, particularly in edge cases. Moreover, the vehicle's lateral dynamics were not thoroughly investigated in this work, which could impact the system's performance in scenarios where lateral stability and control are critical, such as sharp turns or high-speed maneuvers. Addressing these limitations in future work could further enhance the system's robustness, adaptability, and applicability to a broader range of vehicles and driving scenarios.

6.3 Future work

The work presented in this thesis lays a strong foundation for further advancements in autonomous vehicle control systems. Several potential directions for future development can significantly enhance the system's capabilities and versatility. Additional functionalities and driver assistance features, such as adaptive cruise control, could also be developed. These features would further extend the applicability of the system and improve its usability in real-world scenarios. The integration of cameras and additional sensors could provide richer environmental data, enabling the incorporation of computer vision techniques to assist with decision-making processes. This would allow the system to better handle dynamic environments and improve its overall situational awareness.

Regarding the control approach a key area for improvement could be the separation of lateral and longitudinal control, where independent controllers could be implemented. For instance, an NMPC could handle lateral control, while either another NMPC or an alternative approach could manage longitudinal dynamics. This separation would allow for more specialized tuning and better performance in diverse driving scenarios. The implementation of event-triggered NMPC could optimize computational efficiency by updating the control strategy only when significant changes in the environment or vehicle state occur. This would reduce computational load while maintaining high performance or other optimized solver for NMPC OCP can be developed. Similarly, introducing online adaptive parameter tuning could dynamically adjust the controller's parameters, such as weights in the cost function, initial velocity v_0, curvature weights, or other tuning variables, to improve robustness and provide greater flexibility to adapt the control strategy on-the-fly to different driving conditions.

Rewriting the entire control system in Python script would streamline the implementation process without needing co simulation between MATLAB and CARLA, enabling better integration and computational speed within CARLA environment. Additionally, directly extracting reference points, as waypoints

already present in the simulator, from Carla would enable the system to utilize real-time data for better trajectory planning and execution.

By exploring these avenues, the system could evolve into a more robust, adaptive, and feature-rich control framework, paving the way for advanced autonomous driving applications and contributing to the field's ongoing development.

Appendix A

Additional Functions

This script defines a function to compute the curvature of a line represented in a two-dimensional space used in the control system inside the localization function.

Listing A.1: Curvature function

```
function k=LineCurvature2D(Vertices,Lines,CD)
% This function calculates the curvature of a 2D line. It first fits
  from
% the polygons;
%
%  k = LineCurvature2D(Vertices,Lines)
%
% inputs,
%   Vertices : A M x 2 list of line points.
%   (optional)
%   Lines : A N x 2 list of line pieces, by indices of the vertices
%         (if not set assume Lines=[1 2; 2 3 ; ... ; M-1 M])
%
% outputs,
%   k : M x 1 Curvature values
%
%
%
% Example, Circle
%  r=sort(rand(15,1))*2*pi;
%  Vertices=[sin(r) cos(r)]*10;
%  Lines=[(1:size(Vertices,1))' (2:size(Vertices,1)+1)']; Lines(end
  ,2)=1;
%  k=LineCurvature2D(Vertices,Lines);
%
%  figure,  hold on;
%  N=LineNormals2D(Vertices,Lines);
%  k=k*100;
```

```
28 %  plot([Vertices(:,1) Vertices(:,1)+k.*N(:,1)]',[Vertices(:,2)
       Vertices(:,2)+k.*N(:,2)]','g');
29 %  plot([Vertices(Lines(:,1),1) Vertices(Lines(:,2),1)]',[Vertices(
       Lines(:,1),2) Vertices(Lines(:,2),2)]','b');
30 %  plot(sin(0:0.01:2*pi)*10,cos(0:0.01:2*pi)*10,'r.');
31 %  axis equal;
32 %
33 % Example, Hand
34 %  load('testdata');
35 %  k=LineCurvature2D(Vertices,Lines);
36 %
37 %  figure,  hold on;
38 %  N=LineNormals2D(Vertices,Lines);
39 %  k=k*100;
40 %  plot([Vertices(:,1) Vertices(:,1)+k.*N(:,1)]',[Vertices(:,2)
       Vertices(:,2)+k.*N(:,2)]','g');
41 %  plot([Vertices(Lines(:,1),1) Vertices(Lines(:,2),1)]',[Vertices(
       Lines(:,1),2) Vertices(Lines(:,2),2)]','b');
42 %  plot(Vertices(:,1),Vertices(:,2),'r.');
43 %  axis equal;
44 %
45 % Function is written by D.Kroon University of Twente (August 2011)
46
47 % If no line-indices, assume a x(1) connected with x(2), x(3) with x
       (4) ...
48 if(nargin<2)
49     Lines=[(1:(size(Vertices,1)-1))' (2:size(Vertices,1))'];
50 end
51
52 % Get left and right neighbor of each points
53 Na=zeros(size(Vertices,1),1); Nb=zeros(size(Vertices,1),1);
54 Na(Lines(:,1))=Lines(:,2); Nb(Lines(:,2))=Lines(:,1);
55
56 % Check for end of line points, without a left or right neighbor
57 checkNa=Na==0; checkNb=Nb==0;
58 Naa=Na; Nbb=Nb;
59 Naa(checkNa)=find(checkNa); Nbb(checkNb)=find(checkNb);
60
61 % If no left neighbor use two right neighbors, and the same for right
       ...
62 Na(checkNa)=Nbb(Nbb(checkNa)); Nb(checkNb)=Naa(Naa(checkNb));
63
64 % Correct for sampeling differences
65 Ta=-sqrt(sum((Vertices-Vertices(Na,:)).^2,2));
66 Tb=sqrt(sum((Vertices-Vertices(Nb,:)).^2,2));
67
68 % If no left neighbor use two right neighbors, and the same for right
       ...
69 Ta(checkNa)=-Ta(checkNa); Tb(checkNb)=-Tb(checkNb);
```

```
70
71 % Fit a polygons to the vertices
72 % x=a(3)*t^2 + a(2)*t + a(1)
73 % y=b(3)*t^2 + b(2)*t + b(1)
74 % we know the x,y of every vertice and set t=0 for the vertices, and
75 % t=Ta for left vertices, and t=Tb for right vertices,
76 x = [Vertices(Na,1) Vertices(:,1) Vertices(Nb,1)];
77 y = [Vertices(Na,2) Vertices(:,2) Vertices(Nb,2)];
78 M = [ones(size(Tb)) -Ta Ta.^2 ones(size(Tb)) zeros(size(Tb)) zeros(
       size(Tb)) ones(size(Tb)) -Tb Tb.^2];
79 invM=inverse3(M,CD);
80 a=zeros(CD,3);
81 b=zeros(CD,3);
82 a(:,1)=invM(:,1,1).*x(:,1)+invM(:,2,1).*x(:,2)+invM(:,3,1).*x(:,3);
83 a(:,2)=invM(:,1,2).*x(:,1)+invM(:,2,2).*x(:,2)+invM(:,3,2).*x(:,3);
84 a(:,3)=invM(:,1,3).*x(:,1)+invM(:,2,3).*x(:,2)+invM(:,3,3).*x(:,3);
85 b(:,1)=invM(:,1,1).*y(:,1)+invM(:,2,1).*y(:,2)+invM(:,3,1).*y(:,3);
86 b(:,2)=invM(:,1,2).*y(:,1)+invM(:,2,2).*y(:,2)+invM(:,3,2).*y(:,3);
87 b(:,3)=invM(:,1,3).*y(:,1)+invM(:,2,3).*y(:,2)+invM(:,3,3).*y(:,3);
88
89 % Calculate the curvature from the fitted polygon
90 k = 2*(a(:,2).*b(:,3)-a(:,3).*b(:,2)) ./ ((a(:,2).^2+b(:,2).^2)
       .^(3/2));
91 end
```

