POLITECNICO DI TORINO

Master's Degree in Automotive Engineering

Master's Degree Thesis

Development of a NMPC System for Autonomous Vehicles: Integration into the CARLA Simulation Environment and Validation in Complex Scenarios

Supervisors

Prof. Carlo NOVARA

PhD. Mattia BOGGIO

Eng. Fabio TANGO

Candidate

Alfonso Maria DIPALO

July 2025

Summary

One of the most groundbreaking innovations of our time is the development of autonomous vehicle (AVs) technology, which has the potential to significantly improve traffic flow, increase road safety, and support environmental sustainability. However, achieving fully autonomous driving remains a formidable challenge due to the need for advanced control systems capable of managing highly dynamic, nonlinear, and uncertain driving environments. Indeed, AVs must navigate complex traffic scenarios, make real-time decisions, and satisfy strict safety and performance constraints.
In this context, the present thesis focuses on the design and implementation of a control framework based on Nonlinear Model Predictive Control (NMPC) for autonomous driving applications. NMPC is particularly well-suited to address the challenges associated with AVs control, as it enables real-time trajectory optimization and complex decision-making while explicitly accounting for nonlinear vehicle dynamics and input/state constraints. The proposed controller is validated through a comprehensive co-simulation environment that integrates MATLAB/Simulink with the high-fidelity CARLA simulator. This modular and scalable simulation platform simplifies the replication of realistic traffic condition and the comprehensive evaluation of autonomous driving algorithms. Key contributions include the development of essential functionalities within the control system, such as vehicle output data analysis, optimal trajectory generation based on velocity and path curvature, and input signal manipulation to govern vehicle behaviour. A dispatching function is introduced to convert the desired acceleration commands produced by the NMPC into actuator inputs, specifically throttle and braking signals, thereby enhancing practical applicability of the system. This conversion exploits a simplified longitudinal vehicle dynamics model coupled with a simplified gear-shifting model to accurately capture the behaviour of the vehicle under varying operating conditions.
The NMPC controller is extensively tuned to balance tracking accuracy, robustness, and computational efficiency, ensuring real-time feasibility. The interface between MATLAB and CARLA is significantly improved to provide a reliable and efficient co-simulation environment. A comprehensive set of challenging driving scenarios is investigated, including 90° turns, overtaking, and suburban junctions with curves within CARLA simulation environment. Additionally, highly complex maneuvers such as extra-urban merging, urban merging with stop conditions, and multi-vehicle roundabout merging are implemented in Matlab/Simulink, where such scenarios are scarcely found in the literature.
Simulation results demonstrate the robustness and adaptability of the proposed NMPC framework across diverse and challenging scenarios. Performance metrics highlight effective trajectory optimization, precise vehicle control, and real-time decision-making capabilities, even in complex traffic interactions. Additionally, the development of modular and reusable Simulink blocks enhances the flexibility and scalability of the control architecture, supporting future research and development in AVs control.

Keywords: Autonomous vehicles, Control system, Path tracking, Decision making, NMPC (Nonlinear Model Predictive Control), MATLAB, CARLA, Trajectory planning, Input, Output, Lane merging, Roundabout

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my parents for their unconditional love, support, and encouragement throughout my academic journey. Their belief in me has been a constant source of motivation.

I am also sincerely thankful to my relatives and friends who have always been there to offer help, advice, and moral support during challenging times.

A special thanks goes to my thesis supervisors Carlo Novara, Mattia Boggio and Fabio Tango, for their invaluable guidance, expertise, and patience. Their mentorship has been essential to the successful completion of this work.

Finally, I would like to thank everyone who contributed, directly or indirectly, to this thesis and to my personal and professional growth.

Table of Contents

List of Tables

List of Figures

Acronyms

ABS
: Anti-lock Braking System

ACC
: Adaptive Cruise Control

ADAS
: Advanced Driving Assistance System

AEBS
: Automatic Emergency Braking System

AFS
: Adaptive Front-lighting System

AI
: Artificial Intelligence

ALC
: Adaptive Light Control

API
: Application Programming Interface

APS
: Automatic Parking System

ALSM
: Agent Lifecycle and State Management

AV
: Autonomous Vehicle

BSW
: Blind Spot Warning

CARLA
: Car Learning to Act

CC	Cruise Control
CM4SL	CarMaker for Simulink
CNN	Convolutional Neural Network
CoG	Center of Gravity
CoM	Center of Mass
DRSC	Direct Short Range Communication
C-V2X	Cellular Vehicle-to-Everything
DIL	Driver-in-the-Loop
DOF	Degree Of Freedom
DSTM	Dynamic Single Track Model
ECU	Electronic Control Unit
ESP	Electronic Stability Program
FPGA	Field Programmable Gate Array
GB	GearBox
GLONASS	GLObal'naja NAvigacionnaja Sputnikovaja Sistema (Russian)
GNSS	Global Navigation Satellite System

GPS

Global Positioning System

GUI

Graphical User Interface

HDL

Hardware Description Language

HIL

Hardware-in-the-Loop

HMI

Human-Machine Interface

HP

Horse Power

ICE

Internal Combustion Engine

ICEV

Internal Combustion Engine based Vehicle

IMU

Inertial Measurement Unit

LC

Lane Centering

LGSVL

LG Silicon Valley Lab

LK

Lane Keeping

LLC

Logical Link Control

LPF

Low Pass Filter

LQR

Linear Quadratic Regulator

MAC

Medium Access Control

MIL	Model-in-the-Loop
NLP	Non-Linear Problem
NMPC	Non-linear Model Predictive Control
NPC	Non Playable Character
NVS	Night Vision System
OCP	Optimal Control Problem
ODD	Operational Design Domain
PBVT	Path Buffer & Vehicle Tracking
PID	Proportional-Integrative-Derivative
PIL	Processor-in-the-Loop
RGB	Red Green Blue
ROS	Robot Operating System
RPC	Remote Procedure Call
SAE	Society of Automotive Engineers
SIL	Software-in-the-Loop
SIM4CV	Simulation for Computer Vision

SLAM

Simultaneous Localization And Mapping

SQP

Sequential Quadratic Programming

STM

Single Track Model

SUMO

Simulation of Urban MObility

TJA

Traffic Jam Assist

TM

Traffic Manager

TPMS

Tire Pressure Monitoring System

UAV

Unmanned Aerial Vehicle

UE

Unreal Engine

V2I

Vehicle-to-Infrastructure

V2N

Vehicle-to-Network

V2P

Vehicle-to-Pedestrian

V2X

Vehicle-to-Everything

WAVE

Wireless Access in Vehicular Environment

WSMP

WAVE Short Message Protocol

Chapter 1

Introduction

1.1 Autonomous Driving

During the past decade, the development of autonomous driving technology has progressed rapidly, bringing us closer to a future where vehicles are capable of navigating without human intervention. Autonomous vehicles (AVs), often referred to as self-driving cars, combine sensors, software, and artificial intelligence to perceive their surroundings and make driving decisions. These technologies aim to increase road safety, reduce traffic congestion, and provide greater mobility to people who cannot drive.
An autonomous vehicle is a car that can operate without direct input from a human driver. Unlike traditional vehicles, which are entirely controlled by humans, AVs use a lot of technologies, including LiDAR, radar, GPS, cameras, and computers onboard, to analyze the environment in real time. This allows the vehicle to detect road signs, avoid obstacles, follow traffic laws, and make complex decisions such as merging onto a highway or navigating through intersections.

1.2 Challenges of Autonomous Driving

Autonomous driving is one of the most exciting and transformative technologies of our time, but it also comes with a number of significant challenges that researchers and companies are still working hard to overcome.
First and foremost, safety is the biggest concern. Self-driving cars rely on a combination of sensors like cameras, radar, and LIDAR to understand their surroundings. However, these sensors can sometimes struggle in bad weather conditions like heavy rain, fog, or snow, or when objects are hidden from view. Even more challenging is teaching the car's artificial intelligence to correctly interpret what it "sees" and make the right decisions. Unlike human drivers, who can use intuition and experience to handle unexpected situations, AI systems can get confused by unusual or rare events they haven't encountered before. This makes it difficult to guarantee that autonomous vehicles will always behave safely in every possible scenario.
Another major challenge is the sheer complexity of real-world driving. Roads are full of unpredictable elements—pedestrians jaywalking, cyclists weaving through traffic, construction zones, or emergency vehicles suddenly appearing. Autonomous cars need to be able to handle all of these "edge cases" smoothly, which requires incredibly sophisticated

software and a huge amount of data to train the AI. Testing these systems thoroughly is also a massive undertaking. Companies have to simulate millions of miles of driving and conduct extensive real-world tests to prove their technology is reliable enough for public use.
Beyond the technical side, there are also important ethical and legal questions. Laws and regulations around autonomous vehicles are still evolving, and policymakers need to find ways to protect both users and other road users while encouraging innovation.
Infrastructure is another factor. Our current roads and traffic systems were designed for human drivers, not for AI-controlled vehicles. To fully benefit from autonomous driving, cities might need to upgrade traffic signals, add dedicated lanes, or improve digital mapping. These changes require significant investment and coordination.
Finally, public trust plays a huge role in the adoption of autonomous vehicles. Many people are still skeptical or fearful about handing over control to a machine, especially after some high-profile accidents involving self-driving cars. Building confidence through transparency, education, and proven safety records is essential for the technology to become widely accepted.
In summary, while autonomous driving promises safer roads, greater mobility, and many other benefits, it faces a complex mix of technical, ethical, legal, and social challenges. Overcoming these will take time, collaboration, and continuous innovation, but the potential rewards make it a goal worth pursuing.

1.3 Goals and Contributions

The primary goals of this Master's thesis are:

1. **To design and implement a robust Nonlinear Model Predictive Control (NMPC) framework** capable of handling complex vehicle dynamics and real-time constraints for autonomous driving applications.

2. **To establish a high-fidelity co-simulation environment** by effectively integrating MATLAB/Simulink with the CARLA simulator, providing a realistic platform for testing and validation.

3. **To develop essential functionalities within the NMPC system** for accurate vehicle control, including optimal trajectory generation, input signal manipulation, and a specialized dispatcher for actuator commands.

4. **To thoroughly validate the performance and robustness of the developed NMPC system** across a comprehensive set of challenging driving scenarios, ranging from standard maneuvers to complex multi-vehicle interactions in both CARLA and a simplified MATLAB/Simulink environment.

5. **To analyze the impact of key NMPC parameters**, such as the curvature parameter, on vehicle behavior, tracking accuracy, and control effort, thereby providing insights for optimal controller tuning.

Beyond the primary goals outlined above, this thesis makes several key contributions to the development of NMPC systems for autonomous vehicles:

1. **Development of a Novel Dispatcher Block**: A custom-designed Dispatcher block was developed to accurately convert the NMPC-generated acceleration commands into realistic throttle and brake inputs for the vehicle actuators. This block incorporates a simplified, yet effective, longitudinal vehicle dynamics model coupled with a simplified gear-shifting model to accurately capture the vehicle's behavior under varying operating conditions. This approach enhances the practical applicability of the NMPC by translating abstract control signals into actionable vehicle commands.

2. **Enhanced Error Function for Lateral Deviation**: An advanced error function was implemented within the control framework. This function is specifically designed to account for both positive and negative values of the lateral error, enabling the NMPC to precisely determine if the vehicle is deviating towards the right or left of the desired trajectory. This granular understanding of lateral deviation contributes to more accurate and responsive path tracking.

3. **Robust Coordinate and Parameter Transformation Utility**: A dedicated function was developed to facilitate the seamless transformation of crucial vehicle parameters and states from the global reference frame to the vehicle's local reference frame. Specifically, this includes the conversion of global velocities and accelerations. Furthermore, this utility precisely transforms global X and Y coordinates by considering and compensating for the distance between the vehicle's bounding box center and its center of mass/gravity. This precise transformation is critical for accurate real-time control calculations within the NMPC.

4. **Development of a Simplified MATLAB/Simulink System for Complex Scenarios**: A streamlined and efficient MATLAB/Simulink system was developed for the simulation of highly complex driving scenarios, rarely addressed in existing literature. This system places a particular emphasis on:

   - **Refined nlcon Function for Constraint Handling**: Significant effort was dedicated to improving the nlcon function, which defines the nonlinear constraints for the NMPC optimization problem. This enhancement ensures that the vehicle adheres strictly to safety and operational boundaries, particularly crucial in complex interactions.
   - **Adaptive Curvature-Based Velocity Adjustment**: An adaptive velocity function was integrated, which dynamically adjusts the vehicle's speed based on the path curvature. This enables the vehicle to optimally and safely navigate turns, strictly adhering to traffic regulations and avoiding potential collisions, by ensuring an appropriate speed profile for cornering maneuvers. This capability is particularly vital for safe and compliant operation in diverse road geometries.

1.4 Thesis Structure Overview

This thesis presents the development and validation of a Nonlinear Model Predictive Control (NMPC) system for autonomous vehicles, integrated within a high-fidelity co-simulation environment combining MATLAB/Simulink and CARLA. The work addresses the critical challenge of achieving robust and precise vehicle control in complex and dynamic driving scenarios. Notably, this research extends beyond standard autonomous driving maneuvers by focusing on highly intricate and less-explored scenarios in current literature, such as extra-urban and urban lane merging, and multi-vehicle roundabout merging, simulated primarily within the MATLAB/Simulink environment.

The structure of this thesis is as follows:

- **Chapter 1** *Introduction* outlines the organization and provides a brief introduction to the subsequent chapters.
- **Chapter 2** *Autonomous Driving: State of Art* provides a comprehensive overview of autonomous driving technology, including SAE levels, ADAS functions, sensor technologies, and current applications. It also discusses various AV system architectures, communication methods, and the V-cycle development process, alongside an overview of popular simulation environments.
- **Chapter 3** *Vehicle Models* details the mathematical models employed to represent vehicle dynamics, including the 2-Degrees of Freedom longitudinal model and the Single Track Model (bicycle model), crucial for accurate control system design.
- **Chapter 4** *Control Systems* introduces fundamental control strategies for autonomous driving, with a particular focus on the principles of Nonlinear Model Predictive Control (NMPC) and its advantages for trajectory optimization.
- **Chapter 5** *CARLA Environment* describes the CARLA simulator, its key modules, and the developed MATLAB-CARLA interface, which facilitates the co-simulation environment for control system testing.
- **Chapter 6** *NMPC System Design* elaborates on the detailed design and implementation of the NMPC system within Simulink, explaining the various functional blocks and their roles in the overall control architecture.
- **Chapter 7** *MATLAB-CARLA Integration - Simulation Results* presents and analyzes the performance of the NMPC system in a variety of standard driving scenarios, such as indirect right turns and suburban junctions, comparing different controller tunings.
- **Chapter 8** *Complex Scenarios* focuses on the implementation of the NMPC in more intricate and challenging environments not typically found in standard literature, detailing the methodologies for trajectory generation and constraint handling.
- **Chapter 9** *Complex Scenarios - Simulation Results* showcases the performance of the NMPC in these advanced scenarios, including extra-urban and urban lane merging, and multi-vehicle roundabout merging.
- **Chapter 10** *Conclusions* summarizes the key findings of the research, discusses the limitations encountered, and proposes avenues for future work and improvements to the NMPC framework.

Chapter 2

Autonomous Driving: State of Art

2.1 Autonomous Driving Concepts

In the following subsections, various concepts related to autonomous driving will be presented, with particular emphasis on SAE levels, sensors, ADAS and on both present and future applications.

2.1.1 SAE Autonomous Levels

AVs are designed to perform a wide range of driving tasks in different traffic and road conditions. However, the degree to which a vehicle can drive itself depends on its level of automation, which is classified using a standard developed by the "Society of Automotive Engineers" (SAE).
The SAE defines six levels of driving automation, ranging from Level 0 (no automation) to Level 5 (complete automation). These levels help to clarify the capabilities and limitations of each system.

The following are the Autonomous Driving Levels:

- **Level 0 – No Automation:** the driver is fully responsible for all driving tasks. The vehicle may provide warnings or momentary assistance, such as emergency braking, but does not take control.
- **Level 1 – Driver Assistance:** the control is shared between the driver and the system. The system can assist with steering or acceleration/deceleration, but not both at the same time. The driver is not expected to be substituted by the system, but only assisted.
- **Level 2 – Partial Automation:** the vehicle can control both steer and accelerate/decelerate under certain conditions, typically on highways. However, the driver must continue to monitor the environment and be prepared to intervene. This is the field of the *Advanced Driving Assistance Systems* (ADAS).
- **Level 3 – Conditional Automation:** from this level on, the system is able to perform some tasks in place of the driver. The system manages most of the aspects

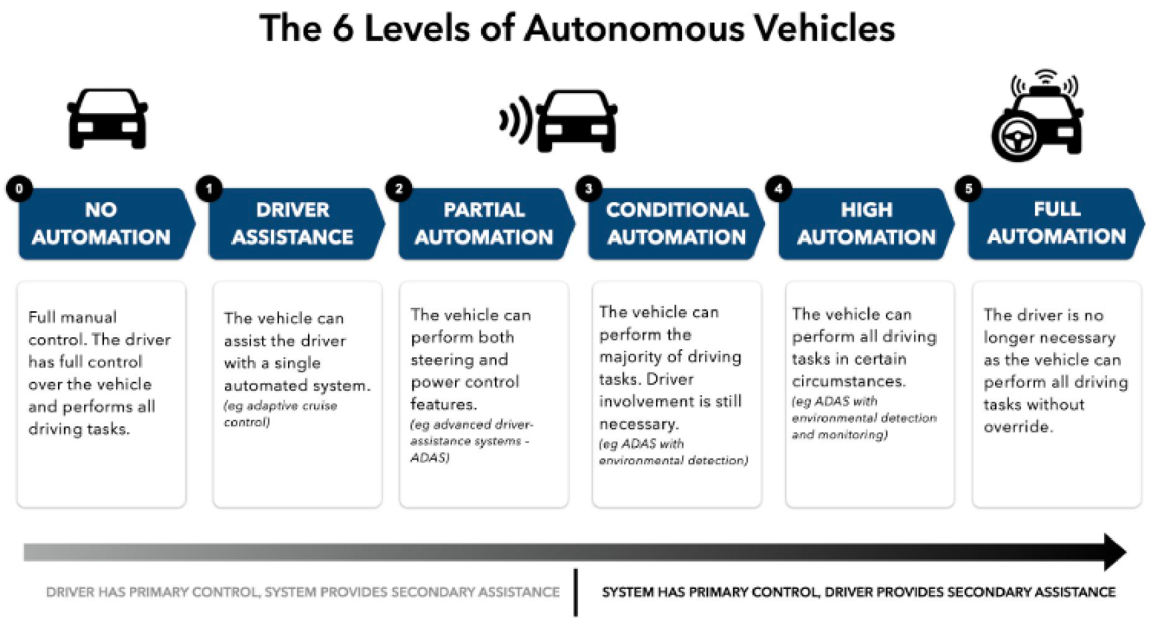

The 6 Levels of Autonomous Vehicles
0
NO AUTOMATION
Full manual control. The driver has full control over the vehicle and performs all driving tasks.
1
DRIVER ASSISTANCE
The vehicle can assist the driver with a single automated system.
(eg adaptive cruise control)
2
PARTIAL AUTOMATION
The vehicle can perform both steering and power control features.
(eg advanced driver-assistance systems - ADAS)
3
CONDITIONAL AUTOMATION
The vehicle can perform the majority of driving tasks. Driver involvement is still necessary.
(eg ADAS with environmental detection)
4
HIGH AUTOMATION
The vehicle can perform all driving tasks in certain circumstances.
(eg ADAS with environmental detection and monitoring)
5
FULL AUTOMATION
The driver is no longer necessary as the vehicle can perform all driving tasks without override.
DRIVER HAS PRIMARY CONTROL, SYSTEM PROVIDES SECONDARY ASSISTANCE
SYSTEM HAS PRIMARY CONTROL, DRIVER PROVIDES SECONDARY ASSISTANCE

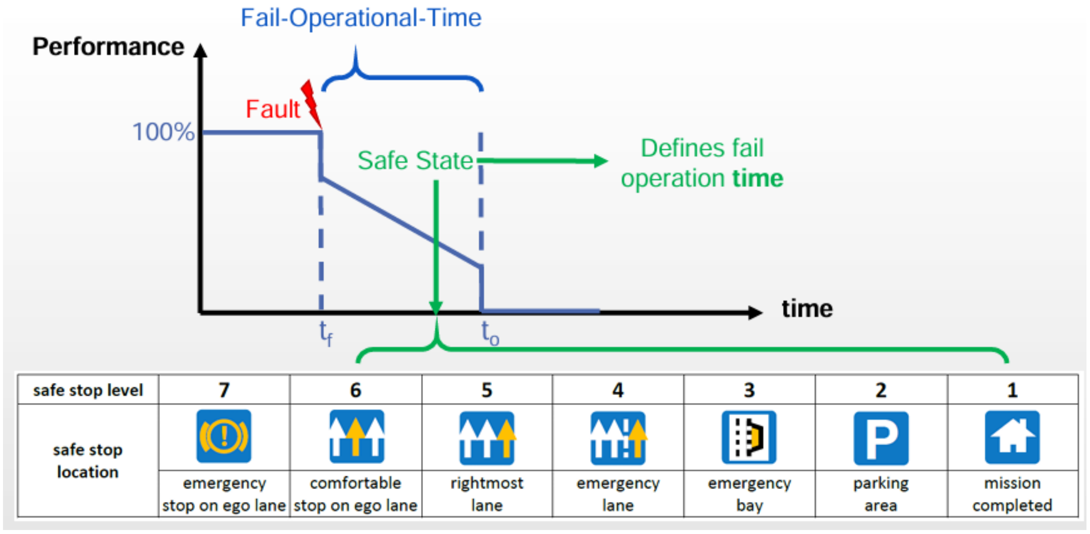
Figure 2.1: SAE levels of Autonomous Driving [1]

of driving in specific scenarios, such as driving on the highway in light traffic. The driver can disengage temporarily but must be ready to take the control if the system asks him to do so (the so called *Take Over Request*).
The problem is that this creates a "grey" area that is difficult to be addressed, since the reaction time of the driver changes a lot, depending on many factors. For this reason, many carmakers skip level 3, directly jumping from level 2 to level 4, avoiding this "grey" area.

- **Level 4 – High Automation:** the vehicle can operate without human intervention in designated areas or conditions (the so called *Operational Design Domain*). Out of these conditions, the system cannot take the control of the vehicle.
  Actually there are prototypes of vehicles belonging to this level, but they are not commercially available.

- **Level 5 – Full Automation:** the vehicle is fully autonomous and capable of operating in all the environments and conditions without any human input. These vehicles may not even include a steering wheel or pedals.
  It is not clear if this level will be ever reached.

Moreover, moving toward higher SAE levels, it becomes more important that the vehicle is able to operate under fault conditions. When there is a fault, the system has to identify the fault operational time, that is a period of time in which the system operates in an autonomous way to return the vehicle to a safe state. This state depends on the safe stop level.



safe stop level	7	6	5	4	3	2	1
safe stop location	emergency stop on ego lane	comfortable stop on ego lane	rightmost lane	emergency lane	emergency bay	parking area	mission completed

Figure 2.2: Fail Operational Behaviour

As it is possible to see from the figure, the lower the safe stop level, the safer the state of the vehicle. Obviously, lower safe stop levels are preferable, but also more demanding. [2]

2.1.2 Operational Design Domain

As it was mentioned in the previous section, SAE range levels depend on the conditions under which a specific autonomous driving function is designed to operate. This set of conditions is called *Operational Design Domain* (ODD), and it includes, but is not limited only to, environmental, topological factors and traffic characteristics. Obviously, the higher the level of autonomy, the wider the ODD. So, to design a function for an autonomous vehicle, ODDs must be established.
The steps that make up an ODD are the following:

- identification of the road/route network;
- definition of the characteristics of the road/route network and the infrastructure of which it is made up of;
- definition and identification of the constraints of which the road/route network is composed.

Some of the main attributes of an ODD are listed below:

- *Scenary Attributes:*
  - Zones: geofences, schools, intersection areas;
  - Drivable area: drivable area type, geometry, signs, area surface;
  - Junctions: normal, compact, double, roundabout attributes, T-junctions, Y-junction, crossroads;
  - Special structures: automatic access control, bridges, pedestrian / rail crossings, tunnels, toll plaza;
  - Fixed road structures: buildings, street lights, street furniture, vegetation;

- Temporary road structures: construction site, roadworks, road signage.

- *Environmental Attributes:*
  - Weather: wind, rainfall, snowfall, cloudiness;
  - Particulates: marine, non-precipitating water droplets or ice crystals, sand and dust, smoke and pollution;
  - Illumination: daytime, night or low-ambient lighting condition, artificial illumination.

- *Dynamic Elements:*
  - Traffic: dynamic agents (special vehicles, blue waves vehicles, animals, pedestrians) and stationary agents (injured road users and animals);
  - Vehicles equipped with autonomous driving functions. [3]

2.1.3 ADAS Functions

In the context of autonomous driving, ADAS technologies play a key role. ADAS stands for Advanced Driver Assistance Systems. They are technologies designed to enhance safety and improve the driving experience by helping the driver avoid accidents or make driving more comfortable.

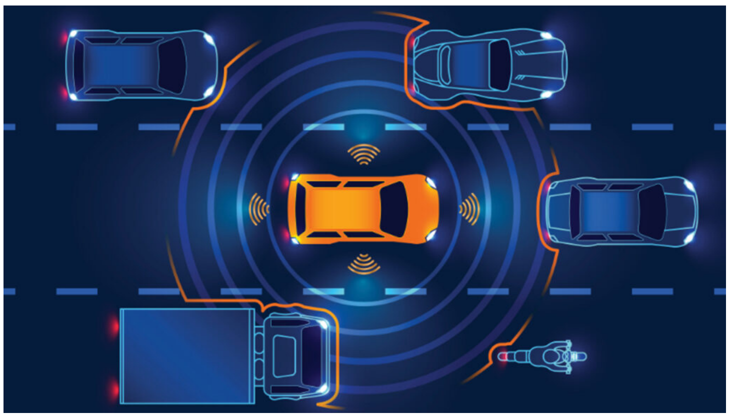


Figure 2.3: Vehicles Detection [4]

The following are the main ADAS functions currently on the market:

- **ABS (Anti-lock Braking System)**: a safety system used to prevent the wheels from skidding excessively and locking during braking, allowing the vehicle to maintain the desired trajectory. This system has been compulsory in Europe since 2004.

- **ACC (Adaptive Cruise Control)**: similar to Cruise Control, with the additional function of controlling the speed to maintain a predefined distance from the vehicle in front. There are different types of ACC: some use a *constant distance gap*, others a *constant time gap*, which is more reliable since the distance depends on the speed of the vehicle. This function makes use of a radar, which monitors in real-time the distance and relative speed to the vehicle in front.

- **AEBS (Automatic Emergency Braking System)**: also in this function, there is the use of radar to detect a possible collision; this safety system prevents the vehicle from colliding with another agent of the road (vehicle, pedestrian, cyclist, or other obstacle) by automatically activating the braking system. This function is mandatory in Europe from 2022.

- **AFS (Adaptive Front-lighting System)**: a system that automatically activates the headlight beams when the vehicle is cornering.

- **ALC (Adaptive Light Control)**: a system that controls the vehicle's headlight beams when another vehicle is detected in front.

- **APS (Automatic Parking System)**: a system that autonomously parks the vehicle. There are 2 types of APS: hands-free (the driver only needs to control the accelerator and brake pedals) and fully automatic.

- **BSW (Blind Spot Warning)**: detects the presence of an obstacle in a blind spot and warns the driver via an acoustic and/or visual signal.

- **CC (Cruise Control)**: a system that allows the driver to maintain a constant cruising speed.

- **ESP (Electronic Stability Program)**: corrects the vehicle's trajectory when excessive slitting is detected or when braking on a curve. It uses various sensors, including wheel speed sensors, accelerometers and yaw rate sensors.

- **LC (Lane Centering)**: requires the driver to have his hands on the steering wheel at all times. This system returns the vehicle to the center of the lane. Unlike Lane Keeping, this system is always active.

- **LK (Lane Keeping)**: brings the vehicle back into the lane. This system only works when the vehicle is on the lane separation line.

- **NVS (Night Vision System)**: allows the driver to detect other vehicles, pedestrian and other obstacles even in adverse conditions (i.e. fog, night and rain). This function requires *thermal cameras* (a type of camera detecting pedestrians and obstacles according their temperature difference with respect to the external environment).

- **TJA (Traffic Jam Assist)**: this function combines ACC, LC to operate safely in heavy traffic situations.

- **TPMS (Tire Pressure Monitoring System)**: this system continuously monitors the tire pressure, i.e. whether the tires are over-inflated or under-inflated, and via a warning light alerts the driver if the pressure is not safely optimal. [5]

2.1.4 Sensors Overview

Sensors play an important role in the field of autonomous driving because, for safety reasons, certain vehicle parameters, such as speed, acceleration, but also the presence of other road users, such as vehicles, pedestrians and cyclists, must be monitored continuously.