Bibliography

[1] Istvan Barabas, Adrian Todoruţ, N Cordoş, and Andreia Molea. «Current challenges in autonomous driving». In: *IOP conference series: materials science and engineering*. Vol. 252. 1. IOP Publishing. 2017, p. 012096 (cit. on p. 1).

[2] Claudine Badue et al. «Self-driving cars: A survey». In: *Expert systems with applications* 165 (2021), p. 113816 (cit. on p. 1).

[3] Lee Gomes. «Hidden obstacles for Google's self-driving cars: Impressive progress hides major limitations of Google's quest for automated driving». In: *Massachusetts Institute of Technology. As of March* 3 (2014), p. 2016 (cit. on p. 2).

[4] Betina Carol Zanchin, Rodrigo Adamshuk, Max Mauro Santos, and Kathya Silvia Collazos. «On the instrumentation and classification of autonomous cars». In: *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE. 2017, pp. 2631–2636 (cit. on p. 3).

[5] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. «A survey of autonomous driving: Common practices and emerging technologies». In: *IEEE access* 8 (2020), pp. 58443–58469 (cit. on p. 4).

[6] Rowan Thomas McAllister, Yarin Gal, Alex Kendall, Mark Van Der Wilk, Amar Shah, Roberto Cipolla, and Adrian Weller. «Concrete problems for autonomous vehicle safety: Advantages of Bayesian deep learning». In: International Joint Conferences on Artificial Intelligence, Inc. 2017 (cit. on p. 5).

[7] JP Laumond. *Robot Motion Planning and Control*. 1998 (cit. on p. 5).

[8] Ramesh Jain, Rangachar Kasturi, Brian G Schunck, et al. *Machine vision*. Vol. 5. McGraw-hill New York, 1995 (cit. on p. 5).

[9] Rajesh Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011 (cit. on pp. 5, 37).

[10] Sterling J Anderson, Sisir B Karumanchi, and Karl Iagnemma. «Constraint-based planning and control for safe, semi-autonomous operation of vehicles». In: *2012 IEEE intelligent vehicles symposium*. IEEE. 2012, pp. 383–388 (cit. on p. 5).

[11] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. «Deeptest: Automated testing of deep-neural-network-driven autonomous cars». In: *Proceedings of the 40th international conference on software engineering*. 2018, pp. 303–314 (cit. on p. 5).

[12] Mariusz Bojarski. «End to end learning for self-driving cars». In: *arXiv preprint arXiv:1604.07316* (2016) (cit. on p. 6).

[13] Huazhe Xu, Yang Gao, Fisher Yu, and Trevor Darrell. «End-to-end learning of driving models from large-scale video datasets». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2174–2182 (cit. on p. 6).

[14] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. «Deepdriving: Learning affordance for direct perception in autonomous driving». In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 2722–2730 (cit. on p. 6).

[15] Volodymyr Mnih et al. «Human-level control through deep reinforcement learning». In: *nature* 518.7540 (2015), pp. 529–533 (cit. on p. 6).

[16] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. «Deep reinforcement learning framework for autonomous driving». In: *arXiv preprint arXiv:1704.02532* (2017) (cit. on p. 6).

[17] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. «Learning to drive in a day». In: *2019 international conference on robotics and automation (ICRA)*. IEEE. 2019, pp. 8248–8254 (cit. on p. 6).

[18] Dario Floreano, Peter Dürr, and Claudio Mattiussi. «Neuroevolution: from architectures to learning». In: *Evolutionary intelligence* 1 (2008), pp. 47–62 (cit. on p. 6).

[19] Dean A Pomerleau. «Alvinn: An autonomous land vehicle in a neural network». In: *Advances in neural information processing systems* 1 (1988) (cit. on p. 7).

[20] Shumeet Baluja. «Evolution of an artificial neural network based autonomous land vehicle controller». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.3 (1996), pp. 450–463 (cit. on p. 7).

[21] Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. «Evolving large-scale neural networks for vision-based reinforcement learning». In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 2013, pp. 1061–1068 (cit. on p. 7).

[22] Lu Chi and Yadong Mu. «Deep steering: Learning end-to-end driving model from spatial and temporal visual cues». In: *arXiv preprint arXiv:1708.03798* (2017) (cit. on p. 7).

[23] Khadige Abboud, Hassan Aboubakr Omar, and Weihua Zhuang. «Interworking of DSRC and cellular network technologies for V2X communications: A survey». In: *IEEE transactions on vehicular technology* 65.12 (2016), pp. 9457–9470 (cit. on p. 7).

[24] Jian Wang, Yameng Shao, Yuming Ge, and Rundong Yu. «A survey of vehicle to everything (V2X) testing». In: *Sensors* 19.2 (2019), p. 334 (cit. on p. 7).

[25] Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. «Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds». In: *2014 IEEE world forum on internet of things (WF-IoT)*. IEEE. 2014, pp. 241–246 (cit. on p. 8).

[26] Marica Amadeo, Claudia Campolo, and Antonella Molinaro. «Information-centric networking for connected vehicles: a survey and future perspectives». In: *IEEE Communications Magazine* 54.2 (2016), pp. 98–104 (cit. on p. 8).

[27] Eun-Kyu Lee, Mario Gerla, Giovanni Pau, Uichin Lee, and Jae-Han Lim. «Internet of Vehicles: From intelligent grid to autonomous cars and vehicular fogs». In: *International Journal of Distributed Sensor Networks* 12.9 (2016), p. 1550147716665500 (cit. on p. 8).

[28] Swarun Kumar, Lixin Shi, Nabeel Ahmed, Stephanie Gil, Dina Katabi, and Daniela Rus. «Carspeak: a content-centric network for autonomous driving». In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 259–270 (cit. on p. 8).

[29] Ekim Yurtsever, Suguru Yamazaki, Chiyomi Miyajima, Kazuya Takeda, Masataka Mori, Kentarou Hitomi, and Masumi Egawa. «Integrating driving behavior and traffic context through signal symbolization for data reduction and risky lane change detection». In: *IEEE Transactions on Intelligent Vehicles* 3.3 (2018), pp. 242–253 (cit. on p. 8).

[30] Mario Gerla. «Vehicular cloud computing». In: *2012 The 11th annual mediterranean ad hoc networking workshop (Med-Hoc-Net)*. IEEE. 2012, pp. 152–155 (cit. on p. 8).

[31] Md Whaiduzzaman, Mehdi Sookhak, Abdullah Gani, and Rajkumar Buyya. «A survey on vehicular cloud computing». In: *Journal of Network and Computer applications* 40 (2014), pp. 325–344 (cit. on p. 8).

[32] Ikram Ud Din, Byung-Seo Kim, Suhaidi Hassan, Mohsen Guizani, Mohammed Atiquzzaman, and Joel JPC Rodrigues. «Information-centric network-based vehicular communications: Overview and research opportunities». In: *Sensors* 18.11 (2018), p. 3957 (cit. on p. 8).

[33] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. «Airsim: High-fidelity visual and physical simulation for autonomous vehicles». In: *Field and Service Robotics: Results of the 11th International Conference*. Springer. 2018, pp. 621–635 (cit. on p. 9).

[34] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. «A survey of autonomous driving: Common practices and emerging technologies». In: *IEEE access* 8 (2020), pp. 58443–58469 (cit. on p. 9).

[35] Ludovic Pintard, Jean-Charles Fabre, Karama Kanoun, Michel Leeman, and Matthieu Roy. «Fault injection in the automotive standard ISO 26262: An initial approach». In: *European Workshop on Dependable Computing*. Springer. 2013, pp. 126–133 (cit. on p. 10).