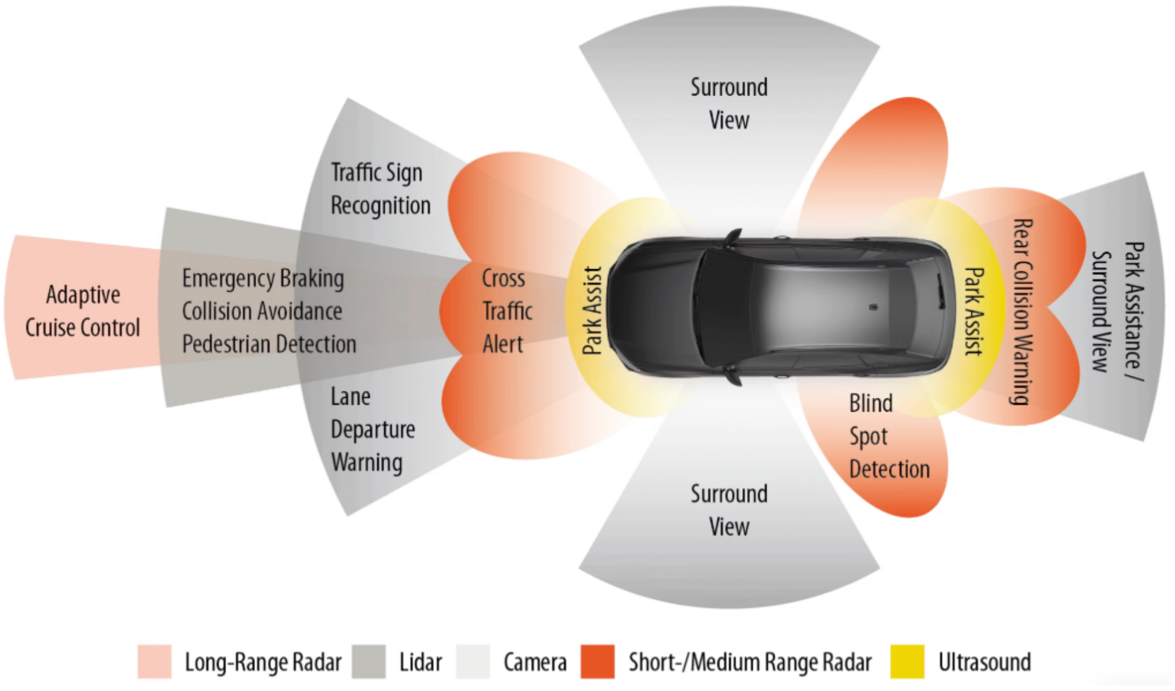


Figure 2.4: Sensors [6]

The main sensors used in autonomous driving are described below:

- **Radar (Radio Detection and Ranging)**: a sensor that emits radio waves that reflect off objects. By measuring the time and frequency shift of the returned signal (Doppler effect), it calculates both distance and relative speed of objects, even in poor visibility (fog, rain, darkness).
  It is robust in all the weather conditions and detects moving objects with a satisfactory accurancy. It has a wide range of ADAS applications: Adaptive Cruise Control (ACC), Forward Collision Warning (FCW) and Automatic Emergency Braking (AEB). [6]

- **LiDAR (Laser Imaging Detection and Ranging)**: it emits rapid laser pulses and measures the time they take to return after hitting objects. This creates a high-resolution map of the surroundings, using some reference points (i.e. pillars in a parking area).
  There are different types of LiDARs, depending on the range (short or long ranges), and depending on the dimension of the map (2D or 3D). This sensor has extremely accurate depth perception and it is excellent for object detection and classification. However, it has some weaknesses: it can be affected by heavy rain, fog, or direct sunlight, and the price is very high (thousands of Euros).
  In the field of ADAS, this kind of sensor can be used for: Obstacle detection, Environment mapping for Level 4–5 autonomy and Path planning. [7]

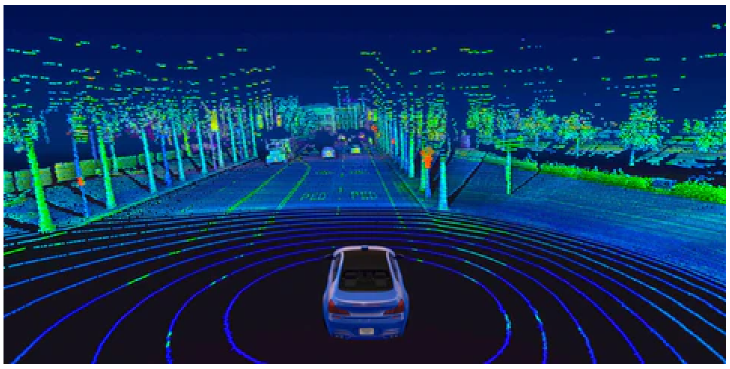


Figure 2.5: 3D LiDAR [8]

- **Cameras (Optical Vision Sensors)**: these sensors typically use RGB image sensors, either monocular or stereo, to capture high-resolution images of the environment. When combined with artificial intelligence (AI) and machine learning algorithms, cameras can detect and classify objects such as vehicles, pedestrians, lane markings, traffic signs, and traffic lights. This rich visual data is essential for interpreting complex scenes and making informed driving decisions.
  There are different types of automotive cameras used in autonomous vehicles, including sensing cameras, surround view cameras, and driver monitoring cameras. Sensing cameras, usually mounted behind the windshield, cover a wide front view and support critical functions like lane keeping assist (LKA), lane change assist (LCA), and automatic emergency braking (AEB). Surround view cameras are placed around the vehicle's perimeter to provide a 360-degree view, aiding in parking and close-range obstacle detection. Driver monitoring cameras focus on the driver's attention and condition to ensure safe operation. [9]

- **Ultrasonic Sensors**: sensors emitting high-frequency sound waves and measure the echo to detect nearby objects. These are short-range sensors (usually between 15 cm and 6 m).
  They are ideal for low-speed maneuvers, and very cheap and reliable.
  Their weaknesses are the limited range and resolution. Their main applications in Autonomous Driving are: Parking Assist and Low-speed collision avoidance. [10]

- **GNSS (Global Navigation Satellite System) + IMU (Inertial Measurement Unit)**: *GNSS* (includes GPS, GLONASS, Galileo, BeiDou) provides geolocation by triangulating satellite signals. *IMU* uses accelerometers and gyroscopes to track motion and orientation. Together, they allow precise localization even when GNSS signals are weak (e.g., tunnels).
  Their main ADAS applications are: High-definition map positioning, Path planning and Stability control.

- **Sensor Fusion**: it is obtained by combining data from multiple sensor types (e.g., radar + LiDAR + cameras) to provide a more reliable and complete understanding of the environment. This allows to mitigate the limitations of individual sensors, improving accuracy and supporting redundancy (critical for L4–L5 autonomy).

2.1.5 Current Applications and Future Outlook

While fully autonomous vehicles (Level 5) are still in the experimental phase, many modern cars already incorporate Level 1 and Level 2 features, such as adaptive cruise control, lane-keeping assistance, and automated parking. Some companies, including *Waymo* and *Tesla*, are testing Level 3 and Level 4 systems on public roads.
In particular, on the autonomous vehicles used by *Waymo*, there is a *sensor fusion* of three sensors: LiDARs (allowing a bird-eye view of pedestrians, cyclists and other vehicles, and covering any blindspots), cameras (long range and 360° of vision field) and radars (allowing greater accuracy in adverse weather conditions) [11].

Figure 2.6: Waymo AV [12]

The development of AVs faces several challenges, including regulatory uncertainty, ethical dilemmas in decision-making, and the need for highly accurate mapping and real-time data processing. Nonetheless, the potential benefits—increased road safety, reduced emissions, and greater accessibility—continue to drive innovation in this field.
In summary, autonomous vehicles represent a significant shift in transportation technology, with the promise of transforming the way we move through the world. As research and testing continue, understanding the different levels of autonomy and their implications is essential for evaluating both the opportunities and limitations of this emerging technology.

2.2 AV System Architectures

Autonomous Vehicle Architectures refer to the organization and design of the hardware and software components that enable the vehicle to perceive the environment, make decisions and move autonomously.
Architectures can be classified according to several criteria:

1. **decision-making approach**;
2. **software architecture**;
3. **SAE autonomous level**;
4. **components** (mechanical, sensors, and so on).

2.2.1 Decision-Making Approaches

The main Decison-Making architectures are the following:

- **Rule-Based Systems** These systems operate by following a predefined set of 'IF-THEN' rules established by experts. Each rule specifies a condition and the corresponding action to be taken when that condition is met.
  For instance, in an autonomous vehicle, a rule might be: 'If the distance to the obstacle is less than 5 m, then slow down'.
  The most important feature of these systems is the *trasparency*: decisions are easy to understand as they are based on explicit rules.
  They can be rigid and do not adapt well to unforeseen situations or complex environments.

- **Heuristic Systems** They use experience-based strategies or 'shortcuts' to solve problems quickly and efficiently, without guaranteeing an optimal but acceptable solution.
  As an example, a vehicle might use a heuristic to choose the lane with the least traffic based on recent observations, rather than analyzing all the possible options.
  These systems are very efficient, enabling quick decisions in complex situations.
  However, they can introduce errors or bias, as they do not always guarantee the optimal solution.
  In autonomous vehicles, *both rule-based systems and heuristics* are often used for behaviour planning and decision-making. [13]

- **AI-Based** It is an approach in which the vehicle makes decisions and interprets its surroundings not thanks to rules handwritten by programmers, but thanks to machine learning, (i.e. experience gathered from data).
  This means that an AI system does not rely on commands such as 'if the traffic light is red, then stop', but rather learns the correct behaviour by observing a large number of real or simulated situations. At the heart of this approach is machine learning, in particular deep learning, which enables the vehicle to learn to see, decide and move in a similar way to a human being, but much faster and on a larger scale.
  One of the first important uses of AI in autonomous vehicles is in perception. Vehicles must be able to recognize objects such as other vehicles, pedestrians, traffic signs, traffic lights and pavements. This is done using deep neural networks, in particular so-called CNNs (Convolutional Neural Networks). These networks manage to transform images from on-board cameras into useful information, such as 'here is a person' or 'this is a lane that bends to the right'.
  A classic example of this approach is the study published by *NVIDIA* in 2016, which showed how a neural network could learn to drive simply by watching footage of human driving. Basically, the system 'sees' what the driver sees and learns to turn the steering wheel in the correct way according to the situation. This is called the end-to-end approach, because it directly links input data (images) to output actions (such as steering). [14]
  But AI does not stop at perception. It is also used in planning, i.e. in deciding what to do: stop, overtake, change lanes, approach an intersection. Here techniques such as *reinforcement learning* can be used, where the system learns by trying out different actions and receiving a reward for the correct ones (e.g. 'you avoided a collision? Great! +1 point'). Another approach is imitation learning, where the system mimics

human behaviour by observing how experienced drivers drive.
Artificial intelligence can also be used in vehicle control, although more classical methods such as PID controllers or predictive models are often preferred in this area. However, in more advanced projects, AI-learned controllers are being experimented with that directly decide how much to brake or how much to turn the steering wheel in real time. [15]

- **Hybrid** The hybrid architecture represents an approach that combines traditional rule-based methods with advanced artificial intelligence (AI) techniques. This combination aims to exploit the strengths of both approaches: the predictability and transparency of rule-based systems and the learning and adaptive capacity of AI systems.

Figure 2.7: Hybrid Architecture [16]

In traditional systems, decisions are made following predefined rules, such as ‘if the traffic light is red, stop’. These rules are clear and easy to interpret, but can be rigid when faced with unforeseen situations. On the other hand, AI techniques, particularly machine learning, allow the vehicle to learn from data and adapt to complex scenarios, although they often operate as a ‘black box’ that is difficult to interpret.
The hybrid architecture integrates these two methodologies: deterministic rules handle standard situations and ensure compliance with road regulations, while AI models deal with more complex and variable scenarios, such as recognizing pedestrians in low visibility conditions or predicting the behaviour of other drivers. For example, a lane change strategy may combine fixed rules for maintaining a safe distance with AI algorithms that assess the speed and trajectory of surrounding vehicles to determine the optimal time to overtake. [17]

2.2.2 Software Architectures

In autonomous vehicles, the software architecture is crucial in determining how the various components of the system interact with each other. There are three main types of software architecture:

- **Centralized** In a centralized architecture, all decisions and data processing take place in a single electronic ‘brain’ of the vehicle. This central computer receives data

from all the sensors (cameras, radar, lidar, GPS, etc.), interprets it, decides what to do and controls the actuators (steering, accelerator, brakes, etc.).
This type of structure is very orderly: each part communicates with a center, which manages everything. It is easier to control and update because there is only one point of intelligence.
This architecture is easier to design and manage overall and more consistent in data, because everything goes through one processor.
However, it can present the *Bottleneck* effect: the central processor can become overloaded. Moreover, a single failure can compromise the entire system and it is less flexible in adapting to complex or evolving systems.
The *NVIDIA DRIVE AGX* architecture is a key example of a centralized approach for autonomous driving. [18]

- **Modular** This architecture divides the software into independent modules, each of which performs a specific task. For example: one module for perception (interpreting the sensors), one for route planning, one for vehicle control, and so on.
  Each module can be developed by different teams, upgraded independently, and even replaced without having to redo everything from scratch. Modularity is a common choice in software engineering in general because it helps manage complexity and promote code reuse.
  The main advantages of this architecture are the increased flexibility in development,

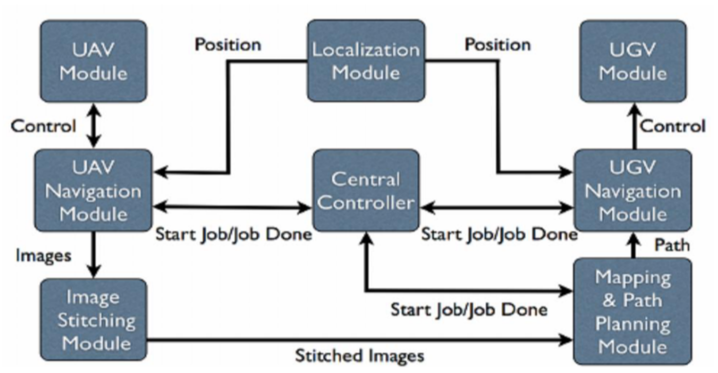


Figure 2.8: Modular Architecture [19]

  the easier maintenance and updates and the fact that each module can be optimized independently.
  However, it needs an efficient communication system between modules and the integration complexity increases (modules must 'understand each other').
  The *Baidu's Apollo* system is an example of an highly modular architecture, with defined APIs between components.

- **Distributed** In a distributed architecture, functionality is distributed over several electronic control units (ECUs) scattered throughout the vehicle. Each ECU is in charge of one aspect (perception, braking, steering, etc.) and communicates with the others via network, often CAN or Ethernet Automotive.
  This is the most common architecture in conventional cars, but was also used in the first autonomous car prototypes. Today, however, it is seen as not very scalable, because it requires a lot of communication and synchronization between the ECUs.

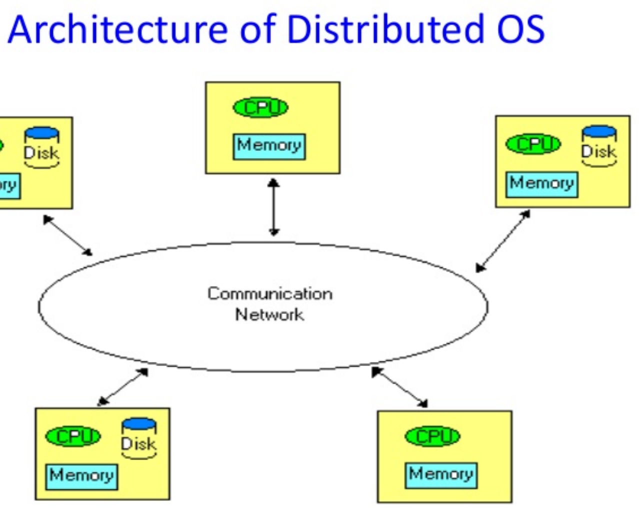


Figure 2.9: Distributed Architecture [20]

This approach reduces the load on a single processor and allows higher *redundancy*: if one unit fails, the rest can continue to work.
However, there are also CONs about this architecture: complexity in communication management, it is difficult to guarantee fast and consistent response times and it is expensive to maintain and update.

2.3 Communication Methods exploited by AVs

Communication between autonomous vehicles is crucial for increasing safety, improving traffic flow and optimizing energy efficiency. Here are the main communication methods that can be used:

1. **V2V (Vehicle-to-Vehicle)** This communication method allows vehicles to communicate directly with each other (i.e. two vehicles exchanging information such as speed, direction, or whether they are about to brake abruptly). This constant dialogue helps vehicles predict the behaviour of others and react accordingly. For example, if a car in front brakes suddenly, the car behind can know instantly, even before the driver (or automatic system) sees the brake lights. This can greatly reduce the risk of rear-end collisions. The main technologies used are:

    - **DRSC (Direct Short Range Communication)** Communication that takes place over short distances. An example of this is using the **WAVE** protocol, which is a specific protocol that enables communication vehicular environment. It is composed of several layers: a *physical* layer (PHY, allowing the direct exchange of information), the *MAC* layer (contains information regarding the Access Points and channels to which the vehicle can connect and their coordination), the *logical Link control* layer (LLC, which functions as a mailbox, i.e. receives packets from the other layers, and organizes their distribution) and other layers representing the destinations (such as the WSMP, explicitly designated for messages in a vehicular environment).

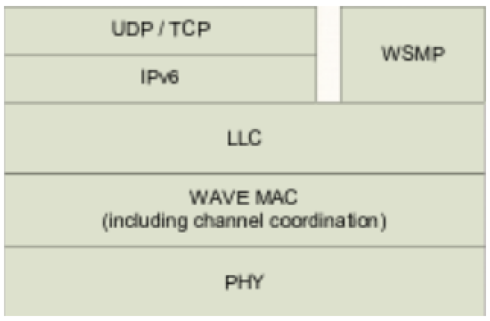


Figure 2.10: WAVE Protocol Stack [21]

- **C-V2X (Cellular to Everything)** A type of communication that exploits the use of cellular networks (3G, 4G, 5G) to communicate not only between vehicles, but also with pedestrians, infrastructure, etc.

2. **V2I (Vehicle-to-Infrastructure)** Here, the vehicle communicates with elements of the road infrastructure: traffic lights, roadside sensors, intelligent signals, toll booths, etc. A practical example is that a vehicle receives data from the traffic light and knows in advance if it will turn red, or if there are roadworks just ahead. This information allows the car to adjust speed, choose alternative routes or stop in time, without relying solely on visual sensors. This is especially useful in low visibility conditions or in complex environments such as cities. The most common technologies are, again, DRSC (such as WAVE, the latest Wi-Fi versions such as 822.11p, and Ethernet) and C-V2X.

3. **V2N (Vehicle-to-Network)** In this case, the vehicle connects to an external network, such as the Internet or the cloud, often via 4G or 5G. This type of communication provides access to global information: real-time traffic news, weather forecasts, software updates of the autonomous driving system, and so on. It is as if the car is connected to a central 'brain' that keeps it constantly updated on what is happening in the world around it. This technology is also what allows data to be exchanged on a large scale: for example, if a car detects a hole or a slippery area, it can communicate it to the cloud, which then transmits it to other cars in the area.

4. **V2P (Vehicle-to-Pedestrian)** With V2P, the autonomous vehicle can communicate with pedestrians and cyclists. This is crucial to protect the most vulnerable road users. For example, a person crossing the road could have an app on their phone that signals their presence to the neighbouring car. Or, thanks to technologies such as Bluetooth or Wi-Fi Direct, the vehicle can detect a pedestrian even if they are not directly visible, for example behind another parked vehicle. This communication helps the car predict pedestrian movements and adapt its driving to avoid accidents, even in complex scenarios such as pedestrian zones or school areas. The most used technologies for this kind of communication are: C-V2X, Wi-Fi and Bluetooth.

5. **V2X (Vehicle-to-Everything)** V2X is a term that encompasses all the types of communication just described: V2V, V2I, V2N and V2P. The goal of V2X is to create a fully connected mobility ecosystem where every vehicle, infrastructure, device and person can exchange information in real time.

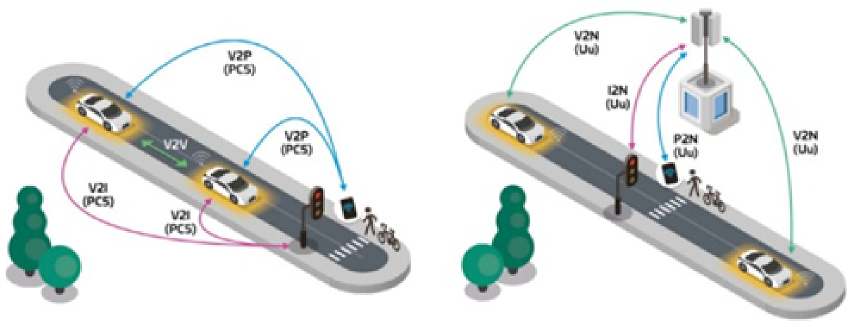


Figure 2.11: V2X Communication

There are two types of V2X:

- **Direct Mode**: direct communication without a network.
- **Indirect Mode**: via the mobile network, i.e. using cellular networks such as 4G or 5G. This communication covers a very wide space (allowing communication between users far away from each other). [3]

2.4 V-Cycle

In the automotive industry, the development of vehicles and their electronic systems is a highly complex process that requires precision, continuous testing and quality control at every stage. To deal with this complexity, a methodological approach called the V-Cycle, or V-model, is used.

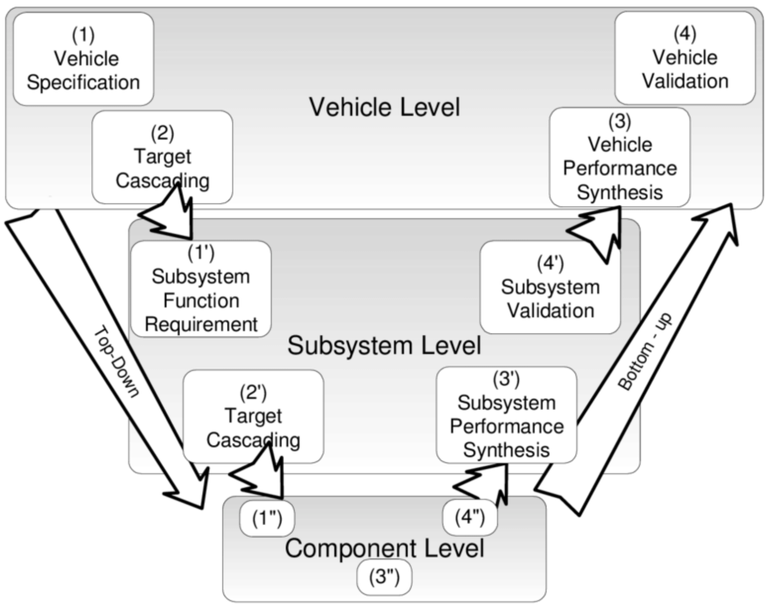

(1) Vehicle Specification
(4) Vehicle Validation
Vehicle Level
(2) Target Cascading
(3) Vehicle Performance Synthesis
(1') Subsystem Function Requirement
(4') Subsystem Validation
Subsystem Level
Top-Down
Bottom - up
(2') Target Cascading
(3') Subsystem Performance Synthesis
(1")
(4")
Component Level
(3")

Figure 2.12: V-Cycle [22]

The V-Cycle is a schematic representation of the systems development process, where:

- The left side of the 'V' represents the design and requirements definition phases;
- The right-hand side represents the testing, verification and validation phases;
- The lowest point of the 'V' corresponds to the actual implementation, i.e. the development of the software or hardware.

From the image above, the main steps in this process can be defined:

1. **Specifications Definition** The customer's requirements are analyzed and the expected functionality of the vehicle or system is specified. This is the basis for all subsequent phases.
2. **System Design and Architecture** The system is broken down into subsystems (e.g. brakes, infotainment, ADAS) and their architecture is defined. [23]
3. **Detailed Design** Each component is designed in detail, both hardware and software. [24]
4. **Implementation** The designed components are implemented physically or coded in software.
5. **Unit Tests** Each individual unit or module is tested isolated.
6. **Integration Testing** Modules are assembled into subsystems and tested as a whole.
7. **System Testing and Validation** The entire system is tested to ensure that it meets the initial requirements and functions correctly under real-life conditions.

A crucial aspect of the V-Cycle is the association between each design phase on the left-hand side and the corresponding test phase on the right-hand side. For example, the requirements defined at the beginning are verified during the validation phase of the system, ensuring that each designed element has been adequately tested and validated. This model is particularly appreciated in the automotive industry for its ability to detect and correct problems early, reducing the risk of errors in the advanced stages of development and optimizing production time and costs.

2.4.1 Simulation and Testing Techniques: MIL, SIL, PIL, HIL, DIL

Within this process, different simulation and testing approaches are used to ensure that each component and system functions correctly before integration into the real vehicle. These approaches are known as MIL (Model-in-the-Loop), SIL (Software-in-the-Loop), PIL (Processor-in-the-Loop) and HIL (Hardware-in-the-Loop). Each of these corresponds to specific phases of the V-Cycle and has distinct purposes.

1. **Model-in-the-Loop (MIL)** This approach is used in the V-Cycle phase of Design and initial development.
   In this phase, mathematical models of components or control systems are tested in a simulation environment. This allows engineers to verify and validate control algorithms and decision logics before implementing them in actual software. [25]

2. **Software-in-the-Loop (SIL)** It is used in the Software development and initial testing phases.
   The control software, written in the final programming language, runs on a computer and interacts with a simulation of the physical system. This allows the code to be tested under controlled conditions, identifying and correcting any errors before moving on to the real hardware. [26]

3. **Processor-in-the-Loop (PIL)** Important when integrating software with specific hardware.
   The software is executed on the microprocessor or control unit intended for the final product, while the rest of the system is simulated. This approach makes it possible to evaluate the performance of the software on the real processor, checking aspects such as execution times and resource utilization.

4. **Hardware-in-the-Loop (HIL)** Approach used in Integration test and final validation phases.
   The control unit is connected to a test bench that emulates the rest of the physical system, including sensors and actuators. This allows the behaviour of the hardware and software to be tested in a controlled environment that replicates real operating conditions, identifying any problems prior to implementation in the vehicle. [27]

5. **Driver-in-the-Loop (DIL)** Approach in which a human driver interacts directly with a simulator that reproduces the behavior of a vehicle or driving environment in real time. This approach allows human reactions and perceptions to be included in the vehicle development and testing process, allows observation and analysis of how a human driver interacts with vehicle systems, providing valuable information on the usability and effectiveness of driver assistance systems.
   In the context of V-Cycle, DIL finds application primarily in the validation and verification phase. [28]

The use of these approaches in the V-Cycle offers numerous advantages [27]:

- **Early detection of errors**: By testing and validating models and software in the early stages, problems can be identified and corrected before they become costly or difficult to solve in the later stages.
- **Reduced development costs and time**: Simulation reduces the need for physical prototypes, speeding up the development process and reducing associated costs.
- **Improved safety**: Testing systems in simulated environments allows evaluation of behaviour in critical or hazardous scenarios without risk to people or equipment.

Summarizing, integrating MIL, SIL, PIL and HIL into the V-Cycle of automotive development is an effective strategy to ensure that control systems are accurately designed, tested and validated, improving the quality and reliability of the final product.

2.5 Simulators' Overview

In the automotive industry, virtual simulators are key tools used in various stages of the V-Cycle, a "V" development model that represents the systems engineering process. In the early stage, on the left side of the "V," high-level simulations of the system help

define the requirements and specifications of the design. Continuing downward, during the development phase, subsystem- and component-level simulations are employed to refine and validate individual parts of the system. Finally, on the right side of the "V," in the testing and validation phase, virtual simulators allow the integration and overall performance of the system to be verified. The use of a single simulation tool through all these phases can improve the efficiency and consistency of the development process. [29]
A virtual simulator in the automotive field is composed of several key elements:

- **Software Models**: Mathematical representations of vehicle components, such as the engine, vehicle dynamics, and electrical systems. These models can be used for both offline (Model-in-the-Loop) and online (Hardware-in-the-Loop) simulations. [30]
- **Hardware Interfaces**: Physical components that enable communication between the simulator and the actual electronic control units (ECUs), facilitating Hardware-in-the-Loop (HIL) testing. [31]
- **Virtual Environment**: A realistic 3D environment that reproduces driving scenarios, traffic and environmental conditions, used to test and validate driver assistance systems and autonomous functions.
- **Analysis Tools and Dashboards**: Software for collecting and analyzing data generated during simulations, providing metrics on performance and user interaction with the system. [32]

These components work together to create a comprehensive simulation environment that supports the development and validation of automotive systems throughout the V-Cycle.
There are several simulators that can be used to conduct simulations on autonomous vehicles. The main ones, which will be described in the following sections are: CARLA, LGSVL, Sim4CV, Microsoft AirSim, CarMaker and SCANeR.