[36] Xi Zhang and Chris Mi. *Vehicle power management: modeling, control and optimization*. Springer Science & Business Media, 2011 (cit. on p. 10).

[37] Punit Tulpule, Ayyoub Rezaeian, Aditya Karumanchi, and Shawn Midlam-Mohler. «Model based design (MBD) and hardware in the loop (HIL) validation: curriculum development». In: *2017 American Control Conference (ACC)*. IEEE. 2017, pp. 5361–5366 (cit. on p. 11).

[38] Hans-Peter Schöner. «Challenges and approaches for testing of highly automated vehicles». In: *Energy Consumption and Autonomous Driving: Proceedings of the 3rd CESA Automotive Electronics Congress, Paris, 2014*. Springer. 2016, pp. 101–109 (cit. on p. 11).

[39] Hans-Peter Schöner. «Simulation in development and testing of autonomous vehicles». In: *18. Internationales Stuttgarter Symposium: Automobil-und Motorentechnik*. Springer. 2018, pp. 1083–1095 (cit. on p. 12).

[40] Prabhjot Kaur, Samira Taghavi, Zhaofeng Tian, and Weisong Shi. «A survey on simulators for testing self-driving cars». In: *2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD)*. IEEE. 2021, pp. 62–70 (cit. on p. 12).

[41] Antonín Šmíd. «Comparison of unity and unreal engine». In: *Czech Technical University in Prague* (2017), pp. 41–61 (cit. on p. 12).

[42] Jessica Van Brummelen, Marie O'brien, Dominique Gruyer, and Homayoun Najjaran. «Autonomous vehicle perception: The technology of today and tomorrow». In: *Transportation research part C: emerging technologies* 89 (2018), pp. 384–406 (cit. on p. 12).

[43] Jelena Kocić, Nenad Jovičić, and Vujo Drndarević. «Sensors and sensor fusion in autonomous vehicles». In: *2018 26th Telecommunications Forum (TELFOR)*. IEEE. 2018, pp. 420–425 (cit. on p. 12).

[44] Yong K Hwang and Narendra Ahuja. «Gross motion planning—a survey». In: *ACM Computing Surveys (CSUR)* 24.3 (1992), pp. 219–291 (cit. on p. 13).

[45] David González, Joshué Pérez, Vicente Milanés, and Fawzi Nashashibi. «A review of motion planning techniques for automated vehicles». In: *IEEE Transactions on intelligent transportation systems* 17.4 (2015), pp. 1135–1145 (cit. on p. 13).

[46] Rui Fan, Jianhao Jiao, Haoyang Ye, Yang Yu, Ioannis Pitas, and Ming Liu. «Key ingredients of self-driving cars». In: *arXiv preprint arXiv:1906.02939* (2019) (cit. on p. 13).

[47] John-Jairo Martinez and Carlos Canudas-de-Wit. «A safe longitudinal control for adaptive cruise control and stop-and-go scenarios». In: *IEEE Transactions on control systems technology* 15.2 (2007), pp. 246–258 (cit. on p. 13).

[48] Shengbo Li, Keqiang Li, Rajesh Rajamani, and Jianqiang Wang. «Model predictive multi-objective vehicular adaptive cruise control». In: *IEEE Transactions on control systems technology* 19.3 (2010), pp. 556–566 (cit. on p. 13).

[49] Fanta Camara et al. «Pedestrian models for autonomous driving part ii: high-level models of human behavior». In: *IEEE Transactions on Intelligent Transportation Systems* 22.9 (2020), pp. 5453–5472 (cit. on p. 13).

[50] Joshua Fadaie. «The state of modeling, simulation, and data utilization within industry: An autonomous vehicles perspective». In: *arXiv preprint arXiv:1910.06075* (2019) (cit. on p. 13).

[51] Sergio Lafuente-Arroyo, Pedro Gil-Jimenez, R Maldonado-Bascon, Francisco López-Ferreras, and Saturnino Maldonado-Bascón. «Traffic sign shape classification evaluation I: SVM using distance to borders». In: *IEEE Proceedings. Intelligent Vehicles Symposium, 2005.* IEEE. 2005, pp. 557–562 (cit. on p. 13).

[52] Yuyan Liu, Miles Tight, Quanxin Sun, and Ruiyu Kang. «A systematic review: Road infrastructure requirement for Connected and Autonomous Vehicles (CAVs)». In: *Journal of Physics: Conference Series*. Vol. 1187. 4. IOP Publishing. 2019, p. 042073 (cit. on p. 13).

[53] Shinpei Kato et al. «Autoware on board: Enabling autonomous vehicles with embedded systems». In: *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE. 2018, pp. 287–296 (cit. on p. 14).

[54] Haoyang Fan et al. «Baidu apollo em motion planner». In: *arXiv preprint arXiv:1807.08048* (2018) (cit. on p. 14).

[55] Ivo Miguel Menezes Silva, Hélder David Malheiro Silva, Fabricio Botelho, and Cristiano Gonçalves Pendão. «Realistic 3D simulators for automotive: a review of main applications and features». In: (2024) (cit. on p. 15).