2.5.1 CARLA

Figure 2.13: CARLA Logo [33]

CARLA (Car Learning to Act) is an open-source simulator designed for autonomous driving research and development. It is a key tool for testing complex algorithms in safe, realistic, and fully controllable environments before moving on to experimentation in real vehicles. [34]
CARLA was developed primarily in C++ and Python, with key support from *Unreal Engine 4*, a graphics engine developed by Epic Games. The use of Unreal Engine enables CARLA to offer realistic environments with high visual fidelity, accurate simulation of

physics and vehicle behavior and advanced weather effects (rain, fog, sun, etc.).
Unreal Engine provides the basic graphics and physics, while the vehicle simulation and control logic is done through C++ and Python.
One of CARLA's strengths is its Python API, which allows developers and researchers to control almost every aspect of the simulator. Here are some of the main functionalities:

- **Vehicle control**: acceleration, braking, steering, engine status, etc.
- **Sensor management**: you can mount RGB cameras, LIDAR, radar, GPS, IMU, etc. on each vehicle and collect data in real time.
- **Environmental control**: dynamic changing of light, weather, traffic density, etc. [35]
- **Scenario building**: creation and scripting of complex scenarios (pedestrians, obstacles, critical situations, etc). The simulator follows the OpenDRIVE standard for roads and urban settings.
- **Data acquisition and logging**: real-time export of data to be used for training and validation of artificial intelligence models.

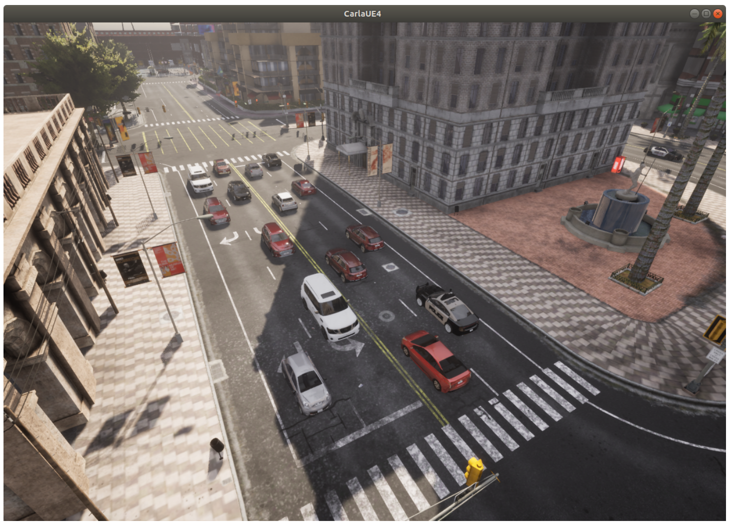


Figure 2.14: CARLA Environment [36]

The whole interaction with the simulated environment takes place via a Python client, which communicates with the CARLA server (running on Unreal Engine) via an RPC (Remote Procedure Call) based protocol.
CARLA is highly modular, which allows integration with machine learning frameworks (such as PyTorch or TensorFlow), interoperability with external simulators such as SUMO for urban traffic simulation and the integration with ROS (Robot Operating System) for advanced robotic projects.
Moreover, CARLA was designed to support both supervised training (dataset labeling, semantic segmentation, depth map, etc.) and reinforcement learning because of its ability to simulate complex environments and measure reward in real time. The following image is an example of CARLA Semantic Segmentation:

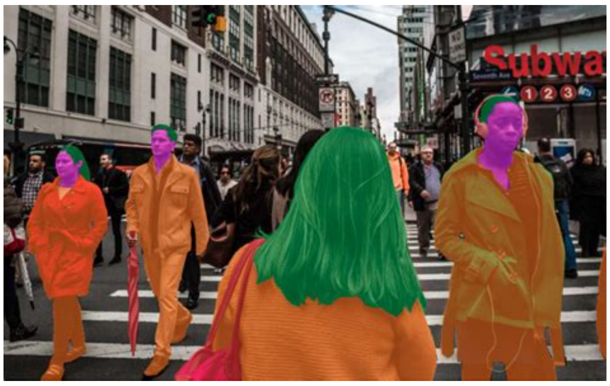


Figure 2.15: CARLA Semantic Segmentation [37]

CARLA has a very active community, is constantly updated and supported by big names in the automotive and academic industries. The project is available on GitHub where versions, fixes, new maps and features are released.

2.5.2 LGSVL

Figure 2.16: LGSVL Logo [38]

LGSVL (LG Silicon Valley Lab) Simulator is a high-fidelity simulator developed by LG Electronics America to support autonomous vehicle research and development. It aims to provide a virtual environment where the entire autonomous driving stack-from perception to planning and control-can be tested and validated without the risk and cost of road testing. [39] It is designed for realistic, interactive simulations that are compatible with real-world driving software such as Apollo (from Baidu) and Autoware.
LGSVL uses Unity, a powerful 3D graphics engine, to generate detailed city, rural, and highway environments, simulated traffic (pedestrians and cyclists can also be added), and realistic visual effects such as fog, sunlight, and rain. In addition, Unity enables photorealistic simulation useful for testing computer vision algorithms.

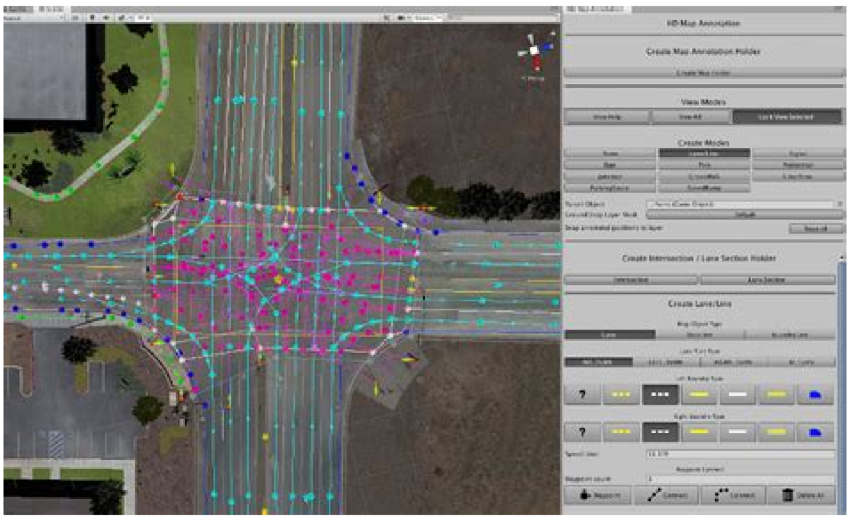


Figure 2.17: Unity Environment [40]

This simulator was developed in C++ (for internal simulator logic within Unity) and in Python (used through an external API, to write scripts that control everything: vehicle spawn, environmental conditions, sensors, data logging).
LGSVL is designed to exactly simulate the sensory configuration of a real vehicle. Various types of sensors can be configured, such as RGB Cameras with intrinsic parameters (FOV, resolution, frame rate), multi-beam 3D LiDAR with intensity and distance return, Radar, IMU (Inertial Measurement Unit) and GPS. It can even simulate sensory errors or latency delays, which are critical for testing the robustness of algorithms.

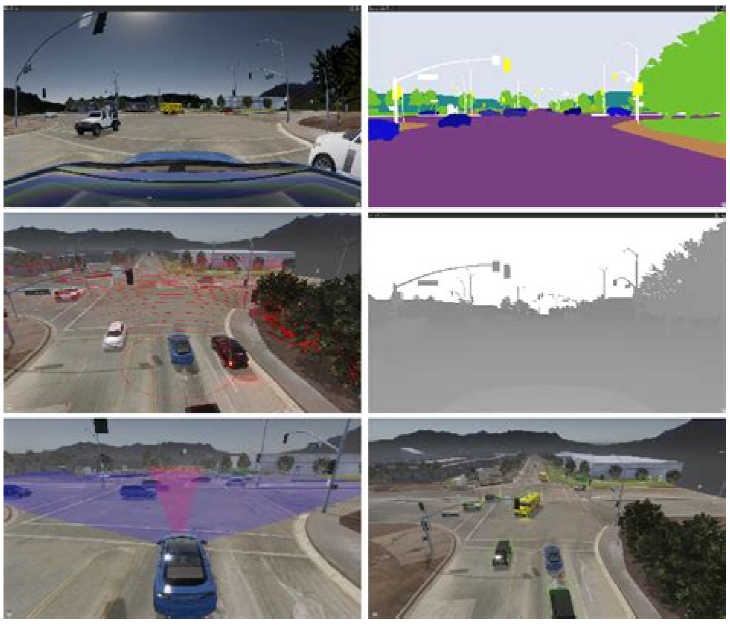


Figure 2.18: Mounted Sensors [41]

One of the strengths of LGSVL is its native integration with Autoware and Apollo, two of the leading open-source autonomous driving stacks. Using this simulator, we can:

- Connect the simulator in real time to the ROS stack.
- Send simulated sensor data to the perception software.

- Receive control commands from the stack and drive in the simulated world.

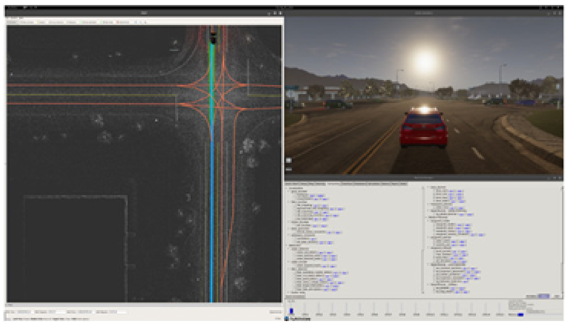


Figure 2.19: Autoware Simulation Interface [41]

This allows end-to-end testing, where all developed code can be tested in simulation before being fitted to a real car.
As in CARLA, weather conditions (rain, sun, clouds, fog), time of day, traffic intensity, and pedestrian and NPC vehicle behavior can be dynamically modified in LGSVL. These options allow to test corner cases (borderline situations), such as sudden crossings, blind intersections or extreme weather conditions.
LGSVL's Python API is powerful and well documented. It allows to:

- Create scenarios (spawn vehicles, pedestrians, objects).
- Control simulated or autonomous vehicles.
- Record data from any sensor (also in .bag format for ROS).
- Launch batch simulations or automated experiments. [41]

2.5.3 Sim4CV

Sim4CV (Simulation for Computer Vision) is a photorealistic simulator designed for training and evaluation across various computer vision applications. Developed using Epic Games' Unreal Engine 4 (UE4), it leverages advanced graphics and a modern physics engine to simulate realistic interactions between vehicles, unmanned aerial vehicles (UAVs), and animated human characters within detailed 3D urban and suburban environments.
The simulator features multiple vehicles, including two passenger cars, a remote-controlled truck, and two UAVs, all operable within various pre-designed maps. An intuitive graphical user interface allows users to adjust simulation settings easily, while an external map editor enables the creation of custom scenarios using modular elements like roads, trees, and buildings, or through procedural generation of road networks. This flexibility facilitates the creation of environments ranging from small neighborhoods to entire cities.
Sim4CV is particularly suited for vision-based research tasks such as Simultaneous Localization and Mapping (SLAM), visual odometry, and object detection and recognition. It supports the automatic generation of synthetic datasets with ground truth data, including pixel-level segmentation, bounding boxes, class labels, and depth maps. The engine allows for multi-view rendering, stereoscopy, structure-from-motion studies, and dynamic view augmentation, with the ability to programmatically position multiple cameras attached to actors and update them at each frame.

The core logic of Sim4CV is written in C++, taking full advantage of Unreal Engine's capabilities for real-time rendering, physics simulation, and object control. Additional integration is supported through Python and MATLAB/Simulink, which can be used for simulation control, data analysis, and interfacing with machine learning frameworks. For instance, researchers can run batch experiments, extract large datasets, or connect real-time tracking algorithms via the external API. [42]

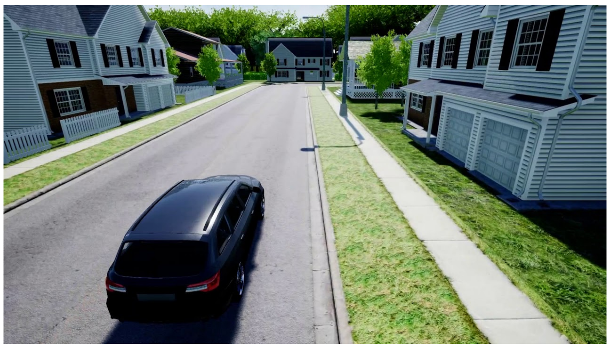


Figure 2.20: Sim4CV Environment [43]

Sim4CV leverages the advanced capabilities of Unreal Engine to provide high-quality graphics and accurate physics simulation. The core of Sim4CV's logic is written in C++, the native language of Unreal Engine. This enables high performance in real-time simulation, direct control of physics, sensors and object dynamics, and advanced integration of control algorithms. Another programming language, although not the main one, is Python, which can be used in external tools for analysis of simulation-generated data, external control of simulation via API, and integration with computer vision models (e.g., TensorFlow, PyTorch). An example of use is extracting datasets or running batch experiments from externally. As the simulators described above, external (hence non-native) integration can be done with Matlab/Simulink for analysis and control. [42]
The simulator provides a robust API that facilitates interaction with the virtual environment, offering control over vehicle movement, actor behaviors, environmental parameters, and more. This makes it possible to embed Sim4CV into broader research pipelines and integrate it with existing computer vision tools or robotic platforms.
Additionally, Sim4CV supports a wide range of input devices, such as flight joysticks, racing wheels, and RGB-D sensors, enabling natural human interaction with the virtual world. This also opens the door for applications in motion capture, where human gestures can be synchronized with the visually and physically rendered simulation.
In summary, Sim4CV is a versatile and powerful simulator tailored for the computer vision community, enabling the development, testing, and validation of perception algorithms in lifelike and controllable environments. Its focus on realism, dataset generation, and integration with popular computer vision tools makes it a valuable platform for autonomous driving research, UAV tracking, and other advanced AI applications. [42] [43]

2.5.4 Microsoft AirSim

Microsoft
AirSim

Figure 2.21: Microsoft AirSim Logo [44]

Microsoft AirSim is an open-source simulator developed by Microsoft Research for artificial intelligence and robotics research, focused on the simulation of autonomous vehicles such as drones and cars. Built initially on Unreal Engine and later extended to support Unity, AirSim provides realistic virtual environments for testing and developing perception, planning and control algorithms. [45]
AirSim provides a comprehensive set of APIs that allow interaction with the simulator to control vehicles, retrieve data from sensors, and customize the simulation environment. These APIs are accessible via Python and C++, providing flexibility for developers to integrate with various frameworks and tools. For example, the API can be used to control a drone, obtain images from simulated cameras, and access sensor data such as LiDAR and IMU. [46]
AirSim supports a variety of input devices to control simulated vehicles, including keyboards, Xbox controllers, and joysticks. This variety allows users to choose the control method that best suits their needs during simulation.
Because of its modular architecture and well-documented API, AirSim can be integrated with other tools and platforms. For example, AirSim can be used in conjunction with real systems to test algorithms in simulated scenarios before implementing them on physical hardware. In addition, AirSim supports integration with machine learning frameworks, facilitating the development and testing of artificial intelligence models. [47] [48]
AirSim offers a wide range of simulated sensors that can be mounted on vehicles, including:

- **RGB cameras**: provide color images of the simulated environment. [49]
- **LiDAR**: generates 3D point clouds for depth perception. [46]
- **IMU**: provides data on acceleration and angular velocity.
- **GPS**: simulates global location data. [50]
- **Depth sensors**: provide information on distance to surrounding objects.

These sensors allow testing of perception and navigation algorithms in a controlled, realistic environment.
AirSim allows users to create and customize simulation environments using Unreal Engine or Unity. Existing scenarios can be modified or new ones created to meet specific research or development needs. In addition, AirSim offers functionality to control environmental conditions, such as time of day and weather conditions, allowing algorithms to be tested in different situations. [45]

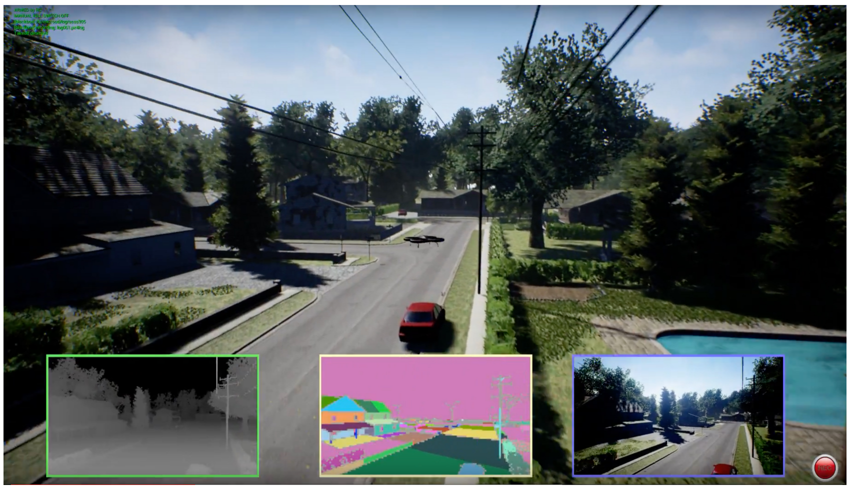


Figure 2.22: AirSim Environment [49]

2.5.5 CarMaker

Figure 2.23: CarMaker Logo [51]

CarMaker is a simulation platform developed by IPG Automotive for virtual testing of vehicles and driver assistance systems. Designed to support the development, calibration, testing and validation of automotive systems in realistic scenarios, CarMaker provides a versatile environment for applications ranging from autonomous vehicles to e-mobility. [51]
CarMaker provides several programming interfaces (APIs) that facilitate integration with other tools and customization of simulations. Among them, the Python API enables test automation and control of simulations via scripts. In addition, CarMaker supports integration with Simulink through the CarMaker for Simulink (CM4SL) add-on, allowing co-simulation and development of complex control models. [52] [53]
The platform is compatible with various input devices, including steering wheels and pedals, to support Driver-in-the-Loop (DIL) applications. This feature allows human drivers to interact directly with the simulation, enhancing the realism of testing.
CarMaker is designed to integrate with a wide range of external tools and frameworks. It supports communication with Robot Operating System (ROS), facilitating the development of robotic applications and autonomous vehicles. In addition, integration with simulation environments such as VISSIM for traffic co-simulation is possible.
The platform offers a wide range of virtual sensors, including cameras, LiDAR, radar, and ultrasonic sensors. These sensors can be configured and parameterized to replicate the specifications of real sensors, enabling accurate testing of perception systems. [51]

Figure 2.24: CarMaker Environment [54]

In addition, IPG Automotive provides comprehensive support to CarMaker users through dedicated forums, detailed documentation, and online tutorials. The active community facilitates knowledge exchange and troubleshooting assistance. [51]

2.5.6 SCANeR

Figure 2.25: SCANeR Logo [55]

SCANeR is an advanced software platform developed by AVSimulation for automotive simulation. It offers tools for the design, validation and training of advanced driver assistance systems (ADAS), autonomous vehicles and human-machine interactions (HMI). The SCANeR development environment is modular, allowing users to customize and configure various aspects of the simulation, including vehicle models, road environments, and traffic scenarios. [55] [56]
SCANeR is designed to integrate with a variety of external tools and platforms. For example, it supports the import of road geometries via the RoadXML format and can be linked to graphics engines such as Unreal Engine for advanced visualization. In addition, SCANeR offers an API that allows integration with third-party software, facilitating the extension of its functionality and interoperability with other systems.
The platform supports various external input devices, including keyboards, mice, and joysticks, enabling more realistic and immersive interaction during simulations. For example, joysticks can be configured to control simulated vehicles, enhancing the user experience. Configuration of these devices is done through software settings, where users can map specific actions to desired hardware controls.
One of SCANeR's distinctive features is its ability to simulate a wide range of automotive

sensors, such as radar, lidar and cameras, enabling in-depth testing and validation of ADAS systems and autonomous vehicles. In addition, SCANeR allows distributed simulations to be run on multiple machines, facilitating the integration of complex models and the development of large-scale scenarios.

Newer versions of SCANeR, such as 2024.2, have introduced significant improvements

Figure 2.26: SCANeR Simulation [55]

in simulation engine performance. For example, there have been efficiency gains in computation time for various modeled sensors, with reductions in total time of up to 46% compared to previous versions. These improvements underscore AV Simulation's ongoing commitment to providing a more efficient simulation experience. [55]

Chapter 3

Vehicle Models

3.1 Models Overview

Physical vehicle models are fundamental tools for understanding and analyzing the dynamic and kinematic behavior of an automobile as it moves. These models are essential for designing and optimizing the performance of vehicles, improving their stability, maneuverability, comfort, and safety. In modern automotive design, the use of mathematical models and simulations has become crucial for predicting vehicle behavior under various driving conditions and for developing advanced driver assistance technologies.
Among the main models used, kinematic and dynamic models play a central role.

- **Kinematic models** They describe vehicle motion as a function of trajectory and velocities, without directly considering the forces involved.
  Kinematic models are mainly divided into two types: the **Single-track Kinematic Model** and the **Dual-Track Kinematic Model**.

- **Dynamic models** They consider the forces, moments and accelerations acting on the vehicle. Starting with simple, linear models, it is possible to arrive at more complex models that take into account nonlinear phenomena and interactions between different physical factors, such as longitudinal and lateral load transfer and tire behavior.
  Dynamic models, on the other hand, are more complex and include the **Single-Track** and **Dual-Track Dynamic Models**. These models describe vehicle motion in detail, including not only the trajectory, but also the effects of longitudinal and lateral forces, roll, and interactions between the vehicle and the road. In particular, high-speed dynamic models (e.g., during high-speed cornering) take into account the effects of loading and weight transfer, which affect the distribution of forces between the front and rear wheels, changing vehicle stability. This type of analysis is crucial for the design of high-performance vehicles, such as racing cars or sports vehicles, which must maintain optimal stability even under high stresses.
  A key part of dynamic models involves the analysis of longitudinal and lateral vehicle dynamics. **Longitudinal dynamics** refers to vehicle behavior in relation to acceleration, braking, and speed management, while **Lateral dynamics** concerns vehicle behavior during cornering and lateral maneuvers, such as oversteer and understeer.
  These aspects are also affected by *load transfer*, which can alter the distribution of forces on the tires and thus affect vehicle traction and stability, both under braking and acceleration.

Another key factor is *roll*, which describes the rotation of the vehicle about its transverse axis, a phenomenon that has a direct impact on cornering stability and the interaction between the suspension and the tires. Rolling is particularly significant during sharp maneuvers or high-speed cornering, and is analyzed in advanced dynamic models to predict vehicle behavior under extreme conditions.
Moreover, belonging to both Lateral and Longitudinal fields, there are also the **Tire models**, essential for simulating vehicle behavior in response to frictional forces between the tires and the road surface.

In the next sections some of them (only Dynamical ones) will be described in detail: **2DOFs Longitudinal model**, **Single Track model** and **Tire Lateral models** (Linear and Non-Linear Pacejka), which are the only ones that are used in this thesis work.

3.2 2DOFs Longitudinal Model

Before describing the model, we have to recall some important concepts about the vehicle architecture (the most simple will be described) and the necessary assumptions. This model was used for the **Dispatching** function.

3.2.1 Longitudinal Vehicle Architecture

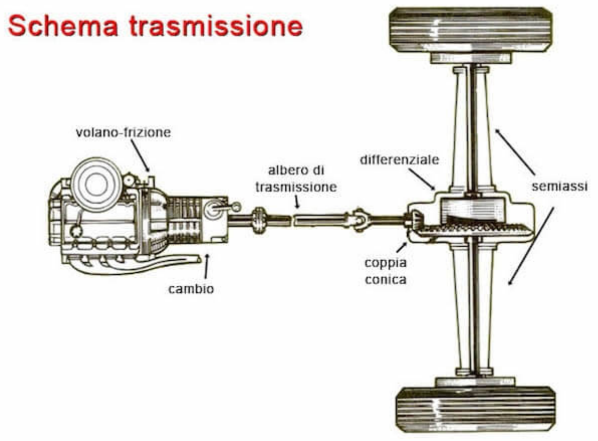


Figure 3.1: ICE-based Vehicle Architecture [57]

This is a simplified outline of the model, considering an ICEV. Here are its main elements:

1. **ICE (Internal Combustion Engine)** This is the leftmost part of the schematic. It generates the power/torque needed for motion. This is done in several stages: *Intake* (the fuel is injected into the engine), *Compression* (the piston rises upward, compressing the mixture and increasing the temperature), *Combustion* (the fuel is burned, this is the phase with the highest pressure and temperature), *Expansion* (the piston descends downward and the pressure and temperature are drastically reduced) and *Exhaust* (the mixture is ejected from the engine). As engine, we can have an Electric Motor powered by a battery (Electric Vehicle architecture), or both an ICE and an Electric Motor (Hybrid architecture).

2. **GB (GearBox)** It is the part immediately to the right of the ICE. It is used to modulate the torque and speed transmitted to the wheels. Depending on the gearbox (manual or automatic), we have different gear ratios:

   - **Manual** The manual transmission is the traditional one, where the driver directly decides when and how to shift gears, using a clutch pedal and a shift lever. In this system, the driver must depress the clutch, move the lever, and then release the clutch, coordinating these movements to shift from one gear to another.
     They usually have 5 or 6 forward gears (transmission ratios > 0), plus reverse (transmission ratio < 0) and neutral gear (transmission ratio $= 0$). More sporty or recent models can go as high as 7 forward gears (e.g., some Porsches or BMWs). So up to 9 gears in total.
   - **Automatic** The automatic transmission handles the shifting between gears by itself, without the driver having to use a clutch or move the lever to shift. This makes driving more comfortable, especially in traffic.
     There are different types of automatic transmissions, which work in different ways:
     (a) *Traditional automatic transmission (with torque converter)* This is the most common type in the US. It uses a torque converter instead of a clutch and changes gears according to engine speed and load.
     It typically has 6, 8 or 9 forward gears, but goes up to 10 gears in high-end models (e.g., Ford, Mercedes, Lexus).
     (b) *Robotic (or automated manual) transmission* This is a manual transmission to which electronic actuators have been added to manage clutch and shifting automatically.
     More energy efficient than a conventional automatic. It usually has 5 or 6 forward gears.
     (c) *Dual-clutch transmission (DSG, DCT, PDK, etc.)* This is a very fast transmission that uses two separate clutches: one for odd gears, one for even gears. This allows almost instantaneous gear changes, which are very popular in sports cars and premium models.
     It usually has 6 or 7 forward gears, but it also goes up to 8 gears.
     (d) *CVT (Continuously Variable Transmission)* This is a continuously variable transmission, widely used in hybrid cars (such as Toyotas) and some city cars. It has no fixed gears, but continuously varies the ratio to keep the engine in the optimal efficiency zone.
     Technically has no fixed number of gears, but some models simulate 6 or 7 virtual ratios to improve driving feel.

   In the model that will be used for simulations, a Traditional automatic transmission is considered, with only forward gears.

3. **Drive Shaft**: transfers motion from the transmission to the differential.

4. **Differential**: distributes torque to the driving wheels, allowing them to turn at different speeds during cornering.

5. **Half-shafts**: distribute the torque from the differential to the driving wheels.

6. **Wheels**: receive the drive torque and allow the vehicle to move.

This type of configuration is typical of front-engine, rear-wheel-drive vehicles.

3.2.2 Model: Assumptions and Equations

The Assumptions of the model are:

1. All the parts of the vehicle are **rigid** (the compliances of each element are neglected).
2. Clutch is not considered in the equations.
3. Slip at tire ground contact is neglected.
4. Flat road.

Equation of the model:

$$m_e a_x = F_x - F_{res} \tag{3.1}$$

where:

- $\boldsymbol{a_x}$: Longitudinal Acceleration.
- $\boldsymbol{m_e}$: Equivalent Mass of the vehicle
- $\boldsymbol{F_x}$: Longitudinal Force applied on the vehicle, making it moving. It is the total Longitudinal Force applied on the wheels.
- $\boldsymbol{F_{res}}$: Resisting forces linked to the vehicle motion.

About $\boldsymbol{F_x}$, we distinguish into 3 cases, according to the value of the longitudinal acceleration:

- $\boldsymbol{a_x > 0}$: $\boldsymbol{F_x}$ is a **Traction** Force.
- $\boldsymbol{a_x < 0}$: $\boldsymbol{F_x}$ is a **Braking** Force.
- $\boldsymbol{a_x = 0}$: $\boldsymbol{F_x = F_{res}}$. The Resultant Force is 0, so the vehicle is traveling at constant speed, or it is stationary.

The Traction force can be evaluated considering the Torque flux from the engine to the wheels. Its expression can be arranged as:

$$F_x = \frac{h_a T_{e_{max}} \tau \eta}{R} \tag{3.2}$$

where:

- $\boldsymbol{ha \cdot T_e max}$ is the product between the **throttle** ($\boldsymbol{h_a}$) and the Maximum Torque of the engine ($\boldsymbol{T_e max}$). This product represents the Actual Torque of the engine.