[56] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. «CARLA: An open urban driving simulator». In: *Conference on robot learning*. PMLR. 2017, pp. 1–16 (cit. on p. 15).

[57] Andrew Sanders. *An introduction to Unreal engine 4*. AK Peters/CRC Press, 2016 (cit. on p. 15).

[58] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. «SUMO–simulation of urban mobility: an overview». In: *Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation*. ThinkMind. 2011 (cit. on p. 16).

[59] Andreas Junghanns et al. «The functional mock-up interface 3.0-new features enabling new applications». In: *Modelica conferences*. 2021, pp. 17–26 (cit. on p. 17).

[60] Guodong Rong et al. «Lgsvl simulator: A high fidelity simulator for autonomous driving». In: *2020 IEEE 23rd International conference on intelligent transportation systems (ITSC)*. IEEE. 2020, pp. 1–6 (cit. on p. 18).

[61] Matthias Müller, Vincent Casser, Jean Lahoud, Neil Smith, and Bernard Ghanem. «Sim4cv: A photo-realistic simulator for computer vision applications». In: *International Journal of Computer Vision* 126 (2018), pp. 902–919 (cit. on p. 19).

[62] Francesco Micheli, Mattia Bersani, Stefano Arrigoni, Francesco Braghin, and Federico Cheli. «NMPC trajectory planner for urban autonomous driving». In: *Vehicle system dynamics* 61.5 (2023), pp. 1387–1409 (cit. on p. 22).

[63] Jean Pierre Allamaa, Petr Listov, Herman Van der Auweraer, Colin Jones, and Tong Duy Son. «Real-time nonlinear mpc strategy with full vehicle validation for autonomous driving». In: *2022 American Control Conference (ACC)*. IEEE. 2022, pp. 1982–1987 (cit. on p. 22).

[64] Wael Farag. «Real-time NMPC path tracker for autonomous vehicles». In: *Asian Journal of Control* 23.4 (2021), pp. 1952–1965 (cit. on p. 22).

[65] Niels van Duijkeren, Tamas Keviczky, Peter Nilsson, and Leo Laine. «Real-time NMPC for semi-automated highway driving of long heavy vehicle combinations». In: *IFAC-PapersOnLine* 48.23 (2015), pp. 39–46 (cit. on p. 22).

[66] Wael Farag. «Complex track maneuvering using real-time MPC control for autonomous driving». In: *International Journal of Computing and Digital Systems* 9.5 (2020), pp. 909–920 (cit. on p. 22).

[67] Maryam Nezami, Dimitrios S Karachalios, Georg Schildbach, and Hossam S Abbas. «On the design of nonlinear MPC and LPVMPC for obstacle avoidance in autonomous driving». In: *2023 9th International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE. 2023, pp. 1–6 (cit. on p. 22).

[68] Mostafa Emam and Matthias Gerdts. «Deterministic Operating Strategy for Multi-objective NMPC for Safe Autonomous Driving in Urban Traffic.» In: *VEHITS*. 2022, pp. 152–161 (cit. on p. 22).

[69] Karl Berntorp, Rien Quirynen, Tomoki Uno, and Stefano Di Cairano. «Trajectory tracking for autonomous vehicles on varying road surfaces by friction-adaptive nonlinear model predictive control». In: *Vehicle System Dynamics* 58.5 (2020), pp. 705–725 (cit. on pp. 22, 29, 30).

[70] Stefano Arrigoni, Francesco Braghin, and Federico Cheli. «MPC trajectory planner for autonomous driving solved by genetic algorithm technique». In: *Vehicle system dynamics* 60.12 (2022), pp. 4118–4143 (cit. on p. 22).

[71] Mattia Boggio, Carlo Novara, and Michele Taragna. «Trajectory planning and control for autonomous vehicles: a "fast" data-aided NMPC approach». In: *European Journal of Control* 74 (2023), p. 100857 (cit. on p. 22).

[72] Mattia Boggio, Carlo Novara, and Michele Taragna. «Nonlinear model predictive control: An optimal search domain reduction». In: *IFAC-PapersOnLine* 56.2 (2023), pp. 6253–6258 (cit. on pp. 22, 43, 44).

[73] Trieu Minh Vu, Reza Moezzi, Jindrich Cyrus, and Jaroslav Hlava. «Model predictive control for autonomous driving vehicles». In: *Electronics* 10.21 (2021), p. 2593 (cit. on p. 22).

[74] Muhammad Awais Abbas, Ruth Milman, and J Mikael Eklund. «Obstacle avoidance in real time with nonlinear model predictive control of autonomous vehicles». In: *Canadian journal of electrical and computer engineering* 40.1 (2017), pp. 12–22 (cit. on p. 22).

[75] Li Zhai, Chengping Wang, Yuhan Hou, and Chang Liu. «MPC-based integrated control of trajectory tracking and handling stability for intelligent driving vehicle driven by four hub motor». In: *IEEE transactions on vehicular technology* 71.3 (2022), pp. 2668–2680 (cit. on p. 22).