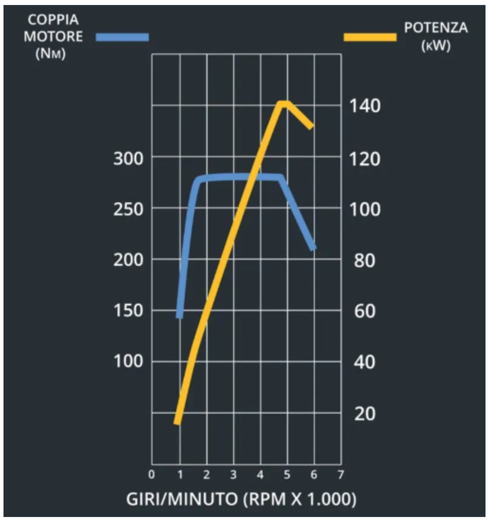


Figure 3.2: Engine Torque vs Engine Speed [58]

As it can be seen from the plot, the Power and the Torque of the vary with the engine rotational speed; but, at the same time, it depends on the throttle. In fact, when it is fully opened, the torque is maximum. So, we can say: $\boldsymbol{h_a = h_a(\omega_e)}$

- $\boldsymbol{\eta}$: Total Transmission Efficiency. In general, it is the product between the gearbox efficiency and that of the differential.
- **R**: Wheel Loaded Radius
- $\boldsymbol{\tau}$: Total Transmission Ratio. It is the product between the Final Ratio $\boldsymbol{\tau_f}$ (constant, taking into account differential, semi-shaft, and so on), and the GB Transmission Ratio $\boldsymbol{\tau_g}$ (a constant value depending to the actual gear). They are evaluated as follows:
  - **GB Ratio**:

$$\tau_g = \frac{EngineRotationalSpeed}{TrasmissionShaftRotationalSpeed} = \frac{\omega_e}{\omega_t} \tag{3.3}$$

  - **Final Ratio**:

$$\tau_f = \frac{TrasmissionShaftRotationalSpeed}{WheelRotationalSpeed} = \frac{\omega_t}{\omega_w} \tag{3.4}$$

Moreover, we can express ω_w in function of V_x:

$$w_w = \frac{V_x}{R} \tag{3.5}$$

Rearranging this equation, and the expressions of $\boldsymbol{\tau_f}$ and $\boldsymbol{\tau_g}$, we obtain:

$$w_e = \frac{V_x \tau}{R} \tag{3.6}$$

So we can say that $\boldsymbol{\omega_e}$ is directly proportional to $\boldsymbol{V_x}$ (the other terms are constant depending only to the actual gears).

Thanks to this relationship between the longitudinal speed of the vehicle and the rotational speed of the engine, we can say: $\boldsymbol{h_a = h_a(V_x)}$
Another term that varies depending on whether the vehicle is in traction or braking, is the Equivalent Mass $\boldsymbol{m_e}$. It is related to the kinetic energy $\boldsymbol{E_k}$ of the vehicle. During the Traction condition:

$$E_k = \frac{1}{2}mV_x^2 + \frac{1}{2}J_e(\frac{V_x\tau}{R})^2 + \frac{1}{2}J_t(\frac{V_x\tau_f}{R})^2 + \frac{1}{2}nJ_w(\frac{V_x}{R})^2 \tag{3.7}$$

Rearranging the expression in order to obtain the equation in the form $\boldsymbol{E_k = \frac{1}{2} \cdot m_e \cdot V_x^2}$, and getting V_x^2 in evidence, we obtain the following formula, valid in Traction condition:

$$m_e = m + J_e(\frac{\tau}{R})^2 + J_t(\frac{\tau_f}{R})^2 + \frac{nJ_w}{R^2} \tag{3.8}$$

where:

- **n**: Number of wheels.
- $\boldsymbol{J_e}$: Moment of inertia of the engine, and of all the parts rotating at the same speed of the engine.
- $\boldsymbol{J_t}$: Moment of inertia of the transmission shaft, and of all the parts rotating at the same speed of the transmission shaft.
- $\boldsymbol{J_w}$: Moment of inertia of the wheels, and of all the parts rotating at the same speed of the wheels.
- **m**: Mass of the vehicle. [59]

In Braking conditions, the Longitudinal force can be evaluated as follows:

$$F_x = -\frac{h_b \sum_{i=n}^{4} T_{b_{max_i}}}{r} \tag{3.9}$$

In this case, $\boldsymbol{h_b}$ is the **Brake**, and, as h_a, varies between 0 and 1.
$\boldsymbol{T_{b_{max_i}}}$ is the Maximum Brake Torque applied on the generic wheel i. [59]
About the equivalent mass in Braking conditions, it is assumed that the whole kinetic energy is dissipated through the brakes at the wheels. So:

$$m_e = m + \frac{nJ_w}{R^2} \tag{3.10}$$

About the Resistance Force $\boldsymbol{F_{res}}$, it is the sum of 3 contributions:

1. **Rolling Resistance**:

$$F_r = C_r mg \tag{3.11}$$

(C_r is the Rolling Resistance Coefficient and g is the gravity acceleration). The assumption is that C_r is constant, since it has a very small variation with respect to the vehicle speed to be considered negligible. The following plot shows this concept:

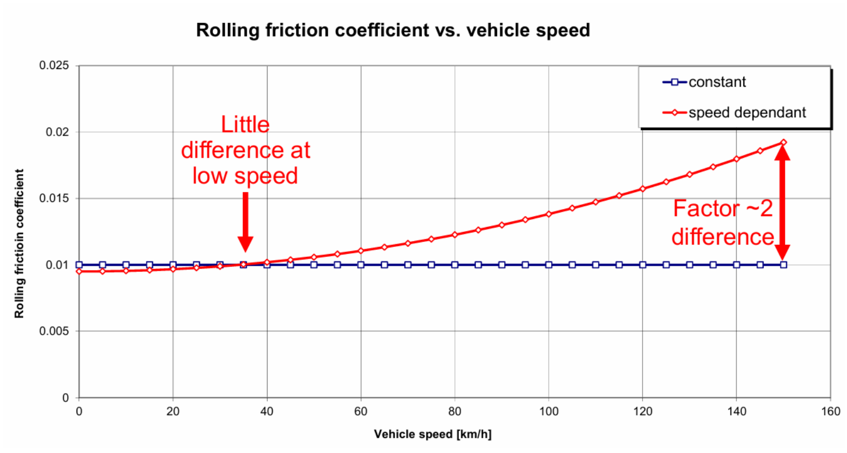


Figure 3.3: Rolling Resistance Coefficient vs Speed [60]

Looking at the plot, it can be seen that C_r is about constant for moderate speeds.

2. **Aerodynamic Resistance**:

$$F_a = \frac{1}{2}\rho A C_x V_x^2 \tag{3.12}$$

where ρ is the air density, A is the Vehicle Frontal Area and C_x is the Aerodynamic Drag coefficient. It varies quadratically with the speed.

3. **Slope Resistance** (neglected due to Flat Road assumption).

3.2.3 Rolling and Aerodynamic Resistances

In the last Section, the different resistances to vehicle motion were described.

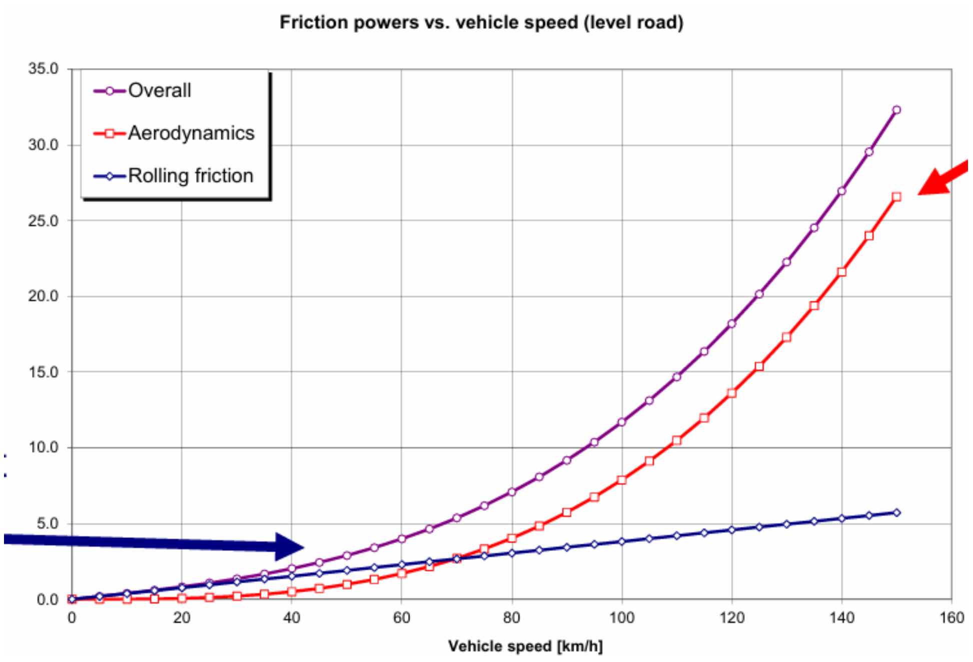


Figure 3.4: Rolling vs Aerodynamic [60]

In this plot only Aerodynamic and Rolling Resistance are considered (flat road assumption). The Rolling resistance is prevalent at low speeds, while the Aerodynamic one presents a major influence to the total at higher speeds.
There is a point in the plot, with a certain vehicle speed, in which these two resistances are equal in values; the velocity of that point is called **Transition Speed**.
Its value can be evaluated by equalizing F_r and F_a:

$$C_r mg = \frac{1}{2}\rho A C_x V_t^2 \tag{3.13}$$

Solving the equation we obtain:

$$V_t = \sqrt{\frac{2C_r mg}{\rho A C_x}} \tag{3.14}$$

Its value is typically in the range 60-80 $\frac{km}{h}$. [60]

3.3 Single Track Model

To describe the lateral dynamics of a vehicle and the interaction between lateral-longitudinal dynamics, we must first describe the vehicle's main reference systems and its degrees of freedom.

3.3.1 Vehicle Global and Local Reference Systems

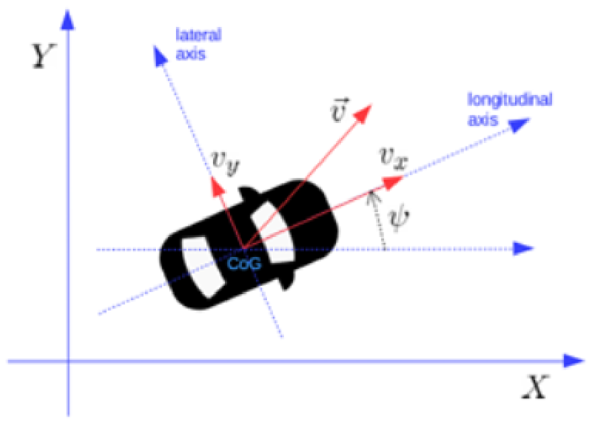


Figure 3.5: Global and Local Reference Systems [61]

From this scheme, it is possible to see the two main reference systems, necessary to describe vehicle dynamics:

- **Global Reference System** It is the 2D Reference system with axes X and Y.
- **Local Reference System** System centered in the vehicle's CoG. In the last image, its x and y axes are called, respectively, *longitudinal axis* and *lateral axis*. More in detail:

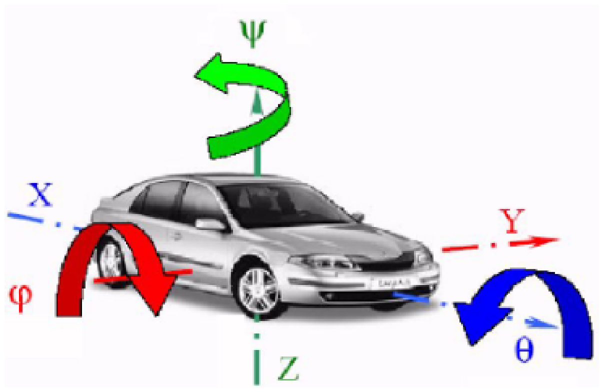


Figure 3.6: Local Reference System [62]

From this last image, we can see the vehicle's **6 DoFs**:

1. **Surge** (Translation along x): forward/backward movement.
2. **Sway** (Translation along y): lateral displacement.
3. **Heave** (Translation along z): lifting/lowering.
4. **Pitch** (Rotation around x): longitudinal tilt (θ).
5. **Roll** (Rotation around y): lateral tilt (ϕ).
6. **Yaw** (Rotation around z): horizontal rotation (ψ). Moreover, it represents the angle between the Longitudinal axis of the Global Reference Frame and that of the Local Frame.

There are also more complete models considering 10 DoFs (including Suspensions' DoFs), and simplified ones, with only 3 DoFs (such as the DSTM).

3.3.2 DSTM: Assumptions and Equations

Assumptions of this model:

- $\boldsymbol{F_{x_2} = 0}$ (Front drive).
- $\boldsymbol{\delta_2 = 0}$ (Only front steering wheels).
- $\boldsymbol{F_{y_i} = -C_i \cdot \alpha_i}$ (Tires Linear Characteristics).
- $\boldsymbol{\alpha_i\ \beta_i\ \delta_i\ \psi}$ very small.
- $\boldsymbol{V_G = const.}$
- Both track (**t**) and vehicle length (**l** = a + b) are negligible with respect to the curvature radius (**R**). Due to this assumption, the track is negligible with respect to the curvature radius, so we consider only 1 wheel per axle instead of 2.

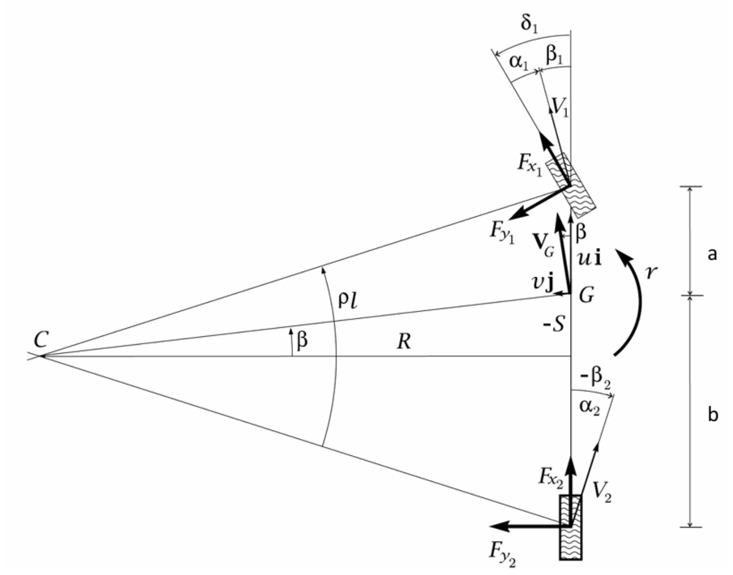


Figure 3.7: DSTM [63]

The following are the parameters and variables present in the scheme:

- $\boldsymbol{F_{x_1}}$ $\boldsymbol{F_{x_2}}$: Tires Longitudinal Forces.
- $\boldsymbol{F_{y_1}}$ $\boldsymbol{F_{y_2}}$: Tires Lateral Forces.
- $\boldsymbol{\alpha_1}$ $\boldsymbol{\alpha_2}$: Tires Side Slip Angles.
- $\boldsymbol{\beta_1}$ $\boldsymbol{\beta_2}$: Tires Slip Angles.
- $\boldsymbol{\delta_1}$: Front Steering Angle
- $\boldsymbol{\beta}$: Vehicle Slip Angle.
- $\boldsymbol{V_1}$ $\boldsymbol{V_2}$: Wheels Speeds.
- $\boldsymbol{V_G}$: Vehicle Speed.
- $\boldsymbol{u}$ $\boldsymbol{v}$: respectively Vehicle Longitudinal and Lateral Speeds.
- $\boldsymbol{r}$: Yaw Angle (in general indicated as $\boldsymbol{\psi}$).
- $\boldsymbol{a}$ $\boldsymbol{b}$: respectively CoG-front axle and CoG-rear axle distances.
- **R**: Curvature Radius.
- **G**: CoG of the vehicle.
- **C**: Center of Curvature.

By doing the equilibrium of forces around x (15), y (16) axes and that of moment around G on z axis (17), we obtain the following equations:

$$\dot{u} - v\dot{\psi} = \frac{1}{m}\left(F_{x_1}\cos(\delta_1) + F_{y_1}\sin(\delta_1)\right) \tag{3.15}$$

$$\dot{v} + u\dot{\psi} = \frac{1}{m}\left(F_{y_1}\cos(\delta_1) + F_{y_2} - F_{x_1}\sin(\delta_1)\right) \tag{3.16}$$

$$\ddot{\psi} = \frac{1}{J_z}\left(aF_{y_1}\cos(\delta_1) - bF_{y_2} + aF_{x_1}\sin(\delta_1)\right) \tag{3.17}$$

where:

- $\boldsymbol{\dot{u} - v\dot{\psi}}$ is the CoG Longitudinal Acceleration, evaluated with respect to the *Global Reference Frame* (see Figure 31). To evaluate it (considering v_x = u and v_y = v):

$$\dot{X} = u\cos(\psi) - v\sin(\psi) \tag{3.18}$$
$$\dot{Y} = u\sin(\psi) + v\cos(\psi) \tag{3.19}$$

  These two are the equations of velocities allowing to pass from the Global to the Local Reference Frame in both x and y directions. The Longitudinal acceleration $\ddot{X}$ is the derivative of $\dot{X}$ in time, so:

$$\ddot{X} = \dot{u}\cos\psi - u\dot{\psi}\sin\psi - \dot{v}\sin\psi - v\dot{\psi}\cos\psi \tag{3.20}$$

  Moreover, under the assumption of small ψ, cos and sin terms can be considered as: $\cos\psi = 1$ and $\sin\psi = 0$, obtaining:

$$\ddot{X} = \dot{u} - v\dot{\psi} \tag{3.21}$$

- $\boldsymbol{\dot{v} + u\dot{\psi}}$ is the CoG Lateral Acceleration. It can be evaluated in the same way as the Longitudinal one, starting from $\dot{Y}$.
- $\boldsymbol{J_z}$: vehicle moment of inertia along z axis.

Starting from Figure 3.7, an expression of the tires' side-slip angles that relates them to the steering angles of the wheels can be derived:

$$\alpha_1 = \beta_1 - \delta_1 \tag{3.22}$$
$$\alpha_2 = \beta_2 \tag{3.23}$$

The first equation is valid for the front wheel (subscript 1) and the second is valid for the rear one (subscript 2).
As can also be seen from the image, the angle β_i is the inclination of the velocity vector of a wheel with respect to the longitudinal axis of the vehicle. So it can be written as:

$$\beta_i = \arctan(\frac{V_{i_y}}{V_{i_x}}) \tag{3.24}$$

where V_{x_i} and V_{y_i} are the components of the velocities at the tires along the x and y axes, respectively.
Looking further at Figure 3.5, in the tires' center (it is called P_i) the Tire Reference Frame, with axes x_i and y_i, is defined. The local reference frame of the vehicle is aligned to that of the tires. So, V_i (velocity at tire ground contact) can be expressed in function of V_G (velocity at vehicle CoG) as follows:

$$\vec{V_i} = \vec{V_G} + \dot{\psi}\ \times \vec{P_iG} \tag{3.25}$$

where $\vec{PG}$ is the distance between the centers of the two reference frames.
By solving this equation, the expressions of V_i along x_i and y_i axes are found:

$$V_{i_x} = u - y_i\dot{\psi} \tag{3.26}$$
$$V_{i_y} = v + x_i\dot{\psi} \tag{3.27}$$

u and v are the components of V_G along x and y axes, while x_i and y_i are the distances of the tires' centers to the CoG, along vehicle axes. About their values:

- $y_i = 0$ for both tires, because tires' centers are on the longitudinal axis of the vehicle, as the CoG (without the assumption of $t \ll R$, y_i would be equal to $\pm\frac{t}{2}$).

- $x_1 = a$ and $x_2 = -b$ (different signs since the local reference frame is centered in the vehicle CoG, so the front tire is in the positive part of the x axis, while the rear one in its negative part).

By substituting these expressions in the *Equation 3.24* for both the tires, we obtain:

$$\beta_1 = \arctan(\frac{v + a\dot{\psi}}{u}) \tag{3.28}$$

$$\beta_2 = \arctan(\frac{v - b\dot{\psi}}{u}) \tag{3.29}$$

By substituting these expressions in both the *Equations 3.22* and *3.23*, they become respectively:

$$\alpha_1 = \arctan(\frac{v + a\dot{\psi}}{u}) - \delta_1 \tag{3.30}$$

$$\alpha_2 = \arctan(\frac{v - b\dot{\psi}}{u}) \tag{3.31}$$

[64] This model is the one that will be used in the NMPC, as it is simple (even computationally) and, at the same time, takes into account many aspects of the longitudinal, lateral and rotational dynamics of the vehicle.

3.4 Lateral Tire Models

Lateral tire models describe the relationship between the lateral forces generated by a tire and other parameters such as slip angle, track width, and ground characteristics. The two best known models are the Linear and the Pacejka's ones.

3.4.1 Linear Tire Model

It is the simplest and assumes that the lateral force generated by the tire is proportional to the drift angle (slip angle). It works only for small drift angles, where the tire behavior is elastic and linear.

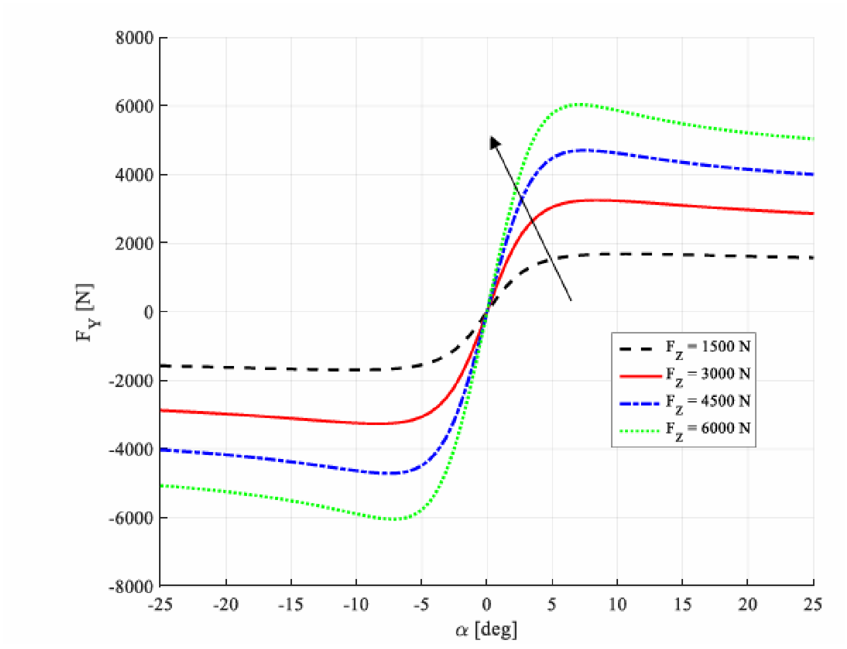


Figure 3.8: Lateral Force vs Side Slip Angle [65]

Its equation:

$$F_y = -C_\alpha \alpha \tag{3.32}$$

where:

- $\boldsymbol{F_y}$: Lateral Force, expressed in [N].
- $\boldsymbol{C_\alpha}$: Cornering Stiffness $[\frac{N}{rad}]$.
- $\boldsymbol{\alpha}$: Side Slip Angle [rad].

The negative sign indicates that the force is opposite to the slip angle.
From the plot, we can also notice that the Tire Lateral Force value increases with the vertical load F_Z, since the Cornering Stiffness increases as vertical load increases, as can be seen from the scheme below:

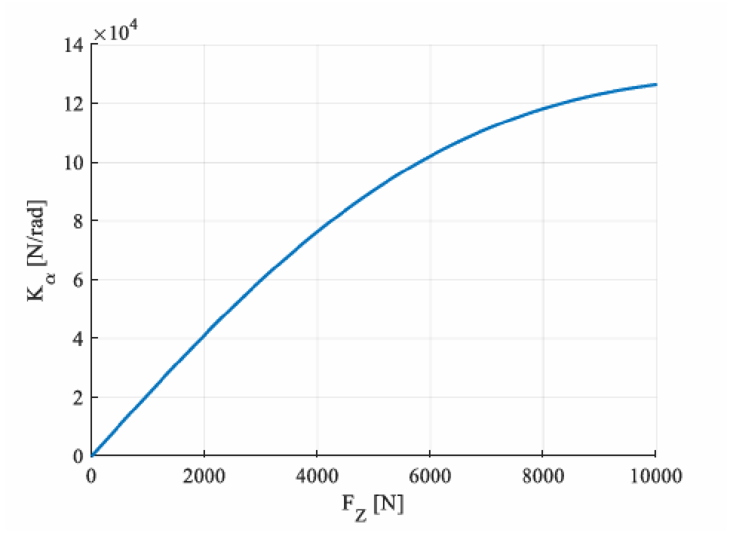


Figure 3.9: Cornering Stiffness vs Vertical Load [65]

K_α is what it is called $\boldsymbol{C_\alpha}$.

3.4.2 Non-linear Pacejka's Model

The Pacejka model is an empirical, nonlinear model that describes with high accuracy the behavior of tires under different conditions (high angles, varying loads, different load conditions, and so on). Its general formula:

$$F_y(\alpha) = D\sin(Catan(B\alpha - E(B\alpha - atan(B\alpha))) \tag{3.33}$$

where:

- $\boldsymbol{F_y}$: Lateral Force, expressed in [N].
- $\boldsymbol{C_\alpha}$: Cornering Stiffness $[\frac{N}{rad}]$.
- **B**: Stiffness factor.
- **C**: Shape factor.
- **D**: Peak factor.
- **E**: Curvature factor.

These terms are estimated experimentally. [66] The following plot is the Pacejka's Lateral Force representation:

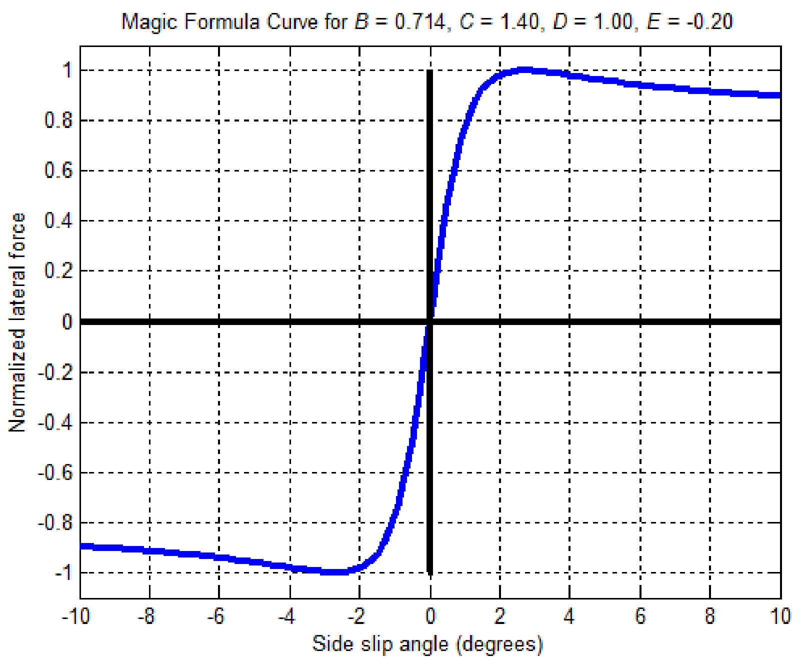


Figure 3.10: Pacejka's Curve [67]

3.4.3 Linear vs Pacejka's: Comparison

The following is a comparison between the two models of the tires, according to different parameters:

1. **Complexity**: the Pacejka's model, being calibrated according to numerous coefficients, has much greater mathematical complexity than the linear model.
2. **Accuracy**: the Pacejka's model is much more accurate, taking into account, thanks to the coefficients, the asphalt conditions, vertical loads, etc.; the linear model, on the other hand, is only valid for small side slip angles, thus being inaccurate.
3. **Typical use**: the linear model is used more for theoretical Analyses and simple controls; the Pacejka's model, being much more accurate, also finds application for more realistic simulations, which take into account many dynamic variations, and also for motorsport. [68]
4. **Experimental data required**: the Pacejka's model, being based on empirical coefficients, requires experimental validations, which is not necessary for the linear model.
5. **Nonlinear behavior**: only the Pacejka's model takes into account the nonlinear behaviors of the tires.

Overall, the linear model is much simpler and has low computational costs; the Pacejka's model is much more accurate and is better suited for more complex dynamic models. However, considering the fact that the system that will be used with the NMPC has a very high computational cost, and the various coefficients of the Pacejka's model would have to be calibrated very often, as the conditions of the simulations change, it is preferable to use the **Linear Model**, which provides greater robustness in this type of application.

Chapter 4

Control Systems

4.1 Control for Autonomous Driving Applications

In the context of autonomous driving, vehicle control systems are organized in a hierarchical structure that enables efficient and safe management of driving operations. This architecture is divided into several functional levels, each with specific tasks.

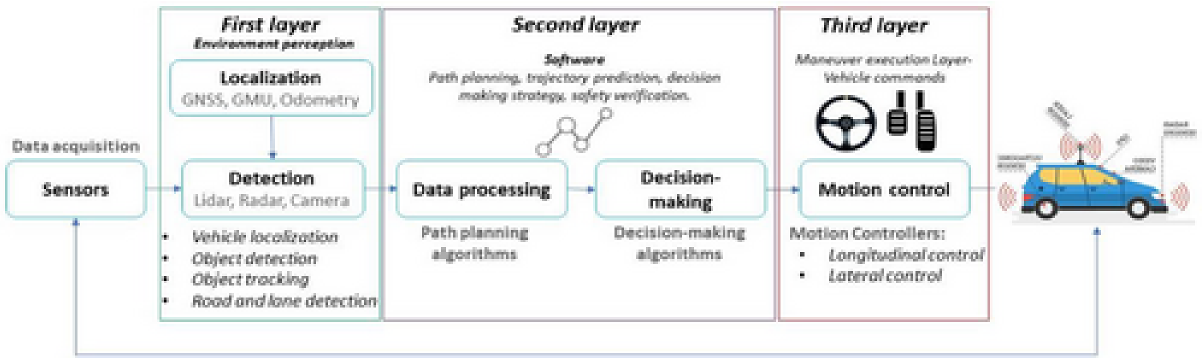


Figure 4.1: Control Layers [69]

1. **Perception** This layer deals with the acquisition and processing of data from sensors such as cameras, lidar, and radar. The goal is to build an accurate representation of the surrounding environment by identifying objects, traffic signals, and other relevant features. This corresponds to the *Data Acquisition* of the scheme (*Sensors* block).

2. **Localization** Using sensor data and high-definition maps, the system determines the precise location of the vehicle on the road. Techniques such as Simultaneous Localization and Mapping (SLAM) are often employed to improve localization accuracy. This is the *First Layer* of the image.

3. **Planning**: In this stage, the system devises an optimal route to the destination, considering factors such as traffic, road conditions, and local regulations. Planning can be divided into:

   - **Strategic planning** Defines the overall route from the starting point to the destination.
   - **Tactical planning** Manages medium-term decisions, such as changing lanes or overtaking.

- **Operational planning** Focuses on immediate actions, such as adjusting speed or steering to avoid obstacles.

This is the *Second Layer* of the image.

4. **Control** This level translates planning decisions into specific commands for vehicle actuators, adjusting acceleration, braking, and steering. Control strategies can be:

   - **Longitudinal control** Manages vehicle speed, maintaining safe distance and adjusting acceleration.
   - **Lateral control** Ensures that the vehicle follows the desired trajectory, keeping in the lane and negotiating turns safely.

   This corresponds to the *Motion Control* (*Third Layer*).

5. **Supervision and Diagnostics** The system continuously monitors its own performance and the status of components to detect anomalies and ensure reliable operation. In case of malfunctions, the system can activate emergency procedures or require human intervention. [69]

Autonomous control systems find application in several areas:

- **Robotaxis** Driverless public transportation services, such as those offered by Waymo, operating in cities such as San Francisco. [70]
- **Delivery vehicles** Autonomous vehicles used to distribute goods, improving logistics efficiency.
- **Autonomous shuttles** Vehicles used in controlled environments, such as university campuses or airports, to transport passengers.

4.2 Feedback Control Architectures and Errors

In control systems, the feedback strategy is a crucial mechanism in which the system continuously monitors its output, and designing a system, called Controller, we want to obtain the desired behaviour of the system. This type of control is essential to ensure stability, accuracy and robustness.

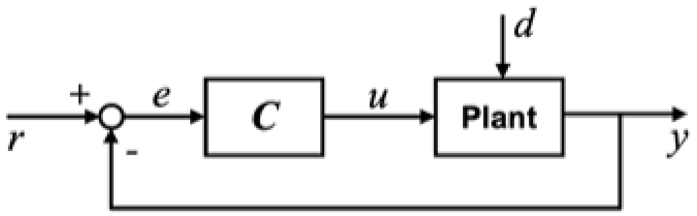


Figure 4.2: Dynamic System Control [71]

From the image, some blocks and parameters can be noticed.
Signals:

- **r**: Reference value of the output, time by time.

- **u**: Command Input, it corresponds to the output of the controller and to the input of the Plant.
- **y**: Output of the Plant, it is the "result" obtained by the system.
- **d**: Disturbance(s), external signals that interfere on the system, changing its behavior.
- **e**: Tracking Error, difference between the reference and the actual value y.

Blocks:

- **Plant**: The dynamic system we want to control.
- **Controller C**: The system we design to make the Plant behaving in the desired way. To design it, we have to solve the so called Control System Problem, consisting in finding the best controller such that $\boldsymbol{y} \simeq \boldsymbol{r}$, taking into account the different signals described above. [71]

The goal of the control is to minimize the Tracking Error e.
There are various types of controllers, and among them the most common are the PID, LQR and NMPC.

4.2.1 PID Controller

PID (Proportional-Integral-Derivative) control is one of the most popular techniques in industrial automation due to its simplicity and effectiveness. It is used in numerous sectors, including automotive, aerospace, power electronics and chemical process automation.

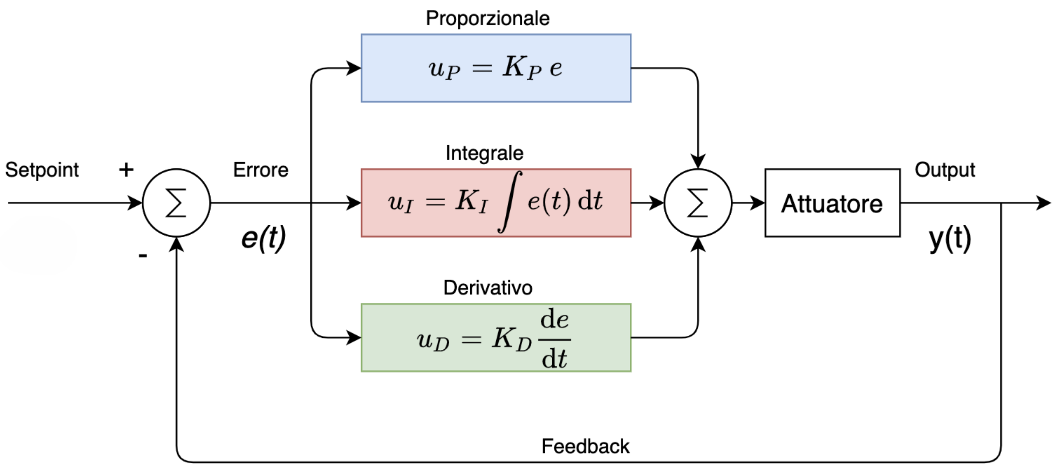


Figure 4.3: PID Control Scheme [72]

It is a feedback control system that regulates a process variable (e.g. temperature, pressure or speed) by comparing the desired value (Setpoint) with the actual measured value. The controller calculates the error between these two values and applies a correction based on three components:

- **P (Proportional)**: corrects the current error. Proportional action produces an output proportional to the current error. An increase in K_P gain reduces the response time, but a too high value can cause instability or oscillation.

- **I (Integral)**: corrects the accumulated error over time. The integral action takes into account the "past history" of the accumulated error, helping to eliminate the actual one. However, a too high integral gain can lead to overshoot and oscillations.
- **D (Derivative)**: anticipates error future based on error variation, improving the stability and dynamic response of the system. However, it is sensitive to noise and is often filtered to avoid unwanted amplification.

PID control law is the following:

$$u(t) = K_P e(t) + K_I \int e(t)\,dt + K_D \frac{de(t)}{dt} \tag{4.1}$$

where:

- **u(t)**: PID output, that is the command applied to the Plant.
- $\boldsymbol{K_P}$ $\boldsymbol{K_I}$ $\boldsymbol{K_D}$: respectively, Proportional Gain, Integral Gain and Derivative Gain.

There are several methods to evaluated these 3 coefficients, and the most common are: Ziegler-Nichols's one (according to K_U coefficient, representative of the Proportional one, and to the oscillation period), Root Locus (used by CARLA) and Tuning by simulations (i.e. Simulink PID Block).
In summary, PID control provides an effective solution to many control problems in industry and beyond. Its ability to combine proportional, integral and derivative actions enables precise and stable control in a wide range of applications. [73] This controller is used for *Carla's Autopilot*.

Root Locus Method

Root Locus is a method for visualizing how the roots of the transfer function (i.e., the poles of the system in feedback) change as gain changes.
Given the system transfer function G(s), the PID Controller in the Laplace Domain is defined as:

$$C(s) = K_P + \frac{K_I}{s} + K_D s \tag{4.2}$$

According to the system (open loop or closed loop), the System Transfer function can be evaluated as:

$$T(s) = C(s)G(s) \quad \text{(Open Loop System Transfer Function)}$$

$$T(s) = \frac{G(s)}{1 + G(s)C(s)} \quad \text{(Closed Loop System Transfer Function)}$$

Then, the Root Locus diagram of T(s) is plotted (i.e. in Matlab using the function *rlocus(T)*) and tune PID parameters in order to obtain the desired behaviour of the system:

- Quick response: leftmost poles (real negative large)
- Less oscillation: complex but more damped poles.

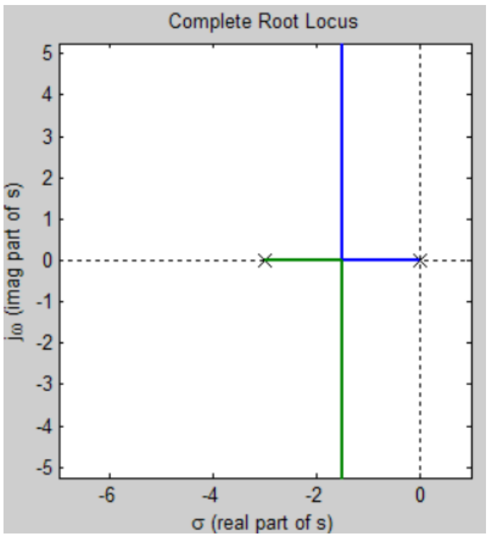


Figure 4.4: Root Locus example [74]

4.2.2 LQR Controller

The Linear Quadratic Regulator (LQR) is an optimal control methodology based on state feedback, widely used for linear time-invariant (LTI) systems. This approach combines dynamic performance and control energy minimization through a balance parameterized by weighting matrices, offering analytical solutions through the algebraic Riccati equation. An LTI system is a system in which both the states x and the command input u don't appear explicitly in function of time (but they can vary according to it). It is described by the equations:

$$\dot{x} = Ax + Bu \tag{4.3}$$

$$y = Cx + Du \tag{4.4}$$

The first equation is the **State Equation** and the second one is the **Output Equation**. The goal of this Controller is to determine a control signal u(t) that minimizes the quadratic cost function:

$$J(u, x) = \int_0^{+\infty} (x^T Q x + u^T R u)\, dt \tag{4.5}$$

where $\boldsymbol{Q} \in \mathbb{R}^{n_x \times n_x}$ (semi-definite positive) and $\boldsymbol{R} \in \mathbb{R}^{n_u \times n_u}$ (definite positive). To minimize this function we have to tune the two matrices Q and R doing a trade off, since:

- **Q** matrix is related to **state signal energy**, and by minimizing it we increase the performance of the system (less overshoot, higher convergence speed and less oscillations).
- **R** matrix is related to **command activity energy**, so to the amplitude of the input signals, reducing energy consumption. [75]

The trade-off is important because, for example, if we have a high value of Q and very low values of R, it will result in a fast system response, but high values of the input commands will be applied.
Practical example: we need to control the operation of an electric motor, whose states are x=[θ; $\dot{\theta}$], so both angular position and velocity of the motor. The input command is

the voltage u=V. If the values of the matrix Q are high, we will have a fast response of the system with a low overshoot, but to do this a very high voltage will be applied, to which an excessive current will correspond, perhaps exceeding the limits of the motor and overheating it.

4.2.3 NMPC

Nonlinear Predictive Control (NMPC) represents one of the most promising control technologies for complex dynamic systems today. Its strength lies in its ability to systematically handle multivariable nonlinear systems, taking into account constraints on the states, inputs and outputs of the system itself ([76] [77] [78] [79] [80]). Unlike Linear Model Predictive Control (MPC), where optimization problems turn out to be convex and generally solvable through quadratic programming [81], NMPC addresses nonconvex optimization problems, which introduces both theoretical and computational challenges [82] [83].
These complexities have necessitated the evolution of MPC toward a nonlinear formulation capable of dealing with more realistic dynamics, nonlinear constraints and performance criteria that are also nonlinear [84]. Both MPC and NMPC require fast and reliable optimization algorithms capable of meeting the time constraints imposed by real-time control. However, for NMPC, the difficulty increases significantly because of the nonconvexity of the problem and the possibility of obtaining multiple local minima, requiring the use of sophisticated algorithms with higher computational costs than linear MPC. During the past decades, significant progress has been carried out in reducing the computational complexity of the NMPC approach ([85] [86] [87] [88] [89]). These advancements have allowed the NMPC implementation also in real-time systems with high sampling rate requirements, see, e.g., [90] [91] [92].
NMPC control operates according to a receding horizon strategy, in which at each sampling instant an Optimal Control Problem (OCP) is solved over a future time horizon, but only the first calculated input is applied. This process is repeated at each time interval.
The three main phases of an NMPC cycle are:

1. **Prediction**: an explicit model of the process is used to predict the behavior of the system over a **prediction horizon $\boldsymbol{T_P}$**
2. **Optimization**: the sequence of controls that minimizes a performance index is calculated.
3. **Application**: the first value of the calculated sequence is applied and the process is repeated at the next step.

Regarding the mathematical formulation of this control algorithm, we consider a nonlinear dynamic MIMO (Multiple Input Multiple Output) system described by the following equations:

$$\dot{x} = f(x, u) \tag{4.6}$$

$$y = g(x, u) \tag{4.7}$$

where $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{n_x}}$ is the state vector, $\boldsymbol{u} \in \mathbb{R}^{\boldsymbol{n_u}}$ is the input vector and $\boldsymbol{y} \in \mathbb{R}^{\boldsymbol{n_y}}$ is the output vector.
The NMPC assumes that the system state is available in real time with a **sampling time $\boldsymbol{T_S}$**, so:

$$x(t_k) = x_k = x(kT_s), \quad t_k = kT_s, \quad k = 0,1,2,\ldots \tag{4.8}$$

If the state is not directly measurable, an Observer or input-output representation is used. At each time t = t_k, both the state and the output of the system are predicted over the whole time interval $\tau = [t, t + T_P]$. T_P is the *prediction horizon*, that corresponds to the number of future instants for which predictions of control states and inputs are calculated through the integration of the equations (3.5, 3.6). Moreover the relationship $T_P \geq T_S$. At any time instant $\tau = [t, t + T_P]$, the output predicted by the NMPC $\hat{y}$ will be in function of the initial state signal x(t) and of the input signal u

$$\hat{y}\Big(\tau\Big) \equiv \hat{y}\Big(x(t), u(t:\tau)\Big) \tag{4.9}$$

where $u(t : \tau)$ corresponds to a generic input in the time interval $[t, \tau]$.

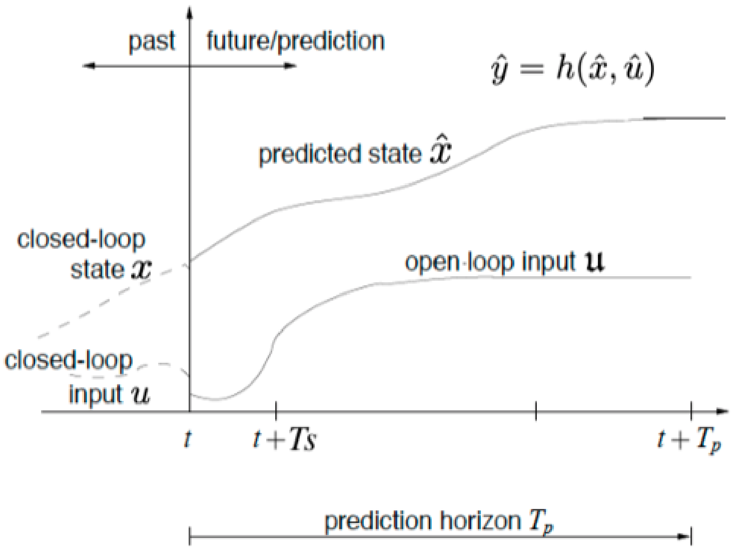


Figure 4.5: NMPC Strategy [93] [94]

At any instant of time we need to find an optimal input $u^*(t : \tau)$ such that the predicted output:

$$\hat{y}(\tau) = \hat{y}(x(t), u^*(t:\tau)) \equiv \hat{y}(u^*(t:\tau)) \tag{4.10}$$

behaves in the desired way in the whole interval τ. Our input u(τ) does not depend on the state in the time interval.
The desired performance is expressed through a cost function that estimates the tracking error and control energy:

$$J(u(t : t + T_p)) \doteq \int_t^{t+T_p} \Big(\|\tilde{y}_p(\tau)\|_Q^2 + \|u(\tau)\|_R^2\Big)\, d\tau + \|\tilde{y}_p(t + T_p)\|_P^2 \tag{4.11}$$

where:

- $\boldsymbol{J(u(t : t + T_p))}$: *Cost Function*, that has to be minimized.
- $\boldsymbol{\tilde{y}_p(\tau) = r(\tau) - \hat{y}(\tau)}$: *Tracking Error* over the time interval.
- $\boldsymbol{u(\tau)}$: Input signal over the interval, representative of the *command activity.*
- $\boldsymbol{\tilde{y}_p(t + T_p)}$: *Final Tracking Error.*

- $||.||_X$: *2-Norm*, weighted according to the generic matrix indicated as X (it can be R, Q or P).
- Q, R and P: *Weighted Matrices*. By tuning them, we decide to give more or less weight to the corresponding terms, taking into account also the trade-off discussed about LQR.

So, at each time instant t = t_k, we have to solve the following optimization problem:

$$u^*(t : t + T_p) = \arg\min_{u(\cdot)} J(u(t : t + T_p)) \tag{4.12}$$

taking into account also the *constraints* that can be on the output and/or state variables (i.e. obstacle and collision avoidance), and/or the command inputs (i.e. interval of possible values).
This type of control is done in real-time, so at each t = t_k, and the solution is usually a local minimum, not a global one (however it is acceptable from a control performance point of view). [93]
The following is the closed-loop scheme of this control algorithm:

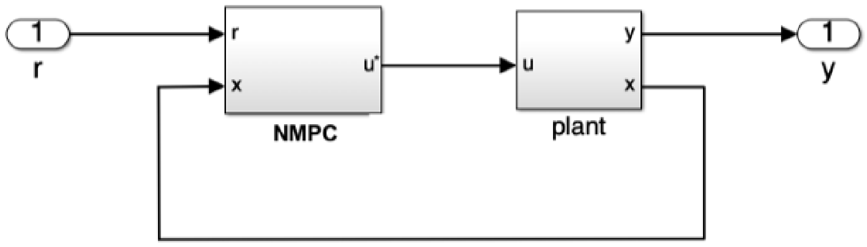


Figure 4.6: NMPC Closed-loop Scheme [93] [94]

OCP resolution techniques can be divided into direct and indirect methods:

1. **Indirect methods** (such as Pontryagin's Maximum Principle) [**<empty citation>**]: less robust and rare in practical applications.
2. **Direct methods**: it discretizes the problem by transforming it into an NLP. Among them:
   - **Single Shooting**: optimizes only the controls, integrating the dynamics.
   - **Multiple Shooting**: optimizes controls and states in multiple intervals.
   - **Direct Collocation**: discretizes equations using collocation points.

As for NLP resolution, they mainly use [95]:

- **SQP (Sequential Quadratic Programming)**: transforms the problem into a sequence of quadratic problems.
- **Interior Point methods**: handle constraints with logarithmic barrier functions.

In the thesis work presented, a Direct Single Shooting approach was used to formulate the OCP and an SQP method was used to solve it.

T_S and T_P Parameters

The choice of the values of sampling time T_S and prediction horizon T_P is crucial, considering a trade-off between performance and computational efficiency of the system. As for T_S, its value depends on the functional level of the system:

1. **High-level Decision Layers** (path planning and trajectory tracking) For this level, sampling times between 100 and 500 ms are adopted, since rapid updates are not required.

2. **Motion Planning and Control Modules** (lateral and longitudinal control). In this case, the sampling time is shorter, between 20 and 100 ms, to ensure smooth and safe control at high speeds.

3. **Perception and Sensory Fusion Modules** The sampling interval is between 10 and 50 ms, to have a constant awareness of the surroundings.

4. **Low-level Actuator Control** (braking, acceleration, steering) For this layer, T_S is extremely short, between 5 and 20 ms.

Another crucial aspect is the choice of the prediction horizon, which is the future time interval over which the controller performs optimization. The length of this horizon significantly affects both the performance of the system and the computational load required.
In general, a longer predictive horizon gives the controller a broader view of the future, allowing for earlier planning of maneuvers. This results in smoother commands, more stable trajectories, and a greater ability to prevent critical situations or obstacles along the way. However, the increase in horizon length also leads to an increase in computational complexity, making the optimization problem more onerous to solve at each sampling instant. In addition, a non-optimal predictive model may introduce errors that, extending over a long horizon, tend to accumulate, reducing the reliability of the decisions made.
In contrast, a shorter T_P significantly reduces the time required to solve the optimization problem, promoting greater controller responsiveness and ensuring real-time execution even on systems with limited computational resources. However, a horizon that is too short could limit the controller's ability to anticipate system evolutions, leading to more impulsive or less stable behavior, especially in scenarios that require medium or long-term planning, such as lane changes on highways or handling large curves.
In conclusion, for the choice of the prediction horizon value, other than the computational cost and the performance, the type of environment and maneuver must be taken into account. The following are the typical values that are adopted:

1. **Urban environment** Short horizons (2-5 s) are preferred to handle obstacles and sudden changes.

2. **Highway** In this case, longer horizons (5-10 s) are adopted, since the traffic is smoother, to favor stability and comfort.

3. **Precision Maneuvers** (i.e. parking) T_S is reduced to 1-3 s to achieve greater accuracy.

4.2.4 Errors: Cross Track Error and Heading Error

In the context of trajectory control of an autonomous or semi-autonomous vehicle, one of the key issues is the proper identification and management of tracking errors with respect to the reference trajectory.

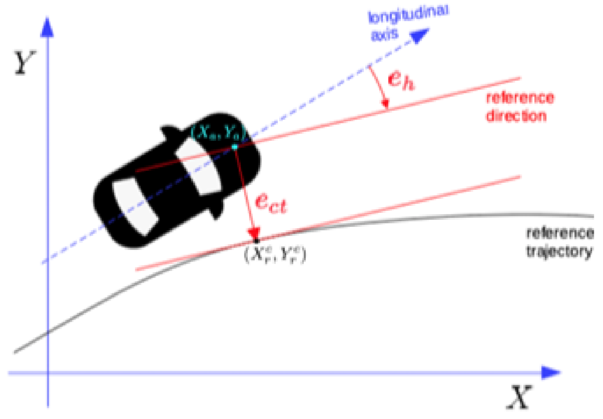


Figure 4.7: Trajectory Errors [61]

In particular, two error measures are crucial for lane keeping systems:

1. **Cross-track Error e_{ct}**: lateral distance between the vehicle and the nearest point on the trajectory.
2. **Heading Error e_h**: difference between the orientation of the vehicle and that of the reference trajectory.

To define these errors, the following notation is introduced:

$$(X_a, Y_a, \psi) \quad \text{(vehicle pose at the front axle)}$$

$$(X_r^c, Y_r^c, \psi_r^c) \quad \text{(point of the reference trajectory closest to the vehicle)}$$

The first one is the point representing the laying of the vehicle at the front axle; the second one is the laying of the point closest to the reference trajectory.
The *Heading Error* formula is:

$$e_h = \psi_r^c - \psi \tag{4.13}$$

The *Cross-track Error* is evaluated as follows:

$$e_{ct} = (Y_r^c - Y_a)\cos\psi_r^c - (X_r^c - X_a)\sin\psi_r^c \tag{4.14}$$

or, equivalently, using the vectorial product between:

$$\boldsymbol{\xi} = \begin{bmatrix} \cos\psi_r^c \\ \sin\psi_r^c \\ 0 \end{bmatrix} \quad \text{(trajectory orientation)},$$

$$\boldsymbol{\rho} = \begin{bmatrix} X_r^c - X_a \\ Y_r^c - Y_a \\ 0 \end{bmatrix} \quad \text{(relative position vector)},$$

$$\boldsymbol{\xi} \times \boldsymbol{\rho} = \begin{bmatrix} 0 \\ 0 \\ e_{ct} \end{bmatrix}$$

The 2-Norm of this vectorial product can be written as $\|\rho\|$ since the two variables are orthogonal. In this way, we can calculate the error:

$$|e_{ct}| = \|\rho\| = \|(X_c^r, Y_c^r) - (X_a, Y_a)\| \tag{4.15}$$

According to the sign of this last error, we know its position with respect to the trajectory:

- $e_{ct} > 0$: vehicle on the right with respect to the reference trajectory
- $e_{ct} < 0$: vehicle on the left with respect to the reference trajectory [61]

In this thesis work, only the *cross track error* will be considered because in the scenarios that will be simulated (mainly straight trajectories or those with small curves) the heading error assumes negligible importance. However, in the presence of tight turns, high-speed maneuvers, or drift dynamics, ignoring this error may lead to instability or incorrect tracking.

Chapter 5

CARLA Environment

5.1 CARLA Modules Overview

This simulator has already been described in the *Section 2.5.1*. In this chapter the description will be focused on its architecture and modules.
The CARLA simulator is based on a scalable client-server architecture. The heart of the simulation resides in the **server**, which handles all the basic aspects such as rendering sensors, simulating physics, updating the state of the virtual world and its actors, and more. For realistic results, it is advisable to run the server on a machine with a dedicated GPU, especially when working with machine learning algorithms.
The **client** side, on the other hand, consists of a series of modules that control the logic of the actors in the simulation and define the conditions of the environment. These operations are made possible through the CARLA API, which is available in both Python and C++. The API serves as an interface between client and server and is constantly evolving, offering new functionality.

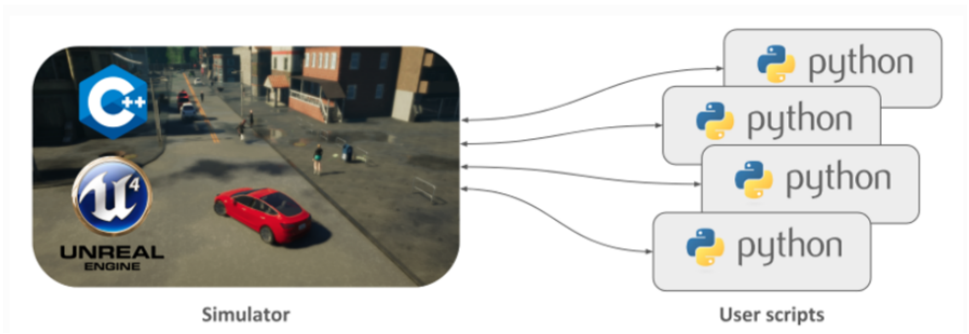


Figure 5.1: CARLA API [96]

About CARLA's main modules, they are summarized below:

- **Traffic Manager**: this is an integrated system that manages the behavior of vehicles not directly controlled by the learning algorithms. This module makes it possible to recreate realistic urban scenarios, simulating credible and consistent traffic.

- **Sensors**: these are critical for gathering information about the surrounding environment. In CARLA, sensors are considered special actors that can be attached to vehicles. The acquired data can be collected and stored for later analysis. The simulator supports different types of sensors, including cameras, radar, lidar, and others.

- **Recorder**: this feature allows the simulation to be recorded at every stage, allowing any instant in time and anywhere in the simulated world to be replayed. This is a very useful tool for tracking and analyzing actor behavior.

- **Integration with ROS and Autoware**: CARLA is designed to be compatible with other development and machine learning environments. For this, it offers a bridge with ROS (Robot Operating System) and supports the implementation of Autoware, thus expanding its possibilities for use.

- **Open assets**: the simulator provides a wide range of assets, such as city maps, customizable weather conditions, and a library of ready-to-use actors. Users can also modify these elements or create new ones by following simple and accessible guidelines.

- **Scenario Runner**: To facilitate the training of autonomous vehicles, CARLA includes a series of predefined scenarios representing different road situations. These runs form the basis of the CARLA Challenge, a competition open to anyone who wants to test their solutions and compare themselves with other participants through a global ranking. [96]

5.1.1 Traffic Manager

Within the CARLA simulator, the **Traffic Manager (TM)** is the module dedicated to controlling vehicles in autopilot mode. Its main purpose is to recreate credible urban traffic conditions, so as to make simulations more realistic and useful for training and validation of autonomous driving systems. The user has the ability to intervene in traffic behavior through a set of customizable parameters, which allows the user to reproduce complex scenarios or particular driving conditions.
TM is implemented on the client side of CARLA and has a modular structure, in which operations are divided into several distinct phases, each with specific tasks and objectives. Each phase is executed on an independent thread, ensuring computational efficiency and orderly data flow management. Communication between phases is via synchronous and unidirectional messages, ensuring that information flows neatly down the operational chain.
One of the most versatile aspects of TM is the ability for the user to modify the behavior of the controlled vehicles. It is possible, for example, to disable speed limit enforcement, force lane changes or activate atypical driving scenarios. This flexibility is critical for training autonomous driving algorithms in nonstandard conditions, helping to make them more robust and adaptable. [97]

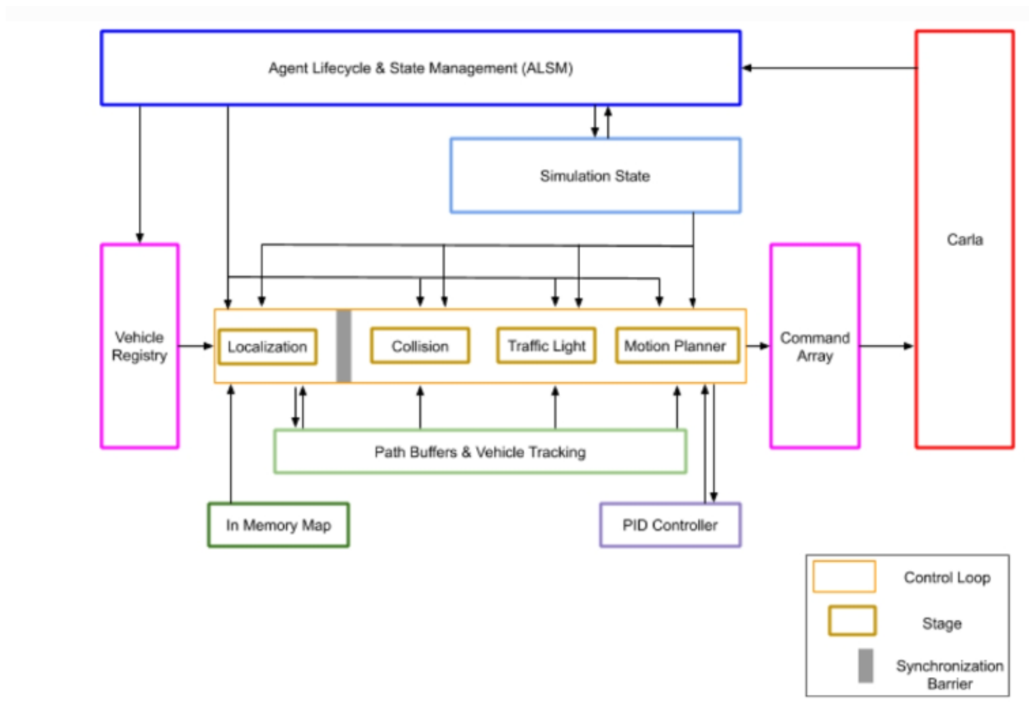


Figure 5.2: Traffic Manager Architecture [97]

The TM architecture is divided into three main phases:

1. **Simulation state management**: This phase is handled by the **Agent Lifecycle & State Management (ALSM)** module, which is responsible for constantly monitoring the presence and state of vehicles and pedestrians in the scene. The module retrieves all information from CARLA's server (it is the only TM component to do so), and keeps it up to date:

   - The list of vehicles controlled by the TM (autopilot)
   - The position, speed, and status data for all actors (including pedestrians)
   - Synchronization with the modules responsible for movement planning and vehicle tracking.