[76] Mooryong Choi and Seibum B Choi. «MPC for vehicle lateral stability via differential braking and active front steering considering practical aspects». In: *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering* 230.4 (2016), pp. 459–469 (cit. on p. 22).

[77] Mooryong Choi and Seibum B Choi. «Model predictive control for vehicle yaw stability with practical concerns». In: *IEEE transactions on vehicular technology* 63.8 (2014), pp. 3539–3548 (cit. on p. 22).

[78] John M Guirguis, Sherif Hammad, and Shady A Maged. «Path tracking control based on an adaptive MPC to changing vehicle dynamics». In: *International Journal of Mechanical Engineering and Robotics Research* 11.7 (2022), pp. 535–541 (cit. on p. 22).

[79] Haidong Wu, Zhenli Si, and Zihan Li. «Trajectory tracking control for four-wheel independent drive intelligent vehicle based on model predictive control». In: *IEEE Access* 8 (2020), pp. 73071–73081 (cit. on p. 22).

[80] Hengyang Wang, Biao Liu, Xianyao Ping, and Quan An. «Path tracking control for autonomous vehicles based on an improved MPC». In: *IEEE access* 7 (2019), pp. 161064–161073 (cit. on p. 22).

[81] Ying Xu, Wentao Tang, Biyun Chen, Li Qiu, and Rong Yang. «A model predictive control with preview-follower theory algorithm for trajectory tracking control in autonomous vehicles». In: *Symmetry* 13.3 (2021), p. 381 (cit. on p. 22).

[82] Chulho Choi and Yeonsik Kang. «Simultaneous braking and steering control method based on nonlinear model predictive control for emergency driving support». In: *International Journal of Control, Automation and Systems* 15.1 (2017), pp. 345–353 (cit. on p. 23).

[83] Mert Batmaz. «Development of Complex Scenarios and Control Algorithms for Autonomous Driving Functions (ADFs) in a Driving Simulator». PhD thesis. Politecnico di Torino, 2024 (cit. on p. 23).

[84] Daniel R Morais and A Pedro Aguiar. «Model Predictive Control for Self Driving Cars: A Case Study Using the Simulator CARLA within a ROS Framework». In: *2022 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. IEEE. 2022, pp. 124–129 (cit. on p. 23).

[85] Zhaodong Zhou, Christopher Rother, and Jun Chen. «Event-triggered model predictive control for autonomous vehicle path tracking: Validation using CARLA simulator». In: *IEEE Transactions on Intelligent Vehicles* 8.6 (2023), pp. 3547–3555 (cit. on p. 23).

[86] Siddharth H Nair, Vijay Govindarajan, Theresa Lin, Chris Meissen, H Eric Tseng, and Francesco Borrelli. «Stochastic mpc with multi-modal predictions for traffic intersections». In: *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2022, pp. 635–640 (cit. on p. 23).

[87] Alexander L Gratzer, Maximilian M Broger, Alexander Schirrer, and Stefan Jakubek. «Two-Layer MPC Architecture for Efficient Mixed-Integer-Informed Obstacle Avoidance in Real-Time». In: *IEEE Transactions on Intelligent Transportation Systems* (2024) (cit. on p. 23).

[88] Farhad Partovi Ebrahimpour and Hasan Ferdowsi. «Multi-Constraint predictive control system with auxiliary emergency controllers for autonomous vehicles». In: *2021 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2021, pp. 274–279 (cit. on p. 24).

[89] HT Madan, P Ramya, M Rakshitha, et al. «Trajectory Tracking and Lane-Keeping Assistance for Autonomous Systems Using Pid and MPC Controllers». In: *2023 International Conference on Smart Systems for applications in Electrical Sciences (ICSSES)*. IEEE. 2023, pp. 1–7 (cit. on p. 24).

[90] Minsung Kim, Donggil Lee, Joonwoo Ahn, Minsoo Kim, and Jaeheung Park. «Model predictive control method for autonomous vehicles using time-varying and non-uniformly spaced horizon». In: *IEEE Access* 9 (2021), pp. 86475–86487 (cit. on p. 24).

[91] Karl Berntorp, Björn Olofsson, Kristoffer Lundahl, and Lars Nielsen. «Models and methodology for optimal trajectory generation in safety-critical road–vehicle manoeuvres». In: *Vehicle System Dynamics* 52.10 (2014), pp. 1304–1332 (cit. on p. 28).

[92] Rien Quirynen, Karl Berntorp, and Stefano Di Cairano. «Embedded optimization algorithms for steering in autonomous vehicles based on nonlinear model predictive control». In: *2018 Annual American Control Conference (ACC)*. IEEE. 2018, pp. 3251–3256 (cit. on p. 28).

[93] Stefano Di Cairano and Ilya V Kolmanovsky. «Real-time optimization and model predictive control for aerospace and automotive applications». In: *2018 annual American control conference (ACC)*. IEEE. 2018, pp. 2392–2409 (cit. on p. 29).

[94] Jeong hwan Jeon, Raghvendra V Cowlagi, Steven C Peters, Sertac Karaman, Emilio Frazzoli, Panagiotis Tsiotras, and Karl Iagnemma. «Optimal motion planning with the half-car dynamical model for autonomous high-speed driving». In: *2013 American control conference*. IEEE. 2013, pp. 188–193 (cit. on p. 29).