   All this data is stored in a structure called *simulation state*, which serves as a cache memory for the next steps, avoiding additional calls to the server during command computation.

2. **Vehicle motion computation**: The logical heart of TM is found in the **control loop**, a cycle consisting of multiple sequential stages that process the behavior of vehicles in autopilot. Each stage of the loop is completed for all vehicles before proceeding to the next, thus ensuring data consistency within the same simulation frame. The stages are summarized as follows:

   (a) **Localization Stage** In this stage, a dynamic path is constructed for each vehicle based on a simplified map grid, called the In-Memory Map. Decisions near intersections are made randomly, and the generated paths are stored in the **Path Buffer & Vehicle Tracking (PBVT)** module, which allows quick access later.

(b) **Collision Stage** Here, potential collision hazards are analyzed: bounding boxes are extended along the path of vehicles and overlaps between trajectories are identified. In case of danger, the hazard is signaled so that the movement is modified appropriately in the next stage.

(c) **Traffic Light Stage** Similar to the previous stage, hazards related to the presence of traffic lights, stop signs or intersections without signage are assessed. In the latter case, a precedence logic is adopted based on the order of arrival of vehicles ("first-in, first-out").

(d) **Motion Planner Stage** Based on the collected data (position, speed, planned route, and detected hazards), vehicle behavior is determined. A *PID controller* is used to calculate the optimal values for throttle, brake, and steering. The result is a set of commands that will be applied to the vehicle.

3. **Applying commands to the simulation** At the end of the control loop, the generated commands are aggregated into a single frame called the Command Array. This array is sent to the CARLA server in bulk, to ensure that the entire simulation is updated within the same frame. Sending can be done synchronously or asynchronously, depending on the configuration.

In summary, the design of the Traffic Manager in CARLA emphasizes a focus on modularity, synchronization, and extensibility. The ability to control every aspect of traffic behavior-from intersection analysis to vehicle light management-makes it an essential tool for realistic and customized simulations. In addition, the multithreading-oriented architecture allows full use of available computational resources, facilitating smooth simulations even in complex scenarios. [97]

5.1.2 CARLA PID Controller

For vehicle control on CARLA, a Controller is used, which acts on the three input controls: throttle, brake and steering angle. This controller takes as input the waypoints to be followed, constantly updating the current position of the vehicle (x, y, yaw, speed, etc.) and calculates the commands to be sent to the simulator.
As for the **Longitudinal Controller**, theoretically it is handled by a PID, but one could also simply fix the input values of throttle and brake. [98]
For the **Lateral Controller** (steering), it is handled totally by a PID. The vehicle must follow a trajectory, so it must "chase" the given waypoints. To do this, it has to correct the heading error (i.e., how far the vehicle is oriented away from the direction to be followed).
Below there is the python code for calculating the error:

```python
desired_yaw = np.arctan2((waypoints[-1][1] - y), (waypoints[-1][0] -
    x))
error_0 = desired_yaw - yaw
```

Listing 5.1: Heading Error Evaluation

where:

- *desired yaw*: the angle that would join the current position with the last waypoint.
- *error 0*: it is the heading error. [98]

The following is the Python implementation of the PID controller:

```
Kp = 1.12924229948271
Ki = 0.0270272495971814
Kd = 1.88301790267392

a1 = -1
b0 = Kp + Ki + Kd
b1 = -(Kp + 2*Kd)
b2 = Kd

steer_output = b0 * error_0 + b1 * self.vars.error_1 + b2 *
    self.vars.error_2 - a1 * self.vars.steer_1
```

Listing 5.2: PID Controller Implementation

where:

- ***K_P K_I K_D***: PID Controller coefficients. They are evaluated by *Root Locus method*, calculating specifications in terms of maximum overshoot, controller effort and steady state error.
- *a1, b0, b1, b2*: Coefficients, evaluated by PID coefficients algebraic operations, that are multiplied by errors and steering command.
- *steer output*: Steering Command expression.

The steering angle expression is in the form:

$$u[k] = b_0 \cdot e[k] + b_1 \cdot e[k-1] + b_2 \cdot e[k-2] - a_1 \cdot u[k-1] \tag{5.1}$$

where *e[k]* is the error 0, so the actual heading error, and *e[k-c]* (c = 1, 2, 3, ...) is the error value at the previous time instants. The same notation is used for *u[.]* (steering command).
At the end of the cycle, errors and the previous command are saved:

```
self.vars.error_2 = self.vars.error_1
self.vars.error_1 = error_0
self.vars.steer_1 = steer_output
```

Listing 5.3: Updating Variables

In this way, in the next cycle the PID will have access to past errors to calculate the derivative and integral coefficients. [98]

5.1.3 Python-CARLA Interface

In order to interact effectively with the CARLA simulator, it was necessary to use the relevant API in the Python language. However, since each version of CARLA is compatible only with certain versions of Python, it was appropriate to adopt a configuration that would ensure stability and consistency between the two environments, even in view of future updates or modifications. For this reason, it was chosen to work within a dedicated virtual environment.
To this end, Anaconda Navigator, a tool that enables simple and centralized management of virtual environments and associated packages, was employed. Through Anaconda, it was possible to create an ad hoc environment, install and update the necessary libraries for CARLA and Python, and keep everything sorted within a specific directory. Anaconda

also provides a command window (prompt) for direct execution of commands, activation of virtual environments, and execution of scripts. In addition, it includes the Spyder editor, a built-in development environment that allows Python scripts to be edited and tested before executing them via the Anaconda prompt. The workflow is depicted in the Figure below:

spyder
python
ANACONDA
CARLA

Figure 5.3: Python-CARLA Interface [96]

To properly configure communication between CARLA and the Python environment:

Listing 5.4: CARLA Environment establishment

```
conda create --name carla_env python=3.7
conda activate carla_env
pip install numpy jupyter pygame opencv-python
```

In this way the connection is established and the CARLA environment is created. The last one code line is used to install Python modules, necessary to run CARLA scripts.

Listing 5.5: CARLA Example script run

```
cd C:\CARLA_0.9.14\PythonAPI\examples
python nome_codice.py
```

This is used to run the Python example scripts, so first change the directory and then run the code by writing its name.py.
To run a new script that is not an example one, it is necessary to write it using Spider and save it. Then, in the same way as the example scripts, change the directory with that of the new script and run it using that procedure.

5.1.4 Autopilot - Data Gathering

During a simulation in CARLA, the parameters of the actors in the scene can be accessed through the Python API. These parameters include, for example, position, speed, and many other useful information. In this thesis work, the focus is on acquiring data about the ego vehicle, i.e., the main vehicle controlled autonomously via CARLA's Traffic Manager. The data collected during autonomous driving of the ego vehicle are subsequently imported into MATLAB, where they are analyzed to study its dynamic behavior in response to certain inputs. In addition, these data are critical for generating the references needed for the development of the NMPC controller and for evaluating the trajectory errors committed by the PID controller built into CARLA's autopilot system.
To activate the ego vehicle autonomous driving mode, simply run the script "**manual_control.py**", which allows a vehicle to be generated on the map and manually controlled by keyboard. By pressing the "P" key, the vehicle switches to autonomous mode. Alternatively, you can directly enable autopilot from a script by setting the appropriate parameter via the

settings method of the world object in CARLA.
The following scripts belong to the **get_state(self)** function, implemented within the “manual_control.py” code to acquire, at each simulation cycle, a set of data critical for analyzing the dynamic behavior of the ego vehicle during autonomous driving in CARLA.

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

    position = self.player.get_position()
    velocity = self.player.get_velocity()
    transform = self.player.get_transform()

    yaw = transform.rotation.yaw * math.pi / 180
    yaw_rate = self.player.get_yaw_rate().z * math.pi / 180
    acceleration = self.player.get_acceleration()
```

Listing 5.6: get_state Function - 1

The elapsed time since the start of the simulation (in milliseconds) is acquired using *pygame.time.get_ticks()*, and the value is saved in the time list. By using "name.append()", the data are saved in .txt files.
Then the current position, velocity and transformation (position + rotation) of the ego vehicle are acquired.
The yaw angle is converted from degrees to radians. Similarly, the angular velocity around the vertical axis (yaw rate) is obtained, again in radians per second.

```python
psition_data_x.append(position.x)
position_data_y.append(position.y)
position_data_z.append(position.z)

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

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

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

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

Listing 5.7: get_state Function - 2

Data of position, velocity, yaw angle, yaw rate and acceleration are saved separately in their respective lists.
Next, the input commands provided to the vehicle: throttle, brake, gear engaged and steering angle are extracted by using the **self.player.get_control()** function and saved in different lists.

```python
map = self.world.get_map()
```

```
waypoint = map.get_waypoint(
    self.player.get_position(),
    project_to_road=True,
    lane_type=(carla.LaneType.Driving | carla.LaneType.Sidewalk)
)

waypoint_list_x.append(waypoint.transform.position.x)
waypoint_list_y.append(waypoint.transform.position.y)

return (...)
```

Listing 5.8: get_state Function - 3

Using the world map, the closest waypoint to the vehicle's current location is determined. The parameter *project_to_road=True* forces the point to be on a road lane.
Then there is the saving of the coordinates of the reference waypoint x and y.
Then the various variables and lists we entered in the script (time_data, position_data.x, waypoint_list.x, etc.) are returned. These have to be inserted within the *return()*.

```
with open(r'C:\your_directory\inputs.txt', 'w') as file:
    file.write('time, acceleration_x, acceleration_y, steering_data\n')
    for i, j, k, u, v, w, z in zip(time_, acceleration_data_x,
   acceleration_data_y, steering_data, throttle, brake, gear):
    file.write(f'{i}, {j}, {k}, {u}, {v}, {w}, {z}\n')
with open(r'C:\your_directory\CARLA_states.txt', 'w') as file:
    file.write('time, position_x, position_y, velocity_x, velocity_y,
   yaw_data, yaw_rate\n')
    for i, j, k, u, v, z, w in zip(time_, position_data_x,
   position_data_y, velocity_data_x, velocity_data_y, yaw_data,
   yaw_rate):
    file.write(f'{i}, {j} ,{k}, {u}, {v}, {z}, {w}\n')
with open(r'C:\your_directory\gearshift_logic.txt', 'w') as file:
    file.write('time, velocity_x, velocity_y, yaw_data, gear\n')
    for i, j, k, u, v in zip(time_, velocity_data_x, velocity_data_y,
   yaw_data, gear):
    file.write(f'{i}, {j}, {k}, {u}, {v}\n')
with open(r'C:\your_directory\waypoints.txt', 'w') as file:
    for i, j in zip(waypoint_list_x, waypoint_list_y):
    file.write(f'{i}, {j}\n')
with open(r'C:\your_directory\simulation_par.txt', 'w') as file:
    for i, j, k, u, v, w, z, a, b, c in zip(position_data_x,
   position_data_y, yaw_data, velocity_data_x, velocity_data_y,
   yaw_rate, acceleration_data_x, acceleration_data_y):
    file.write(f'{i}, {j}, {k}, {u}, {v}, {w}, {z}, {a}, {b}, {c}\n')
```

Listing 5.9: CARLA text files

In this last part of the "manual_control" script, written in the section of called "game loop," the data from the ego vehicle variables previously saved in the *name.append()* files are returned to the .txt files in tabular form. Finally, they are saved in the .txt files are saved in the directories indicated inside *with open()*.

5.1.5 Vehicle Parameters Gathering

In order to accurately model vehicle dynamics and increase the level of detail of the model, it is necessary to take into consideration the specific characteristics of the selected vehicle, and to accurately determine certain key parameters. One of the most relevant elements

for describing the dynamic behavior is the position of the CoM (center of mass) of the vehicle.
In the CARLA simulator, the vehicle position provided through the API represents the center of the bounding box (ground projection of the CARLA vehicle) in the simulated space, which does not necessarily coincide with the center of mass. Since the vehicle's dynamics are defined with respect to the latter, it is essential to know its actual position.
Through the **get_physics_control()** method, provided by the CARLA API, a number of physical parameters of the vehicle, including wheel characteristics, can be accessed and modified.

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

Listing 5.10: get_physics_control Function

This piece of code is part of the "manual_control" script, inserted after the ego vehicle spawn section, within the *restart(self)* loop. In this way, immediately after the simulation starts and the vehicle is spawned, the vehicle parameters appear on the command prompt.
The main ones, which will be considered in the thesis work are:

- Torque curve (x axis: Engine Rotational Speed [rpm], y axis: Engine Torque [$(N \cdot m)$])
- Maximum Engine Rotational Speed [rpm]
- MOI (Engine Moment of Inertia) [$kg \cdot m^2$]
- Vehicle Mass [kg]
- Aerodynamic drag coefficient []
- GB and Final transmission ratios []
- Parameters of all the 4 wheels, including radius [m], Maximum Braking Torque [$N \cdot m$] and Maximum Steering Angle [deg] for each wheel, Cornering [$\frac{N}{rad}$] and Longitudinal [N] Stiffnesses
- Distance (x,y) of the CoM from the center of the bounding box of the vehicle [m]

About geometrical parameters (vehicle length, width and height), they can be obtained from the function $self.player.bounding\_box.extent$, put inside the $get\_state(self)$ function.
The *bounding box* in CARLA is a three-dimensional box that encloses an object (vehicle, pedestrian, etc.) and defines its position, size, and orientation in simulation space. It serves as a simple representation of the volume occupied by the object, useful for detection, collisions and visualizations.
Specifically for a vehicle, the bounding box is defined by:

1. *Center* (location in the CARLA Global system, corresponding to the geometric center of the rectangular parallelepiped enclosing the vehicle).
2. *Dimensions* (semi-length, semi-width, semi-height).
3. *Orientation* (rotation)

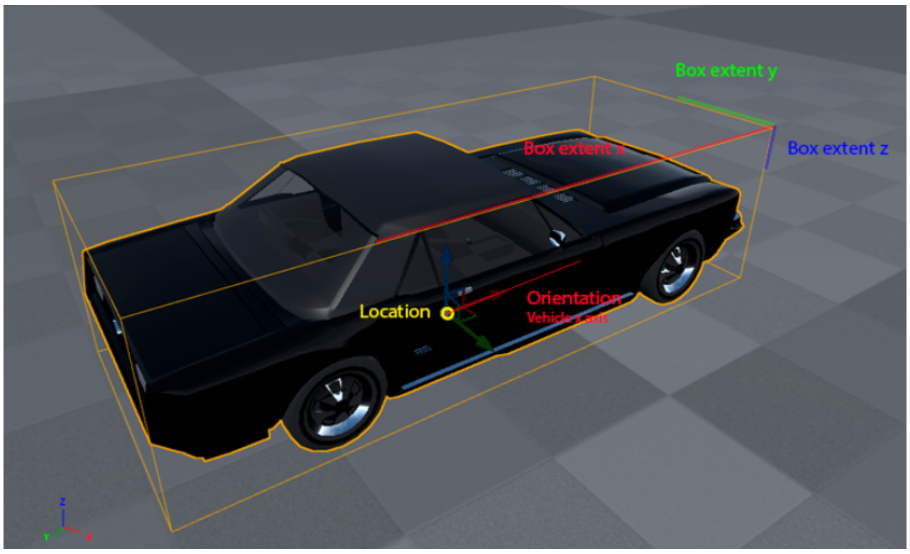


Figure 5.4: CARLA Bounding Box [99]

Below there is the piece of script to obtain these geometrical parameters:

```python
bb_extent = self.player.bounding_box.extent
vehicle_length = bb_extent.x * 2
vehicle_width = bb_extent.y * 2
vehicle_height = bb_extent.z * 2

# Print Dimensions
print(f"Vehicle Dimensions: - L: {vehicle_length:.2f}m, W:
   {vehicle_width:.2f}m, H: {vehicle_height:.2f}m")
```

Listing 5.11: self.player.bounding_box.extent Function

The Geometrical dimensions of the bounding box (that of the vehicle too) in the three axes x, y, z are included in the variable "extent". Due to the fact that we obtain the semi-dimensions of the vehicle, we have to double their values. Notice that the Bounding Box center is the geometric one, not the CoG of the vehicle.

5.1.6 Matlab-CARLA Interface

Communication between the CARLA simulator and the MATLAB environment is not straightforward, as there is no official direct interface to enable integration between the two. However, two ways of connecting these environments are proposed in the CARLA documentation.
The first option is to use a bridge based on **ROS (Robot Operating System)**. This solution, although more efficient in handling large volumes of data, has some critical issues: it requires numerous configurations and dependencies, and it is particularly optimized for Linux operating systems. Considering that the project in question runs on machines with Windows operating systems and does not require large-scale processing, a second integration mode was preferred.
The second option, actually used, involves the use of a **Python bridge**, as shown in the Figure below.

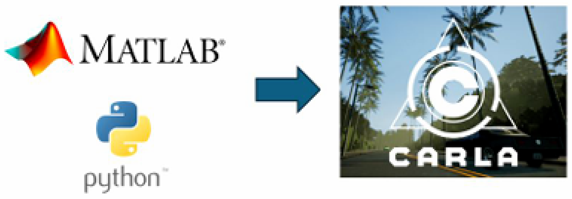


Figure 5.5: Matlab-CARLA Interface [96]

To ensure the proper communication, it is essential that the versions of MATLAB and Python are compatible. In this case, **Matlab** version **R2021b**, which is compatible with **Python 3.7**, was considered.
The following are the main steps to configure the communication environment via Anaconda Prompt within the already prepared virtual environment:

Listing 5.12: Python 3.7 module Installation

```
pip install setuptools==33.1.1
easy_install carla-0.9.14-py3.7-win-amd64.egg
```

This is the *setuptools* module installation, and after that, there is the addition of the path "$C:\ Python37\ Scripts$" to the path which contains the Carla Environment. *setuptools* is an essential package for managing and installing Python modules. It provides tools for packaging and distributing packages, including those in .egg format.
easy_install is a Python package installation tool, provided by *setuptools*. It is used to install an .egg package (an old but supported distribution format), which *pip* does not handle directly.
The .egg package contains the Python interface for the version of CARLA (in this case 0.9.14, compatible with Python 3.7 and Windows 64-bit). By installing it, Matlab will be able to access the Python CARLA module and use its API via the Python-Matlab bridge.
Then, it is advisable to run Matlab 2021b "as administrator" since the access to the necessary Python libraries (e.g., numpy, pygame, opencv-python) requires access to folders such as “Program Files” or/and other files in protected locations, and MATLAB requires administrator privileges to properly interact with them.
The following script is executed in Matlab:

Listing 5.13: Matlab-CARLA Interface

```
python_executable = 'D:\\python37\\python.exe';

carla_path = 'C:\\CARLA_0.9.14\\PythonAPI\\carla\\dist\\carla-0.9.14-py3.7-
    win-amd64.egg';

if count(py.sys.path, carla_path) == 0
    insert(py.sys.path, int32(0), carla_path);
end

carla = py.importlib.import_module('carla');
```

python_executable is the python path; *carla_path* is the CARLA simulator directory; the *if* is used to verify that the CARLA path is present in the Python modules (*py.sys.path*). If not present (count()=0), it is added to those modules. Finally, in the last line, the Python module 'carla' is imported to Matlab.

5.1.7 CARLA Environment in Matlab/Simulink

This section describes in detail the process of integrating the CARLA simulation environment with Matlab/Simulink, within which the simulation is performed.
The following pieces of code enable this integration via the Simulink **Matlab System** block, which allows, via the Python API, to establish a connection with the CARLA virtual world and actively interact with it.

Listing 5.14: CARLA Environment in Matlab - 1

```matlab
classdef Carla_env < matlab.System

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

    % Pre-computed constants
    properties (Access = private)
        client;
        car;
        spa1= importdata("CARLA_states.txt");
        world;
        actor;
    end

    methods(Access = protected)
        function setupImpl(obj)

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

First the public properties, i.e., the vehicle inputs (throttle, brake, steering), are defined. These properties can be modified.
Then the private properties, i.e., *client*, *car* (the ego-vehicle), *spa1* (the states of the vehicle during the simulation, taken from the "CARLA_states.txt" file), *world* (object representing the CARLA world) and *actor* (other actors present during the simulation) are defined. These properties cannot be modified.
To connect to Carla's client, the *setupImpl()* method is used, which is executed in the initial phase and allows to change the world settings, generate the necessary actors and manage their spawn (particularly of the ego vehicle).

Listing 5.15: CARLA Environment in Matlab - 2

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

            spa2= obj.spa1.data(1,:);
            if spa2(6)<0
```

```
                spa2(6)=(2*pi+spa2(6))*180/pi;
            else
                spa2(6)=(spa2(6))*180/pi;
            end
            spawn_point.location.x = 94.41;
            spawn_point.location.y = 209;
            spawn_point.location.z = 0.6;
            spawn_point.rotation.yaw =0;

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

Here the vehicle spawn is handled, so the type of vehicle (in this case a SEAT Leon), and the spawn location are chosen.
Then the yaw angle is taken from *spa1* and converted to degrees in the range $[0, 360]°$.
Now, the spawn point of the vehicle to be controlled is set (x, y, z, yaw). This can also be done by taking these coordinates directly from *spa2()*, but in this case they are set manually.
Finally, the vehicle is spawned in the virtual environment and the autopilot is turned off.

Listing 5.16: CARLA Environment in Matlab - 3

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

            vehicle_transform = obj.car.get_transform();
            position=vehicle_transform.position;
            velocity=obj.car.get_velocity();
            acceleration=obj.car.get_acceleration();
            orientation = vehicle_transform.rotation;

            position_x = position.x;
            position_y = position.y;
            yaw_angle = double(deg2rad(orientation.yaw));
            velocity_x = velocity.x;
            velocity_y = velocity.y;
            w = obj.car.get_yaw_rate().z*pi/180;

            acceleration_x = acceleration.x;
            acceleration_x = acceleration.y;

            control = obj.car.get_control();
            control.steer = rad2deg(steeringangle_input)/70;
            control.throttle = throttle_input;
            control.brake = brake_input;


            ze = [position_x;position_y;yaw_angle;velocity_x;velocity_y;w];

            obj.car.apply_control(control);
        end
```

The *stepImpl()* function is executed cyclically throughout the simulation and allows, at each time step, interaction with the virtual environment via Python commands. At this stage, *getter* and *setter* methods can be used to acquire vehicle state information and issue driving commands to the ego vehicle. This function sets the control commands

(throttle, brake, and steering) of obj.car and returns the vehicle states (ze) and the two acceleration components.
Then speed, acceleration, location, orientation, and vehicle control parameters are all extracted via CARLA's Python API, which provides specific methods for reading the current state of an actor (in this case, the vehicle). Some parameters are converted: yaw_angle in [°], yaw_rate in [$\frac{rad}{s}$] and steer in [°] and then normalized in the range [0, 1] (it is divided by its maximum value).

Listing 5.17: CARLA Environment in Matlab - 4

```
        function [ze,ax,ay] = isOutputComplexImpl(~)
            ze = false;
            ax = false;
            ay = false;
        end

        function [ze,ax,ay] = getOutputSizeImpl(~)
            ze = [6,1];
            ax = [1,1];
            ay = [1,1];
        end

        function [ze,ax,ay] = getOutputDataTypeImpl(~)
            ze = 'double';
            ax = 'double';
            ay = 'double';
        end

        function [ze,ax,ay] = isOutputFixedSizeImpl(~)
           ze = true;
           ax = true;
           ay = true;
        end
```

ax and *ay* are the x and y components of vehicle acceleration, respectively.
These functions define the characteristics of the output signals:

- *isOutputComplexImpl*: indicates that the outputs are real if false.
- *getOutputSizeImpl*: specifies the size of the outputs.
- *getOutputDataTypeImpl*: specifies the type of output variable (double in this case).
- *isOutputFixedSizeImpl*: all the outputs have fixed size if true.

Listing 5.18: CARLA Environment in Matlab - 5

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

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

The following are the functions used in this piece of code:

- *resetImpl*: if empty (like in this case) no discrete states have to be reset.
- *delete()*: at the end of the simulation (*isempty(obj.car)*) destroys the controlled vehicle to free up resources in the CARLA world.

Chapter 6

NMPC System Design

6.1 Simulations' Overview

The goal of the project is to develop a control system based on a nonlinear predictive controller (NMPC) for autonomous driving in an urban environment. This system will be able to follow a predefined path and perform additional functions to support integration with decision logic. The entire control system will be implemented in Simulink, taking advantage of the ability to connect and communicate with the CARLA simulation environment through the use of the *Matlab System* block. Several simulations were carried out by transporting the CARLA environment to Matlab/Simulink.
The first one is carried out in an urban environment, involves comparing the NMPC controller with the CARLA autopilot controlled via a PID. In order to make a meaningful comparison between the performance of the NMPC system and that of CARLA's default PID autopilot, it is necessary that both behaviors are initially as similar as possible. For this reason, the control system will be configured to maintain a constant speed of 30 $\frac{km}{h}$, as is the case in CARLA's autopilot in straight sections. The path to be followed will be the same as that used by the autopilot in the relevant scenario, so that trajectory errors and differences in behavior can be directly compared.
In the second, the same scenario as the first is exploited, but the ego vehicle goes at a higher speed; this scenario is critical to see if the NMPC still gives acceptable results even at 50 $\frac{km}{h}$.
The third and fourth ones involve a suburban interchange: first the behavior of the vehicle in a part of that scenario is evaluated, then whether it successfully completes an overtaking maneuver.
As in the scenario where the comparison between the autopilot and the system implemented with the NMPC takes place, in the others the cross-track error is used as the main parameter to evaluate the performance and adaptability of our system.

6.2 Data Extraction

Before describing the system (Simulink blocks and Matlab codes) in detail, it is necessary to explain how the information about the trajectories to be followed and the data obtained from CARLA's autopilot were obtained.
With the code used for Data Gathering (see *Section 5.1.4*), ego vehicle parameters during trajectory tracking in the simulated scenarios were collected and saved to text files. Then,

with the following pieces of Matlab code, the variables are extracted:

Listing 6.1: Data Extraction - 1

```
zeC = importdata('CARLA_states.txt');
uC = importdata('inputs.txt');
traj = importdata('waypoints.txt');
```

zeC contains CARLA vehicle states for each sampling time instant, *uC* contains the commands, and *traj* the reference trajectory to be followed during the simulation.

Listing 6.2: Data Extraction - 2

```
T = zeC(:,1);
X = zeC(:,2);
Y = zeC(:,3);
psi = zeC(:,6));
Vx = zeC(:,4);
Vy = zeC(:,5);

T1 = uC(:,1);
Ax = uC(:,2);
Ay = uC(:,3);

Xw = traj(:,1);
Yw = traj(:,2);

Reference = [Xw,Yw];
spawn_point = [Xw(1) Yw(1) psi(1)];
```

We start from the *zeC* matrix, which contains the temporal evolution of the vehicle state. Several pieces of information are extracted from it:

- *T* represents the time trend, obtained by taking the first column of the matrix.
- *X* and *Y* are the coordinates of the vehicle's position in the horizontal plane, respectively, corresponding to the second and third columns.
- *psi*, extracted from the sixth column, represents the yaw angle, i.e., the orientation of the vehicle with respect to the X axis.
- *Vx* and *Vy*, obtained from the fourth and fifth columns, respectively, represent the velocity components along the X and Y axes.

Next, we work on the *uC* matrix, which collects the control commands applied to the vehicle:

- *T1* is again time, but referred to the instants when commands are sent.
- *Ax* and *Ay*, on the other hand, are the accelerations commanded along the X and Y axes.

The trajectory to be followed by the vehicle is contained in the *traj* matrix. From here, the desired coordinates (*Xw, Yw*) are extracted.
These two sets of data (Xw, Yw) are then combined into the *Reference* variable, which represents the entire target trajectory in the form of a list of coordinate pairs. Finally, the *spawn_point* is created, which defines the initial point from which the vehicle is to depart, using the first value of these coordinates: *Xw(1)*, *Yw(1)* and *psi(1)*.

The data extracted from both *zeC* and *uC* matrices are the states and the commands of the CARLA autopilot, and will be used for the comparison with the NMPC system, while the coordinates of *Reference* and *spawn_point* will be used as reference trajectory and spawn point of the ego-vehicle controlled by the NMPC respectively.

6.3 Simulink Full System

The following is the full system implemented in Simulink:

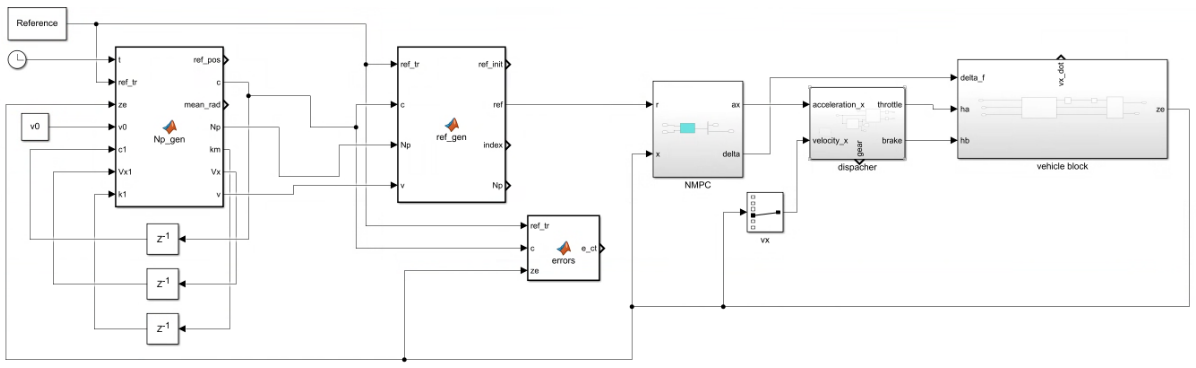


Figure 6.1: Simulink Full System

There are several blocks that will be discussed in the next chapters: "*Np_gen*", "*ref_gen*", "*errors*", "*NMPC*", "*dispatcher*" and the "*vehicle block*".

6.3.1 "Np_gen" Block

This function is responsible for locating the position of the car within the reference trajectory and calculating the average curvature of the trajectory in the following meters. The **Inputs** of the function are as follows:

- *ref_tr*: matrix containing the points on the reference trajectory *Reference*, in (x, y) coordinates.
- *ze*: state of the vehicle at the current time (position, angle, velocity, etc.).
- *v0*: reference velocity of the vehicle in straight line (at zero curvature).
- *c1*: index of the position on the trajectory in the previous step.
- *Vx1*: longitudinal speed of the vehicle at the previous step.
- *k1*: average curvature calculated at the previous step.

The following are the **Outputs**:

- *ref_pos*: point in the trajectory that the vehicle is currently following.
- *c*: index of that point within the trajectory.
- *Np*: number of meters to be considered for control prediction.
- *km*: average curvature of the trajectory in the following meters.

- *Vx*: longitudinal velocity of the current vehicle (extracted from ze state).

This function extracts the vehicle position, current longitudinal velocity *Vx*, and orientation from the state *ze*.
It also calculates which point on the trajectory *ref_tr* is closest to the vehicle's current position with respect to the reference trajectory *ref_tr*, and returns *c*. If *c* turns out to be too far behind or too far ahead of *c1* (more than ±0.4 m), it assumes that there has been a localization error (e.g., at an intersection) and performs an alternative estimate of the current position, calculating the distance traveled by the vehicle as *Vx1* · *Ts* (T_S is the sampling time), and running the trajectory from *c1* and summing the point-by-point distances until the distance traveled is exceeded, thus updating *c*.
To calculate the average curvature *km*, over the next 23 m (a value set for urban scenarios), points about 1 m apart along the trajectory are selected. Then a matrix of segments is constructed and the curvature in each section is calculated with the "*LineCurvature2D*" function, and finally any outliers (NaN) are removed and the average km is calculated. To avoid abrupt changes, km is limited to ±0.015 m from the previous value *k1*.
To calculate the desired velocity *v*, the dedicated equation is applied:

$$v = \frac{v_0}{10|k_m| + 1} \tag{6.1}$$

If the curvature is high (tight curve), the speed is reduced. A minimum of 10 m/s is also set.
From this velocity, Np (in meters) is calculated, that is, by how many meters the prediction horizon extends:

$$N_P = \frac{v}{3.6} T_P \tag{6.2}$$

where T_P is the Prediction Horizon.

6.3.2 "ref_gen" Block

The function extracts 50 equidistant points along the trajectory from the current position, up to a maximum distance of Np meters, and returns them as a flat reference trajectory (in a single vector). This trajectory will then be used by the controller (e.g., NMPC) to calculate the actions to be applied to the vehicle.
The **Inputs** of the function are *ref_tr*, *c* and *Np*, while the **Output** is *ref*: column vector of size 2*Ns x 1, which contains the coordinates of the points along the trajectory to be used as reference for control in the next prediction horizon. The points are equidistant and distributed over the trajectory of length Np.
During initialization, the function calculates the total number of points *N* in the ref_tr trajectory, sets *Ns* = 50, which is the number of points to be extracted along the trajectory to construct the controller's reference trajectory, initializes an index vector to save the indices of the selected points, and creates a *target_dist* vector of Ns values equidistant between 2 meters and Np meters. These are the target distances from point c for each of the points to be selected.
For point selection, for each target_dist(i) distance, the function runs a loop to cycle through the next points on the trajectory (ref_tr) starting at the current index *c*. For each candidate point k, it calculates the accumulated distance to the next points (*c+1* to *k*) by summing the Euclidean distances. As soon as the accumulated distance exceeds the target distance *target_dist(i)*, point k is saved in *index(i)*.

For trajectory construction, a ref_init array of size 2 x Ns is initialized where the coordinates (x, y) of each selected point will be saved; if a value of *index(i)* is zero (i.e., a point far enough away has not been found), then it is set to N (last available point). Each selected point is copied from the reference trajectory ref_tr into *ref_init*.
The final output contains in order the coordinates (x1, y1), ..., (x50, y50), which represent the reference trajectory to be used for predictive control in the next Np meters.

6.3.3 "errors" Block

Below there are the Matlab Function pieces of code to calculate the *cross-track error*.

Listing 6.3: error Function - 1

```
function e_ct   = error(ref_tr, c, ze)
    N = length(ref_tr(:,1));

    % Find reference segment endpoints
    if c < N
        % Get previous and next points
        prev_point = ref_tr(c-1, 1:2)';
        next_point = ref_tr(c+1, 1:2)';

        % Compare distances
        dist_prev = norm(prev_point - ze(1:2));
        dist_next = norm(next_point - ze(1:2));

        % Choose closer segment
        if dist_prev > dist_next
            p1 = c;
            p2 = c+1;
        else
            p1 = c-1;
            p2 = c;
        end
    else
        p1 = c-1;
        p2 = c;
    end
```

The **Inputs** of this function are *c* (the index of the trajectory point closest to the vehicle), *ref_tr* and *ze* (actual state of the vehicle), while the **Output** is the *cross-track error*.
N is the number of points in the trajectory.
The “if else” piece of script is used to find the segment formed by the trajectory points closest to the vehicle: *p1* and *p2*. The distances of the points before and after *c* from the vehicle are calculated. “$c < N$” means that *c* is not the last point of the trajectory. If the distance to the previous point is larger than the point following *c*, *p1* is assumed to be equal to *c*, and *p2* becomes *c+1*. Conversely, *p1* = *c-1* and *p2* = *c*. Whereas, if *c* is the last point of the trajectory: *p1* = *c-1* and *p2* = *c*.

Listing 6.4: error Function - 2

```
% Get segment coordinates
    x1 = ref_tr(p1, 1);
    y1 = ref_tr(p1, 2);
    x2 = ref_tr(p2, 1);
    y2 = ref_tr(p2, 2);

    % Calculate trajectory angle
    psi_tr = atan2(y2 - y1, x2 - x1); % Direction from p1 to p2
    psi_trc = wrapTo2Pi(psi_tr);

    % Vehicle states
    x = ze(1);
    y = ze(2);
    psi = ze(3);

    % Cross-track error (SCALAR calculation)
    e_ct = -(x - x1) sin(psi_trc) + (y - y1) cos(psi_trc);

end
```

Then the coordinates (x, y) of points *p2* and *p1* are defined, and from these the reference yaw angle of the *psi_trc* trajectory is calculated, considering the segment formed by the two points.
Then the coordinates (x, y) of points *p2* and *p1* are defined, and from these the reference yaw angle *psi_trc* of the trajectory is calculated, considering the segment formed by the two points. Then it is converted to the corresponding angle in the interval $[0, 2\pi]$ using the function *wrapTo2pi()*, and the actual position (x, y) and orientation of the vehicle are extracted from the state *ze*.
Finally, the cross-track error is calculated with the equation *4.14*. That formula is used and not *vecnorm()* since it is important to plot the error with both positive and negative values to understand the position of the vehicle with respect to the trajectory (see *Chapter 4.2.4*).

6.3.4 "dispatcher" Subsystem

This function in the Simulink Complete System is what is called "*dispatcher*." Its main purpose is to convert longitudinal acceleration commanded by the NMPC into throttle and brake commands, to be given as input to CARLA's vehicle. The following is the Simulink block diagram of this function:

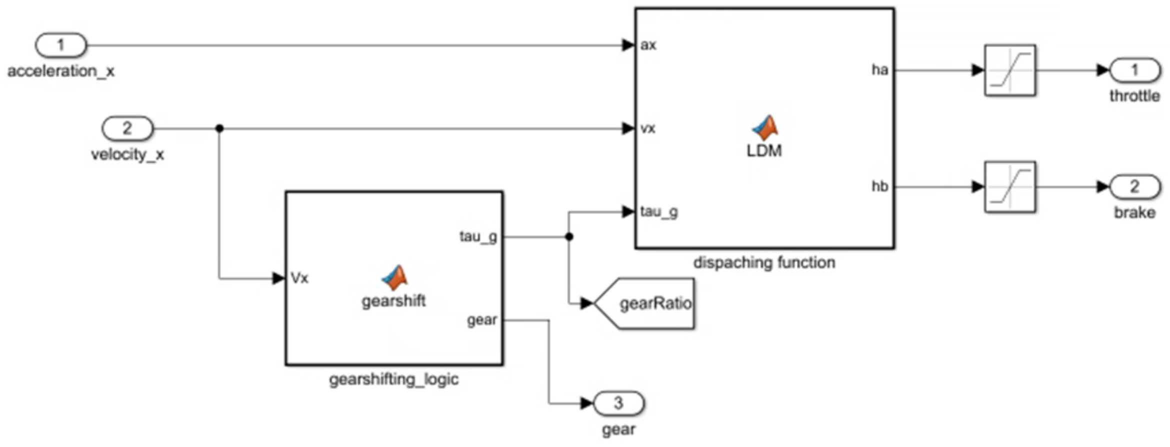


Figure 6.2: Dispatching Function

The actual dispatching function is the one labeled in the Matlab Function "*LDM*", which is an acronym for "Longitudinal Dynamics Model," and is the function that, using a simplified longitudinal dynamics model converting acceleration to throttle and brake.

The second one ("*gearshifting_logic*") is a function that handles the gear to be used, which is necessary for the conversion.

"gearshifting_logic"

The following is the code of that Matlab Function block:

Listing 6.5: dispaching funtion - geashifting logic

```
function [tau_g, gear] = gearshift(Vx)
    tau_vect = [3.46 2.05 1.3 1 0.91 0.76];
    shiftSpeed = [0 21/3.6 36/3.6 57/3.6 74/3.6 82/3.6];

    gear=1;

    % Determinazione della marcia in base alla velocità
    if (Vx >= shiftSpeed(1) && Vx <= shiftSpeed(2))
        gear = 1;
    elseif (Vx > shiftSpeed(2) && Vx <= shiftSpeed(3))
        gear = 2;
    elseif (Vx > shiftSpeed(3) && Vx <= shiftSpeed(4))
        gear = 3;
    elseif (Vx > shiftSpeed(4) && Vx <= shiftSpeed(5))
        gear = 4;
    elseif (Vx > shiftSpeed(5) && Vx <= shiftSpeed(6))
        gear = 5;
    elseif (Vx > shiftSpeed(6))
        gear = 6;
    end

    % Assegnazione del rapporto di trasmissione corrispondente

    tau_g = tau_vect(gear);
end
```

The **Input** of this function is the longitudinal speed of the vehicle Vx, and the **Output** is the GB gear ratio *tau_g* value of the corresponding gear.
Two solutions can be adopted for logic in gear shifting: one based on engine revolutions, and one based on vehicle longitudinal speed (when driving a vehicle, an arrow appears on the dashboard indicating upshift or downshift, and the vehicle speed at that time corresponds to the limit speed in a specific gear). In this thesis work, the second logic was chosen.
Regarding the variables in the code we have:

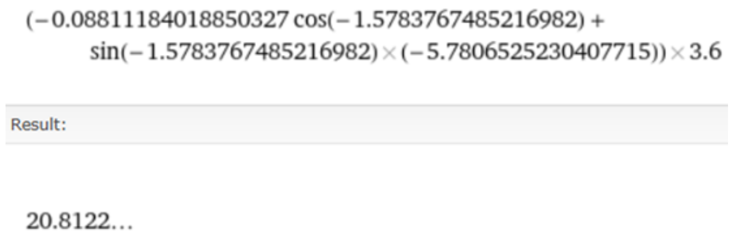
- *gear*: integer number (in this case from 1 to 6) representing the actual gear of the vehicle.
- *tau_vect*: vector with the various gearbox ratios (6 values correspond to 6 gears) and the higher the value, the lower the gear.
- *shiftSpeed*: vector with the limit speeds in each gear, thus containing the gear changing speeds.

These *shiftSpeeds* are estimated by exploiting the relationship between the angular velocity of the motor w_e and the longitudinal velocity of the vehicle Vx. The expression that summarizes this relationship is *Equation 3.6*. w_e is expressed in $[\frac{rad}{s}]$, and from the

CARLA data obtained through the *get_physics_control()* function there is the Maximum Engine Rotational Speed $n_{max} = 5500$ rpm. Converting n_{max} to $[\frac{rad}{s}]$, so as w_e, and rearranging the above equation, we obtain:

$$V_{n_{(n+1)}} = \frac{C \cdot n_{\max}}{\tau_g(n) \cdot \tau_f \cdot 60} \tag{6.3}$$

C is the circumference of the wheels, which is calculated as $2 \cdot \pi \cdot r$; r (wheel radius) and *tau_f* (final ratio) are obtained from the *get_physics_control()* function, and they are equal to 0.335 m and 2.6 respectively.
n is the actual gear, and n+1 is the next one. $\tau_g(n)$ is the GB transmission ratio at gear n. So $V_{n_{(n+1)}}$ is the *shiftSpeed* from n to n+1 gear, so the limit speed at gear n.
By using that equation, and by substituting the value of GB ratio for each gear n, each *shiftSpeed* is evaluated in $\frac{m}{s}$. But, in this way the *shiftSpeed* in the gear 1 would be 21.44 $\frac{m}{s}$, so about 77 $\frac{km}{h}$, that is a totally unrealistic value for a passenger car.
In addition, by entering into *Equation 3.16* the values of longitudinal speed, lateral speed and yaw angle of the vehicle from the file *"gearshift_logic.txt"* (see *Chapter 5.1.4*), in first gear before shifting to neutral gear during the simulations, it can be seen that the limiting speed in first gear of the vehicle in the CARLA Environment varies between 20.5 and 21.5 $\frac{km}{h}$, so it can be assumed as 21 $\frac{km}{h}$. An example of this calculus is expressed in the following figure:


$$(-0.08811184018850327\cos(-1.5783767485216982) + \sin(-1.5783767485216982) \times (-5.7806525230407715)) \times 3.6$$

Result:

20.8122...

Figure 6.3: shiftSpeed(1) Evaluation

So, to make the values obtained from that formula realistic and to obtain $V_{1_{(2)}} \simeq 21$ $\frac{km}{h}$, it was decided to take the value obtained directly in $\frac{km}{h}$. So, the following values are obtained, and approximated to the nearest integer number:

- $V_{1_{(2)}} = 21.44\ \frac{km}{h}$, so: $V_{1_{(2)}} = \mathbf{21}\ \frac{km}{h}$
- $V_{2_{(3)}} = 36.20\ \frac{km}{h}$, so: $V_{2_{(3)}} = \mathbf{36}\ \frac{km}{h}$
- $V_{3_{(4)}} = 57.08\ \frac{km}{h}$, so: $V_{3_{(4)}} = \mathbf{57}\ \frac{km}{h}$
- $V_{4_{(5)}} = 74.21\ \frac{km}{h}$, so: $V_{4_{(5)}} = \mathbf{74}\ \frac{km}{h}$
- $V_{5_{(6)}} = 81.55\ \frac{km}{h}$, so: $V_{5_{(6)}} = \mathbf{82}\ \frac{km}{h}$

The following table summarizes the speed ranges for each gear:

Gear	Vehicle Speed [km/h]
1	$0 \leq V_x \leq 21$
2	$21 < V_x \leq 36$
3	$36 < V_x \leq 57$
4	$57 < V_x \leq 74$
5	$74 < V_x \leq 82$
6	$V_x > 82$

Table 6.1: Gear vs Vehicle Speed

These values are quite realistic.
Note that in this logic the shift into neutral gear during gearshift is neglected, and the gearshift speed between two consecutive gears is assumed to be the same (no difference between upshift and downshift speeds).

"dispatching function"

This function uses the simplified 2DOFs model discussed in *Section 3.2*.

Listing 6.6: dispaching funtion - LDM

```
function [ha,hb] = LDM(ax,vx,tau_g)

% Parameters (SEAT LEON)
m = 1318;                % kg
Jw = 1.4;                % kg*m^2 (moment of inertia of wheels)
Jt = 0.02;               % kg*m^2 (moment of inertia of transmission shaft)
Je = 1;                  % kg*m^2 (moment of inertia of engine)
Te_max = 468;            % N*m (maximum engine torque at 4500 rpm)
Tb_max = 600;            % N*m (maximum brake torque per wheel)
eta = 0.85;              % \ (gearbox-differential efficiency)
tau_f = 2.6;             % \ (final transmission ratio)
Cx = 0.3;                % \ (aerodynamic drag coefficient)
Cr = 0.014;              % \ (rolling resistance coefficient)
rho = 1.225;             % kg/m^3 (air density)
n = 2;                   % \ (number of driving wheels)
n_nd = 2;                % \ (number of non-driving wheels)
r = 33.5 / 100;          % m (wheel radius)
A = 2.66;                % m^2 (frontal area)

%1st model of long. dynamics
me_tr = m + Je*(tau_g*tau_f/r)^2 + Jt*(tau_f/r)^2 + (n+n_nd)*Jw/r^2;
me_bk = m + (n+n_nd)*Jw/r^2;
Fres = 0.5*rho*Cx*A*vx^2 + Cr*m*9.81; %slope neglected
if ax > 0
    ha = (me_tr*ax + Fres)*r/(Te_max*eta*tau_g*tau_f);
    hb = 0;
elseif ax == 0
    ha = 0;
    hb = 0;
else
    ha = 0;
    hb = -(me_bk*ax + Fres)*r/((n+n_nd)*Tb_max);
end
end
```

The **Inputs** of the model are:

- *ax*: optimal longitudinal acceleration commanded by the NMPC.
- *vx*: longitudinal velocity, which corresponds to one of the *ze* states of the ego-vehicle.
- *tau_g*: current gear of the vehicle, obtained from the "*gearshifting_logic*" function. In this case we are considering a SEAT Leon with a 6 gears GB, so we need to adopt a function that chooses the gear during motion.

While the **Outputs** are:

1. **Throttle (ha)** Indicates the opening of the throttle control. In internal combustion vehicles, it represents the opening of the throttle valve that regulates the amount of air (and thus fuel) in the engine. In electric vehicles, it directly controls the output power of the engine.
2. **Brake (hb)** Represents the intensity of braking applied through the vehicle's braking system (discs, drums, regenerative braking in electric vehicles).

Both outputs, representing pedal and/or valve openings, are expressed in percentages or values in the range [0, 1]. To do this, two *Saturation* blocks have been introduced that require the two outputs to be between 0 and 1.
As for the calculation of the two commands to be given to the vehicle, they are calculated using two different equations of the longitudinal dynamics of the vehicle.
When the vehicle is stationary, or is at constant speed (ax = 0), it is assumed that there is neither acceleration nor braking, so ha = 0 and hb = 0. It should also be made clear that Matlab's machine precision approximates even very small values (also order of 10^{-2} - 10^{-1}) to 0; therefore, even for values of $\mathbf{ax} \simeq \mathbf{0}$, performing the calculations, Matlab's inherent approximation gives *ha, hb = 0*
When the vehicle is accelerating ($\mathbf{ax} > \mathbf{0}$), *hb = 0* is imposed, and ha is calculated from the acceleration and other parameters. The same is true for deceleration ($\mathbf{ax} < \mathbf{0}$): in this case *ha = 0* and *hb* is calculated using the longitudinal dynamics formula seen in *Section 3.2.2* in the braking case. Note that there are two different equivalent masses:

1. **me_tr** (during traction, then under acceleration condition) It uses the expression already seen above, and considers the GB gear ratios tau_g and the final gear ratio tau_f, the various moments of inertia (wheels *Jw*, drive shaft *Jt* and engine *Je*) and the unladen mass of the vehicle *m.* It is worth reiterating that during traction there is a "flow" of kinetic energy from the engine to the wheels, involving all of these vehicle components.
2. **me_br** (during braking) Here only the mass *m* and the moment of inertia of the wheels *Jw* are considered, since it is assumed that the brakes there are on the wheels dissipate all the kinetic energy. An equivalent mass equal to that in traction should be considered for a “flow” of kinetic energy involving all those elements of the vehicle, as in the case of regenerative braking of an electric/hybrid vehicle (in the opposite direction to the acceleration condition), a condition in which the kinetic energy is recovered from the vehicle's battery, thus going from the wheels, then to the powertrain and finally to the battery.

As for *ha* evaluation, it is obtained as:

$$ha = \frac{(m_{e\_tr} \cdot a_x + F_{res}) \cdot r}{T_{e\_max} \cdot \eta \cdot \tau_g \cdot \tau_f} \tag{6.4}$$

The total longitudinal traction force is:

$$Fx = \frac{\eta \cdot \tau_g \cdot \tau_f \cdot T_{e\_max}}{r}$$

This is the sum of the longitudinal forces acting on each wheel:

$$F_x = \sum_{i=n}^{4} F_{x_i}$$

Fres is the resisting force which is taken as the sum of rolling resistance and aerodynamic resistance. Slope resistance is neglected because in the simulated scenarios the roads are flat. This total drag contains only fixed terms, except speed, which varies quadratically in the aerodynamic drag expression.
As for the other terms, they are all fixed except ax (acceleration in x direction), vx (velocity in x direction), and tau_g (which also varies with speed). So we can say: *ha = ha(ax, vx, tau_g)* (also just *ha = ha(ax, vx)* because *tau_g* = f(vx)).
For the calculation of hb, the following formula is used:

$$hb = -\frac{(m_{e\_bk} \cdot a_x + F_{res}) \cdot r}{(n + n_{nd}) \cdot T_{b\_max}} \tag{6.5}$$

Regarding this equation under braking, it can be seen that the GB gear ratios do not appear, and the only variables are acceleration and speed. Thus: *hb = hb(ax, vx).* It can be seen that, in this case, the total longitudinal force under braking is:

$$Fx = -\frac{(n + n_{nd}) \cdot T_{b\_max}}{r}$$

obtained by summing up the longitudinal forces on each wheel. These are with - sign because they oppose the motion and they are all equal since, from the *get_physics_control()* function we get that the Maximum Torque at the wheels T_{b_max} is the same for all of them, which is not always true in reality (usually the brakes between front and rear wheels are different).
For more details on the model, view *Section 3.2.*

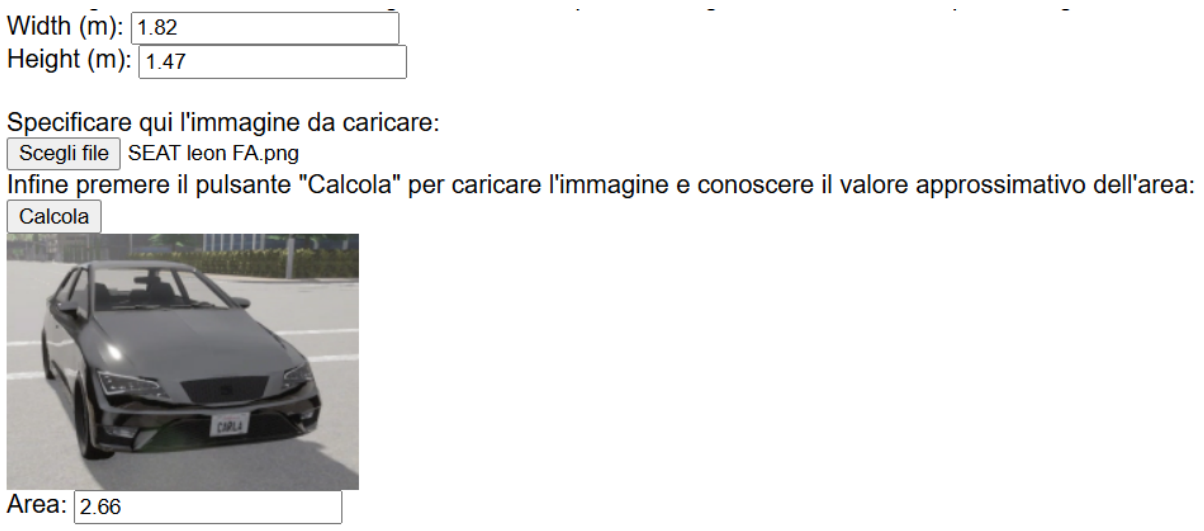
Parameters Estimation

Almost all of the parameters that are given in the *LDM* function code were derived from the simulator, with the *get_physics_control()* function, except for the following ones:

- *Jw*: wheel moment of inertia
- *Jt*: drive shaft moment of inertia
- *eta*: total transmission efficiency
- *A*: vehicle frontal area

- *Cr*: rolling resistance coefficient

It is important to get a good estimation of these parameters because they affect the longitudinal dynamics of our vehicle to a certain extent, thus indirectly influencing the states and commands that are imposed on the vehicle to be controlled.
Regarding *Jw* and *Jt*, typical values were taken from catalogs found around the web, resulting in:
$Jw = 1.4\ kg \cdot m^2$ and $Jt = 0.02\ kg \cdot m^2$.
About the estimation of the next parameters, it was decided to consider values with only 2 significant decimal places.
The frontal area of the vehicle was estimated using a web page [100], in which one must enter width, height in m, and a photo of the vehicle in which its frontal area is visible. Below there is a photo of the web page where the calculation was performed:


Width (m): 1.82
Height (m): 1.47

Specificare qui l'immagine da caricare:
Scegli file SEAT leon FA.png
Infine premere il pulsante "Calcola" per caricare l'immagine e conoscere il valore approssimativo dell'area:
Calcola

Area: 2.66

Figure 6.4: Frontal Area Evaluation [100]

Both width and height of the vehicle were obtained from the *self.player.bounding_box.extent* function. By this calculation, it was obtained:
$A = 2.66\ m^2$
Regarding *eta* estimation, again typical values were found on the web, both for gearbox and differential efficiencies in urban settings (most scenarios are at urban speeds). Here are some values that were found:

1. 6-gear Manual Transmission efficiency: 92% - 96%

2. 6-gear Automatic Transmission efficiency (that of our vehicle): 85% - 90%

3. Typical Differential efficiency (in a classic front- or rear-wheel drive vehicle): 95% - 98% [101]

Considering that the maximum torque developed by the engine in the CARLA simulator (468 Nm) is significantly higher than that of a real reference vehicle, such as the first-generation SEAT Leon (250 Nm), it was deemed appropriate to adopt an efficiency value corresponding to the maximum obtainable for an automatic transmission, i.e. $\eta_g = 90\%$. Meanwhile, for the differential, it is necessary to consider that it varies meanwhile the vehicle travel on a straight or on a curve road. When the vehicle goes on a curve, the differential must allow the wheels to turn at different speeds: this results in more internal work between the gears and, consequently, a slight increase in mechanical losses due to friction and power recirculation within the differential itself. Efficiency, therefore, tends to decrease slightly in cornering compared to straight ahead, although under normal driving conditions this variation is modest and hardly noticeable in practice. [102] In the simulated scenario there are some curve, so we assume the typical minimum value for differential efficiency, i.e., $\eta_d = 95\%$. The overall transmission efficiency is then calculated as:

$$\eta = \eta_g \cdot \eta_d = 0.90 \cdot 0.95 = 0.855$$

The approximation to the nearest integer/decimal is by excess from .51, then obtaining, to the second decimal place:
eta = 0.85
Finally, as for the rolling resistance coefficient Cr, it was estimated using the Transition Speed expression discussed in *Section 2.2.3*. The Transition Speed of this vehicle is not known, but it is known to be between 60 and 80 $\frac{km}{h}$. To minimize the maximum error we can obtain by considering any value in that range, its mean value is assumed, i.e.: $V_t = 70\frac{km}{h}$. By substituting this value into *Equation 3.14*, and rearranging the formula, we get:

$$Cr = \frac{(\rho \cdot A \cdot Cx \cdot (\frac{70}{3.6})^2)}{(2 \cdot m \cdot g)} = 0.01433$$