[95] Sherif Nekkah et al. «The autonomous racing software stack of the KIT19d». In: *arXiv preprint arXiv:2010.02828* (2020) (cit. on p. 31).

[96] Eugenio Tramacere, Sara Luciani, Stefano Feraco, Angelo Bonfitto, and Nicola Amati. «Processor-in-the-loop architecture design and experimental validation for an autonomous racing vehicle». In: *Applied Sciences* 11.16 (2021), p. 7225 (cit. on p. 32).

[97] Massimo Guiggiani et al. «The science of vehicle dynamics». In: *Pisa, Italy: Springer Netherlands* 15 (2014) (cit. on p. 32).

[98] Danilo Caporale, Alessandro Settimi, Federico Massa, Francesco Amerotti, Andrea Corti, Adriano Fagiolini, Massimo Guiggiani, Antonio Bicchi, and Lucia Pallottino. «Towards the design of robotic drivers for full-scale self-driving racing cars». In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 5643–5649 (cit. on p. 32).

[99] Fabian Christ, Alexander Wischnewski, Alexander Heilmeier, and Boris Lohmann. «Time-optimal trajectory planning for a race car considering variable tyre-road friction coefficients». In: *Vehicle system dynamics* 59.4 (2021), pp. 588–612 (cit. on p. 32).

[100] Robin Verschueren, Mario Zanon, Rien Quirynen, and Moritz Diehl. «Time-optimal race car driving using an online exact hessian based nonlinear MPC algorithm». In: *2016 European control conference (ECC)*. IEEE. 2016, pp. 141–147 (cit. on p. 32).

[101] Tantan Zhang, Yueshuo Sun, Yazhou Wang, Bai Li, Yonglin Tian, and Fei-Yue Wang. «A survey of vehicle dynamics modeling methods for autonomous racing: Theoretical models, physical/virtual platforms, and perspectives». In: *IEEE Transactions on Intelligent Vehicles* (2024) (cit. on p. 32).

[102] Ademir AC Júnior, Sanjay Misra, and Michel S Soares. «A systematic mapping study on software architectures description based on ISO/IEC/IEEE 42010: 2011». In: *Computational Science and Its Applications–ICCSA 2019: 19th International Conference, Saint Petersburg, Russia, July 1–4, 2019, Proceedings, Part V 19*. Springer. 2019, pp. 17–30 (cit. on p. 33).

[103] Sagar Behere and Martin Törngren. «A functional architecture for autonomous driving». In: *Proceedings of the first international workshop on automotive software architecture*. 2015, pp. 3–10 (cit. on p. 33).

[104] Omveer Sharma, Nirod C Sahoo, and Niladri B Puhan. «Recent advances in motion and behavior planning techniques for software architecture of autonomous vehicles: A state-of-the-art survey». In: *Engineering applications of artificial intelligence* 101 (2021), p. 104211 (cit. on p. 34).

[105] Max Schwenzer, Muzaffer Ay, Thomas Bergs, and Dirk Abel. «Review on model predictive control: An engineering perspective». In: *The International Journal of Advanced Manufacturing Technology* 117.5 (2021), pp. 1327–1349 (cit. on pp. 35, 38).

[106] James B Rawlings. «Tutorial overview of model predictive control». In: *IEEE control systems magazine* 20.3 (2000), pp. 38–52 (cit. on pp. 35, 38).

[107] Gunnar Hillerström and Kirthi Walgama. «Repetitive control theory and applications-a survey». In: *IFAC Proceedings Volumes* 29.1 (1996), pp. 1446–1451 (cit. on p. 35).

[108] Em Poh Ping, Khisbullah Hudha, and Hishamuddin Jamaluddin. «Hardware-in-the-loop simulation of automatic steering control for lanekeeping manoeuvre: outer-loop and inner-loop control design». In: *International journal of vehicle safety* 5.1 (2010), pp. 35–59 (cit. on p. 36).

[109] Ehab I Al Khatib, Wasim MF Al-Masri, Shayok Mukhopadhyay, Mohammad A Jaradat, and Mamoun Abdel-Hafez. «A comparison of adaptive trajectory tracking controllers for wheeled mobile robots». In: *2015 10th international symposium on mechatronics and its applications (ISMA)*. IEEE. 2015, pp. 1–6 (cit. on p. 36).

[110] Alan SI Zinober. «Deterministic control of uncertain systems». In: *Proceedings. ICCON IEEE International Conference on Control and Applications*. IEEE. 1989, pp. 645–650 (cit. on p. 36).

[111] Jinkun Liu. *Sliding mode control using MATLAB*. Academic Press, 2017 (cit. on p. 36).

[112] Jessy W Grizzle, Christine Chevallereau, Aaron D Ames, and Ryan W Sinnet. «3D bipedal robotic walking: models, feedback control, and open problems». In: *IFAC Proceedings Volumes* 43.14 (2010), pp. 505–532 (cit. on p. 36).

[113] Jong-Min Yang and Jong-Hwan Kim. «Sliding mode control for trajectory tracking of nonholonomic wheeled mobile robots». In: *IEEE Transactions on robotics and automation* 15.3 (1999), pp. 578–587 (cit. on p. 36).

[114] Tomás Carricajo Martin, Marcos E Orchard, and Paul Vallejos Sánchez. «Design and simulation of control strategies for trajectory tracking in an autonomous ground vehicle». In: *IFAC Proceedings Volumes* 46.24 (2013), pp. 118–123 (cit. on p. 36).

[115] Himajit Aithal and S Janardhanan. «Trajectory tracking of two wheeled mobile robot using higher order sliding mode control». In: *2013 International Conference on Control, Computing, Communication and Materials (ICCCCM)*. IEEE. 2013, pp. 1–4 (cit. on p. 36).

[116] Ashish Tewari. «MODER CONTROL DESIGN». In: (2002) (cit. on p. 37).

[117] Noor Hafizah Amer, Hairi Zamzuri, Khisbullah Hudha, and Zulkiffli Abdul Kadir. «Modelling and control strategies in path tracking control for autonomous ground vehicles: A review of state of the art and challenges». In: *Journal of intelligent & robotic systems* 86 (2017), pp. 225–254 (cit. on p. 37).

[118] Kwang S Lee, In-Shik Chin, Hyuk J Lee, and Jay H Lee. «Model predictive control technique combined with iterative learning for batch processes». In: *AIChE Journal* 45.10 (1999), pp. 2175–2187 (cit. on p. 38).

[119] Alan SI Zinober and David H Owens. *Nonlinear and adaptive control: NCN4 2001*. Vol. 281. Springer Science & Business Media, 2002 (cit. on p. 38).

[120] Carlos E Garcia and Manfred Morari. «Internal model control. A unifying review and some new results». In: *Industrial & Engineering Chemistry Process Design and Development* 21.2 (1982), pp. 308–323 (cit. on p. 38).

[121] Jacques Richalet. «Industrial applications of model based predictive control». In: *Automatica* 29.5 (1993), pp. 1251–1274 (cit. on p. 38).

[122] Liuping Wang et al. *Model predictive control system design and implementation using MATLAB*. Vol. 3. Springer, 2009 (cit. on p. 38).

[123] Giuseppe Franze, Massimiliano Mattei, Luciano Ollio, and Valerio Scordamaglia. «A robust constrained model predictive control scheme for norm-bounded uncertain systems with partial state measurements». In: *International Journal of Robust and Nonlinear Control* 29.17 (2019), pp. 6105–6125 (cit. on p. 39).

[124] David Q Mayne, James B Rawlings, Christopher V Rao, and Pierre OM Scokaert. «Constrained model predictive control: Stability and optimality». In: *Automatica* 36.6 (2000), pp. 789–814 (cit. on p. 39).

[125] S Joe Qin and Thomas A Badgwell. «An overview of nonlinear model predictive control applications». In: *Nonlinear model predictive control* (2000), pp. 369–392 (cit. on p. 39).

[126] Jacques Richalet, André Rault, JL Testud, and J Papon. «Model predictive heuristic control». In: *Automatica (journal of IFAC)* 14.5 (1978), pp. 413–428 (cit. on p. 39).

[127] Carlos E Garcia, David M Prett, and Manfred Morari. «Model predictive control: Theory and practice—A survey». In: *Automatica* 25.3 (1989), pp. 335–348 (cit. on p. 39).

[128] Frank Allgöwer and Alex Zheng. *Nonlinear model predictive control*. Vol. 26. Birkhäuser, 2012 (cit. on p. 39).

[129] David Q Mayne, Saša V Raković, Rolf Findeisen, and Frank Allgöwer. «Robust output feedback model predictive control of constrained linear systems». In: *Automatica* 42.7 (2006), pp. 1217–1222 (cit. on p. 39).

[130] Moritz Diehl, Hans Georg Bock, Holger Diedam, and P-B Wieber. «Fast direct multiple shooting algorithms for optimal robot control». In: *Fast motions in biomechanics and robotics: optimization and feedback control* (2006), pp. 65–93 (cit. on p. 39).

[131] Mattia Boggio, Luigi Colangelo, Mario Virdis, Michele Pagone, and Carlo Novara. «Earth gravity in-orbit sensing: MPC formation control based on a novel constellation model». In: *Remote Sensing* 14.12 (2022), p. 2815 (cit. on p. 39).

[132] KS Holkar and LM Waghmare. «Discrete model predictive control for dc drive using orthonormal basis function». In: (2010) (cit. on p. 40).

[133] Swati Mohanty. «Artificial neural network based system identification and model predictive control of a flotation column». In: *Journal of Process Control* 19.6 (2009), pp. 991–999 (cit. on p. 40).

[134] KS Holkar and Laxman M Waghmare. «An overview of model predictive control». In: *International Journal of control and automation* 3.4 (2010), pp. 47–63 (cit. on pp. 40, 44).

[135] Rolf Findeisen and Frank Allgöwer. «An introduction to nonlinear model predictive control». In: *21st Benelux meeting on systems and control*. Vol. 11. Veldhoven. 2002, pp. 119–141 (cit. on p. 44).