Rounding to the third decimal place:
Cr = 0.014
This value is also consistent with CARLA's Python documents, which state that, by convention, *Cr* is between 0.007 and 0.014. [103]

6.3.5 NMPC Dynamics Model

The heart of NMPC is the prediction of the future behavior of the system (in our case, the system is a vehicle) in response to given commands. To make these predictions, the controller needs to know the dynamics of the system, that is, how state variables evolve over time based on inputs. This is the role of the model: to provide a mathematical (typically nonlinear) description of the system to be controlled.
In this case, the model used by the NMPC is the *Single-Track Model* (see Section 2.3). As was explained earlier, the STM simplifies the vehicle to two virtual wheels (one front and one rear), assuming symmetrical behavior between the left and right sides. This reduces the number of equations and variables, while retaining the fundamental dynamics (such as inertia, lateral forces, and steering). In practice, it is less complex to compute, but still faithful enough for the NMPC.
The following are the Matlab pieces of script of that model:

Listing 6.7: Single-Track Model - 1

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

% Parameters
a = 1.168;          % [m]
b = 1.568;          % [m]
m = 1318;           % [kg]
J = 2345;           % [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_f = u(2,:);

% Slip angles
alpha_f = atan2(vy + a*w, vx) - delta_f;
alpha_r = atan2(vy - b*w, vx);

% Lateral forces
Fyf = -Cf.*alpha_f;
Fyr = -Cr.*alpha_r;
```

The **Inputs** of the model are the State variables (feedback control) and the NMPC commands (longitudinal acceleration *ax* and the front steering angle *delta_f*). About the State variables, they are:

- *X, Y*: longitudinal and lateral coordinates of the vehicle CoG in the CARLA Global Reference Frame.
- *vx, vy*: longitudinal and lateral CoG velocity with respect to the vehicle Local Reference Frame (they are aligned with x and y axes of the vehicle respectively).
- *psi, w*: yaw angle and yaw rate (variation of yaw angle in time) respectively.

The Slip angles are evaluated using equations (3.30) and (3.31), and Lateral forces are calculated using the Linear Tire model discussed in *Section 3.4.1*.

Listing 6.8: Single-Track Model - 2

```
% 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.*cos(delta_f) + Fyr);
w_dot = 2/J*(a*Fyf - b*Fyr);

% State derivative
xdot = [X_dot; Y_dot; psi_dot; vx_dot; vy_dot; w_dot];
y = x([1 2] ,:);
```

Whereas, the State equations (i.e., the equations describing the changes in the state variables) were evaluated in the following way:

- The first two are the equations (3.18) and (3.19), respectively, and they calculate the velocities in the longitudinal and lateral coordinates of the vehicle in CARLA's global system (*X_dot* and *Y_dot*) using the yaw angle of the vehicle and the two components of the velocities in the vehicle's local system (*vx* and *vy*).

- The third specifies that w is the derivative of the yaw angle *psi*.

- The last three are the characteristic equations of the STM (equations (3.15), (3.16), (3.17)).

Finally, the State derivatives are saved in the vector *xdot*, and the **Outputs** are the CoG coordinates in the Global Reference Frame.

Single-track Model Parameters Estimation

Regarding the parameter values used:

- m (vehicle mass), Cf and Cr (front and rear Lateral Stiffnesses, respectively) were already given by the *get_physics_control*() function.

- a, b (distances of the front and rear axles from the CoG of the vehicle, respectively) and J (moment of inertia of the vehicle about the vertical axis) were derived from parameters given by the same function.

For the calculation of a and b, the following Matlab code was used:

Listing 6.9: a and b Estimation

```matlab
pFR = [1653.336548; -375.503632];  % front right wheel
pFL = [1652.978394; -522.682678];  % front left wheel
pRR = [1926.931763; -376.169769];  % rear right wheel
pFL = [1926.573242; -523.348816];  % rear left wheel
xCoG = 0.2;                        % [m]

xF = (pFL(1) + pFR(1))/2;  % average front x coordinate
yF = (pFL(2) + pFR(2))/2;  % average front y coordinate
xR = (pRL(1) + pRR(1))/2;  % average rear x coordinate
yR = (pRL(2) + pRR(2))/2;  % average rear y coordinate

l = sqrt((xF-xR)^2 + (yF-yR)^2)/100  % wheelbase
a = l/2 - xCoG
b = l/2 + xCoG
```

From the *get_physics_control()* function we obtain several wheel parameters, including the 3D position vector of each wheel at the vehicle spawn point in the CARLA global reference system (z coordinate is not written because it is the same for all the 4 points). For the convention of the wheels (front right, front left, etc.) the first two wheels are the front ones with a view from above, so the wheels with the largest x coordinates are considered the front ones, and those with the largest (least negative) y coordinates were assumed to be the right wheels. The figure below summarizes this convention:

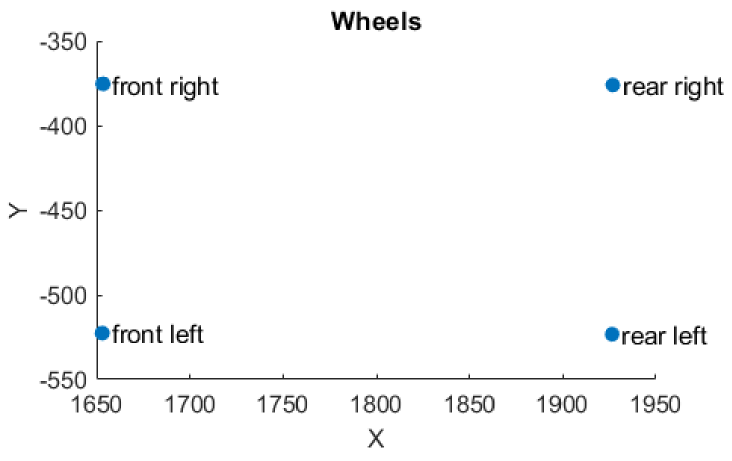


Figure 6.5: Wheels' Coordinates

The wheelbase l was calculated using the formula for the distance between two points, that are the middle ones in the front (xF, yF) and in the rear (xR, yR) axles, then dividing by 100 to obtain the numerical value in [m]. Then, considering that the CoG of the vehicle is ahead of the geometric center of CARLA's vehicle by a distance *xCoG* along the longitudinal axis of the vehicle, a and b were calculated as the semi-wheelbase ±*xCoG* (CoG being shifted toward the front axle of the vehicle: $a < b$, so -*xCoG* for the calculation of a, and +*xCoG* for the calculation of b). From these calculations, it is obtained:
l = 2.736 m, a = 1.168 m, b = 1.568 m
Meanwhile, for the moment of inertia J evaluation, the vehicle was assumed as a rectangular parallelepiped. So, it was calculated by using the generic expression of moment of inertia of a rectangular parallelepiped and by applying the Huygens-Steiner theorem (CoG is translated on x the direction of the vehicle by xCoG with respect to the bounding box center):

Listing 6.10: J Estimation

```
m = 1318;       % [kg]
L = 4.19;       % vehicle length [m]
W = 1.82;       % vehicle width [m]
H = 1.47;       % vehicle height [m]
xCoG = 0.2;     % [m]

J = 1/12*m*(L^2 + W^2) + m*xCoG^2
```

L, W and H are obtained by the *self.player.bounding_box.extent* function. Then the value calculated by this expression is approximated to the nearest integer number, obtaining:
$J = 2345\ kg \cdot m^2$

6.3.6 "vehicle block" Subsystem

The following is the "vehicle block" subsystem in Simulink:

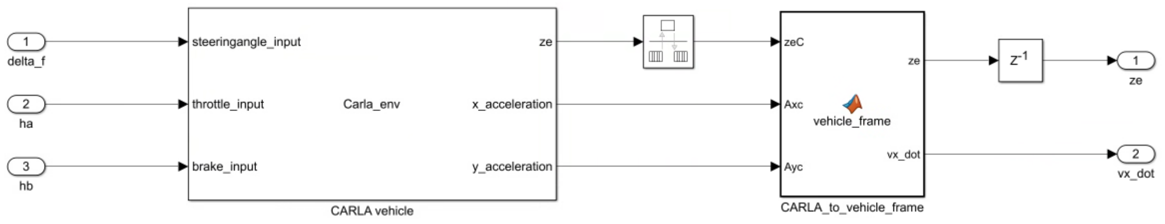


Figure 6.6: Vehicle Subsystem

It is made up of two main blocks: "*CARLA vehicle*" (to transpose the CARLA environment in Simulink) and the "*CARLA_to_vehicle_frame*" (used to convert some variables from the Global Reference frame into the respective ones into the Vehicle Reference frame).
Moreover there are two auxiliary blocks:

- **Rete Transition** In Simulink it is used to manage the transfer of data between blocks operating at different sample rates. This block ensures that data is transferred correctly from one block operating at a certain update rate to another block using a different rate, avoiding data integrity or synchronization problems.
  It is necessary to insert the initial values of the states *ze* (so states' values at the spawn) and CARLA sampling time (declared equal to 0.05 s).
- **Delay 1 Step** It is used to solve some algebraic-loop errors in Simulink. It is used only in some point of the block diagram in which these errors are not solved by the command "Minimize algebraic loop" in the Simulink settings. Data of that block are the default ones.

"CARLA vehicle" Block

It is a Matlab System Simulink block, based on the *Carla_env* function discussed in the *Section 5.1.7* (used to transpose the CARLA Environment into Matlab/Simulink).
The **Inputs** of the system are:

1. *ha*: Throttle command, coming from the "*dispacher*" subsystem.
2. *hb*: Brake command, an output of the "*dispacher*" too.
3. *delta_f*: Front Steering angle, imposed by the NMPC system.

While the **Outputs** of the system are ze (vehicle states) and *acceleration_x* and *acceleration_y* (respectively longitudinal and lateral accelerations). All these data are expressed in the Global (CARLA) reference frame. Moreover, the icon "Interpreted execution" is imposed, because the other option ("Code generation") cannot manage dynamic data.

"CARLA_to_vehicle_frame" Block

This function is necessary because, as it is discussed about the STM used to approximate the dynamics of our vehicle, some variables are expressed in the Vehicle local reference frame, and not in the global one, in which all the states of the "*CARLA vehicle*" block are defined. So, it is needed to convert some of them into the respective ones in the other reference frame. To do this, the script below is used:

Listing 6.11: CARLA_to_vehicle_frame function

```
function [ze, vx_dot] = vehicle_frame(zeC, Axc, Ayc)

xCoM = 0.2;      %m (distance vehicle C - CoM)

% CARLA states (global frame)
Xc = zeC(1,:);
Yc = zeC(2,:);
yaw = zeC(3,:);
Vxc = zeC(4,:);
Vyc = zeC(5,:);
yaw_rate = zeC(6,:);

% Vehicle CoG frame
X = Xc + xCoM*cos(yaw);
Y = Yc + xCoM*sin(yaw);
vx = Vxc.*cos(yaw) + Vyc.*sin(yaw);
vy = -Vxc.*sin(yaw) + Vyc.*cos(yaw);
vx_dot = Axc.*cos(yaw) + Ayc.*sin(yaw);

ze = [X,Y,yaw,vx,vy,yaw_rate]';
end
```

In this code the CARLA states *zeC* are taken from the Carla Environment Matlab System and used the conversion. The following variables are evaluated:

- *X, Y* They are the Vehicle CoG coordinates expressed in the Global reference frame. The CARLA coordinates *Xc* and *Yc* are that of the bounding box center, and to pass to the vehicle CoG are summed up to the variable *xCoG* (obtained by the *get_physics_control*() function). It is the distance, along the longitudinal axis of the vehicle, between the geometrical center and the center of mass of the vehicle. Due to the fact that the CARLA frame is inclined of *yaw* (yaw angle) with respect to the vehicle frame, we need to multiply *xCoG* by the sin and cos of that angle, and then sum to the CARLA coordinates *Xc* and *Yc*.

- *vx, vy* They are the vehicle velocities in longitudinal and lateral axes respectively. Vxc and Vyc are the vehicle velocities in the CARLA global reference frame, so we need to do the inverse operation of that done in the equations (3.18) and (3.19). From these ones, the rotation matrix to pass from the Local to the Global reference frame is obtained:
$$XY_{xy} = \begin{pmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{pmatrix}$$
It has to be multiplied by the vector containing the velocities in the local reference frame v = *[vx; vy]*.
To do the opposite transformation, we have to do the transpose of XY_{xy}, obtaining:
$$xy_{XY} = \begin{pmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{pmatrix}$$
By multiplying it by the vector V = *[Vxc; Vyc]* (velocity vector in the Global reference frame), v components (*vx, vy*) are evaluated.

- *vx_dot* It is the longitudinal acceleration of the vehicle in the vehicle reference frame. It can be evaluated in the same way of the velocity *vx*, starting from the components

of the acceleration in the CARLA reference frame *Axc, Ayc.*
The calculus of *vx_dot* is important to compare the behaviour of the longitudinal acceleration commanded by the NMPC and that of the real acceleration of the vehicle.

6.3.7 NMPC Simulation Parameters

The NMPC controller uses as input the current state of the vehicle and a set of 50 reference points to follow for each model output variable. From this data, the controller performs an optimization, according to a moving block formulation, over the prediction horizon of the cost function, calculating at each instant the optimal commands to be applied. As explained in the *Section 3.2.3*, the commands to be optimized are acceleration and steering angle, with the goal of minimizing the tracking error of the x and y references.
The *Optimization Criterion* of the controller is based on the following cost function:

$$J(u(t:t+T_p)) \doteq \int_t^{t+T_p} \left(\|\tilde{y}_p(\tau)\|_Q^2 + \|u(\tau)\|_R^2 + \left\| \frac{d}{d\tau} u(\tau) \right\|_{R_{sd}}^2 \right) d\tau + \|\tilde{y}_p(t+T_p)\|_P^2 \quad (6.6)$$

The equation is equal to *3.11*, with the exception of a term: $\|\frac{d}{d\tau}u(\tau)\|_{R_{sd}}^2$. This term is necessary because the moment you have very high Q matrix terms, even though the values of R are very low, there are high variations in the input commands. So, adding R_{sd} the variation of commands in the whole time interval τ is damped, promoting smoother inputs.
About the simulation parameters' values of the NMPC, they are shown in the following table:

NMPC Parameters	**Values**
T_s	0.05
T_p	3
n	6
R	$\begin{bmatrix} 0.06 & 0 \\ 0 & 1.3 \end{bmatrix}$
Q	$\begin{bmatrix} 1000 & 0 \\ 0 & 1000 \end{bmatrix}$
P	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
R_{sd}	$\begin{bmatrix} 22.8 & 0 \\ 0 & 494 \end{bmatrix}$
ncp	[0, 10, 50, 100]
ub	[5, 0.8727]
lb	[-8, -0.8727]

Table 6.2: NMPC Parameters' Values

Regarding the choice of T_s value, considering that our main focus is on motion planning and control, a sampling time of 50 ms was chosen as a compromise between responsiveness and computational load.
For T_p, since our vehicle mainly operates in an urban scenario characterized by sudden maneuvers in tight spaces, a prediction horizon of 3 s was chosen.

The Q and R matrices were accurately tuned, giving higher priority to reducing tracking error, thus obtaining very high Q values, and very low R values. The same is true for the R_{sd} matrix, obtaining values that are very similar in proportion to those of the R one.
Another important parameter of the NMPC system is *ncp*, which represents the points in the Prediction Horizion where commands change/are applied, expressed in T_p%. These values were tuned very precisely with simulations, also taking into account the number of nodes in the T_p to be used: with fewer points, worse performance results, while with more nodes, the computational load increases. In this case, at the beginning of the horizon the controls are applied, and at 10%, 50% and the end of the Prediction Horizon they vary.
Finally, *ub* and *lb* represent the upper and lower limits of the two input controls (longitudinal acceleration and steering angle), so, respectively, the maximum and the minimum values of the two NMPC commands. These values were estimated, taking into account the parameter values obtained from CARLA's *get_physics_control*() function.

ub & lb Estimation and Validation

Estimating the upper and lower bounds of the NMPC commands was done differently depending on the data available.
For the steering angle, its maximum value (in absolute value) is given to us by the *get_physics_control()* function, which gives information on various parameters for each wheel (maximum braking torque, maximum steering angle, radius, lateral stiffness at maximum load, longitudinal stiffness, and so on). From the parameters given, it appears that only the front wheels are steered, with a $\delta_f = 70°$. However, to stay away from the physical limits of the vehicle, it was assumed that this control is in the range [-50, 50]°.
Thus:
$\boldsymbol{\delta_{f_{max}}}$ **= 0.8727 rad**, $\boldsymbol{\delta_{f_{min}}}$ **= -0.8727 rad**
Whereas, regarding the longitudinal acceleration to be given as input to the vehicle, it can be estimated with the formulas used in the *LDM* function of the "*dispacher*." The formula in traction condition ($a_x > 0$) is used for the upper limit, and the expression in braking ($a_x < 0$) for calculating the lower limit. Since these are maximum values for the vehicle, both throttle and brake are assumed to have the maximum value ($h_a, h_b = 1$).
As for the upper limit, that is the maximum positive value of longitudinal acceleration, it is derived from the following expression:

$$m_{etr}(n) \cdot a_x = \frac{T_{emax} \cdot \tau_g(n) \cdot \tau_f \cdot \eta}{r} - F_{res}(V_x)$$

The only variables (other than a_x) are:

- $\tau_g(n)$, which is the GB gear ratio, varying with gear n.
- The Total Resisting Force F_{res}, which varies with the longitudinal speed V_x due to the contribution of aerodynamic drag.
- The Equivalent Mass m_{etr} (varying to a lesser extent than the other terms).

Thus, it can be said that maximum longitudinal acceleration occurs for $\tau_g(1)$ (in gear 1 the transmission ratio is the maximum one) and $V_x = 0$ (minimum velocity, neglecting the reverse motion), thus when the vehicle starts from standstill. Substituting into the equation the numerical values of these parameters seen in Section 6.3.5, we obtain:

$$a_x(\tau_g(1), V_x = 0) = 5.0240 \frac{m}{s^2}$$

Approximating to the nearest integer number: $\boldsymbol{a_{xmax} = 5\frac{m}{s^2}}$
Regarding the equation of motion under braking:

$$m_{ebr} \cdot a_x = -(\frac{4 \cdot T_{bmax}}{r} + F_{res}(V_x))$$

Except for a_x, the only variable term is the Resisting Force. As for the maximum deceleration value (thus maximum a_x in absolute value in this condition), it is obtained for the maximum V_x. This can be calculated by rearranging *Equation 3.6*, converting n_{max} to $[\frac{rad}{s}]$ and using $\tau_g(6)$ (the highest gear):

$$V_{xmax} = \frac{n_{max} \cdot r \cdot \frac{\pi}{30}}{(\tau_g(6) \cdot \tau_f)} = 97.6448\frac{m}{s}$$

Substituting this velocity value, and the other parameters to the previous equation of motion under braking, yields:

$$a_x(V_{xmax}) = -8.5947\frac{m}{s^2}$$

Considering that the maximum speed obtained is about 352 $\frac{km}{h}$ (a too-high maximum speed value, typical for a Formula 1 car and not for a passenger car) and that in CARLA documents the default minimum acceleration value is -8 $\frac{m}{s^2}$, the value obtained can be approximated to the lowest integer number, yielding: $\boldsymbol{a_{xmin} = -8\frac{m}{s^2}}$
So, to recap:

$$\delta_f \in [-0.8727, 0.8727]$$

$$a_x \in [-8, 5]$$

To validate these results, so to verify that these results reflect the values of a real vehicle, an additional parameter is evaluated: $\boldsymbol{t_{0-100}}$ (the time taken by the vehicle to go from 0 to 100 $\frac{km}{h}$).
In order to evaluate it, the same expression was used as for the calculation of a_{xmax} (h_a, h_b = 1, since the time with the maximum acceleration conditions must be found). Starting from a standstill and arriving at 100 $\frac{km}{h}$, the vehicle uses all the 6 gears, so that formula should be used in this calculation, but changing the speeds (the gear change speeds were used) and the value of the gear ratio τ_g. The following code was used for this calculation:

Listing 6.12: Evaluation of t_{0-100} - 1

```
%Parameters
Te_max = 468;
Tb_max = 600;
m = 1318;
vs = [0 21/3.6 36/3.6 57/3.6 74/3.6 82/3.6]';
vf = 100/3.6;
tau_g = [3.46 2.05 1.3 1 0.91 0.76]';
tau_f = 2.6;
eta = 0.85;
Jw = 1.4;
Jt = 0.02;
Je = 1;
Cx = 0.3;
A = 2.66;
Cr = 0.014;
```

```
rho = 1.225;
r = 0.335;

%Equivalent Masses evaluation
me1 = m + Je*(tau_g(1)*tau_f/r)^2 + Jt*(tau_f/r)^2 + Jw*4/(r)^2
me2 = m + Je*(tau_g(2)*tau_f/r)^2 + Jt*(tau_f/r)^2 + Jw*4/(r)^2
me3 = m + Je*(tau_g(3)*tau_f/r)^2 + Jt*(tau_f/r)^2 + Jw*4/(r)^2
me4 = m + Je*(tau_g(4)*tau_f/r)^2 + Jt*(tau_f/r)^2 + Jw*4/(r)^2
me5 = m + Je*(tau_g(5)*tau_f/r)^2 + Jt*(tau_f/r)^2 + Jw*4/(r)^2
me6 = m + Je*(tau_g(6)*tau_f/r)^2 + Jt*(tau_f/r)^2 + Jw*4/(r)^2

%Accelerations evaluation
a1_0 = (Te_max*tau_g(1)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(1))^2 - Cr*9.81*m)/
    me1
a1_21 = (Te_max*tau_g(1)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(2))^2 - Cr*9.81*m)/
    me1
a2_21 = (Te_max*tau_g(2)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(2)+0.001)^2 - Cr
    *9.81*m)/me2
a2_36 = (Te_max*tau_g(2)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(3))^2 - Cr*9.81*m)/
    me2
a3_36 = (Te_max*tau_g(3)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(3)+0.001)^2 - Cr
    *9.81*m)/me3
a3_57 = (Te_max*tau_g(3)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(4))^2 - Cr*9.81*m)/
    me3
a4_57 = (Te_max*tau_g(4)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(4)+0.001)^2 - Cr
    *9.81*m)/me4
a4_74 = (Te_max*tau_g(4)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(5))^2 - Cr*9.81*m)/
    me4
a5_74 = (Te_max*tau_g(5)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(5)+0.001)^2 - Cr
    *9.81*m)/me5
a5_82 = (Te_max*tau_g(5)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(6))^2 - Cr*9.81*m)/
    me5
a6_82 = (Te_max*tau_g(6)*tau_f*eta/r - 0.5*rho*A*Cx*(vs(6)+0.001)^2 - Cr
    *9.81*m)/me6
a6_100 = (Te_max*tau_g(6)*tau_f*eta/r - 0.5*rho*A*Cx*(vf)^2 - Cr*9.81*m)/
    me6
```

The parameters are the same and are given in the same way as the *LDM* function, except for the *shiftSpeed*, which is given as *vs*, equivalent mass in traction, which is given as *me* and the final velocity *vf*.
Regarding the conventions used:

- *mei*: equivalent mass *me*, calculated at gear *i*.

- *ai_v*: longitudinal acceleration at gear *i* and speed *v*.

For some speeds, *0.001* $\frac{m}{s}$ has been added because the value of *vs* in that gear is out of the range (example: in gear 2 the speed is greater than *21* $\frac{km}{h}$, so it is written as *vs(2)+0.001*).
In the table below, there are the results:

Gear	Speed [km/h]	Acceleration [m/s²]
1	0	5.0240
	21	5.0161
2	21+	3.7796
	36	3.7598
3	36+	2.5724
	57	2.5223
4	57+	1.9476
	74	1.8889
5	74+	1.7068
	82	1.6737
6	82+	1.3618
	100	1.2738

Table 6.3: Gear vs Speed vs Acceleration

The following is the MATLAB code to evaluate the Acceleration Map with respect to the vehicle speed.

Listing 6.13: Evaluation of t_{0-100} - 2

```matlab
v = linspace(vs(1), vf, 10000); % velocità in m/s
gear = zeros(size(v));
tau_g = zeros(size(v));

% Assegna la marcia corretta per ogni velocità
for i = 1:length(v)
    if v(i) >= vs(1) && v(i) <= vs(2)
        gear(i) = 1;
    elseif v(i) > vs(2) && v(i) <= vs(3)
        gear(i) = 2;
    elseif v(i) > vs(3) && v(i) <= vs(4)
        gear(i) = 3;
    elseif v(i) > vs(4) && v(i) <= vs(5)
        gear(i) = 4;
    elseif v(i) > vs(5) && v(i) <= vs(6)
        gear(i) = 5;
    elseif v(i) > vs(6)
        gear(i) = 6;
    end
    tau_g(i) = taug(gear(i));
end

% Calcolo della massa equivalente e accelerazione
me = m + Je.*(tau_g.*tauf/r).^2 + Jt.*(tauf/r).^2 + Jw*4/(r^2);
ax = (Temax.*tau_g.*tauf.*eta/r - 0.5*rho*A*Cx.*(v).^2 - Cr*9.81*m)./me;

% Calcolo del tempo totale tramite integrazione numerica
dt = diff(v) ./ ax(1:end-1); % dt = dv/a
t0_100 = sum(dt);
```

First, the code creates a list of speeds starting from the initial speed (vs(1)) all the way up to the final speed (vf). It doesn't just pick a few speeds; it chooses 10,000 points evenly spaced between these two values. This is like taking a very detailed snapshot of the car's

speed as it gradually increases.
Next, for each speed in this list, the code decides which gear the car should be, matching each speed with the correct gear.
Then, the equivalent mass me and the longitudinal acceleration ax are evaluated, using the expressions seen before.
Finally, the code figures out how much time it takes to go from one speed to the next small speed increment. Since acceleration is the rate of change of speed, the time to increase speed by a tiny amount is the speed increment divided by the acceleration at that point. By adding up all these tiny time intervals from the start speed to the final speed, the code finds the total time it takes to accelerate through the entire range.
Below there is the Acceleration versus vehicle speed plot in the interval [0, 100] km/h.

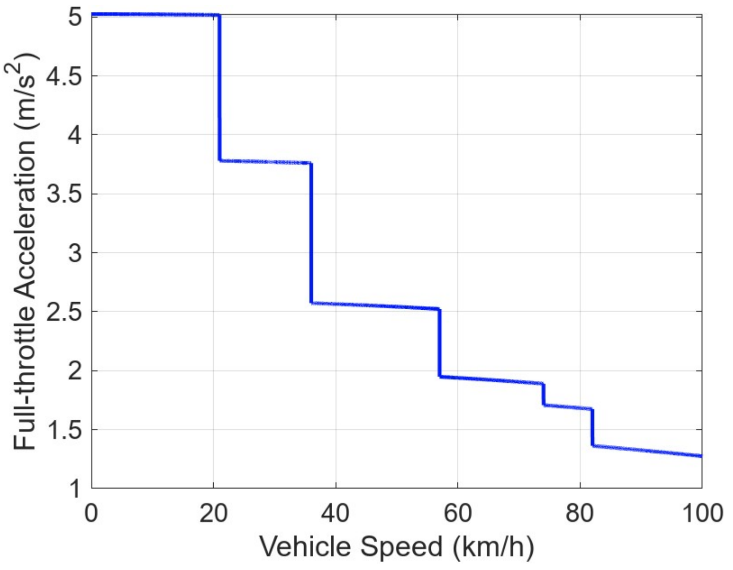


Figure 6.7: Acceleration Map

The Acceleration map has a stairs-shape, in which the abrupt changes in accelerations are in correspondance of the gearshifts. The higher is the speed, the lower is the longitudinal acceleration.
The 0 to 100 km/h time of this car results 12.1 seconds, practically equal to that of a Seat Leon 1.9 TDI with 100 CV. [104]
Since the acceleration time from 0 to 100 $\frac{km}{h}$ of the simulated vehicle turns out to be nearly coincident with that of a corresponding real version of the same model, its longitudinal dynamics can be assumed to be sufficiently representative of reality. Consequently, the implemented control strategy, having demonstrated good performance on such a virtual vehicle, is potentially transferable to a real vehicle as well, with a good chance of maintaining similar behavior.

Chapter 7

MATLAB-CARLA Integration - Simulation Results

This thesis chapter will present several scenarios with the results obtained from the vehicle. These aforementioned scenarios serve to study the adaptability and robustness of the control system discussed in the previous chapters under conditions very similar to those in the real world. The scenarios will be presented from the simplest (*indirect left turn*) to the most complex (*overtaking maneuver*) and several parameters will be discussed, including speed, acceleration and deviation from the trajectory.
The NMPC parameters, shown in the previous chapter, are the same for all these scenarios, and have been tuned to try to obtain the best possible result for the different conditions, then making a trade-off between the various results obtained.

7.1 Indirect Left Turn

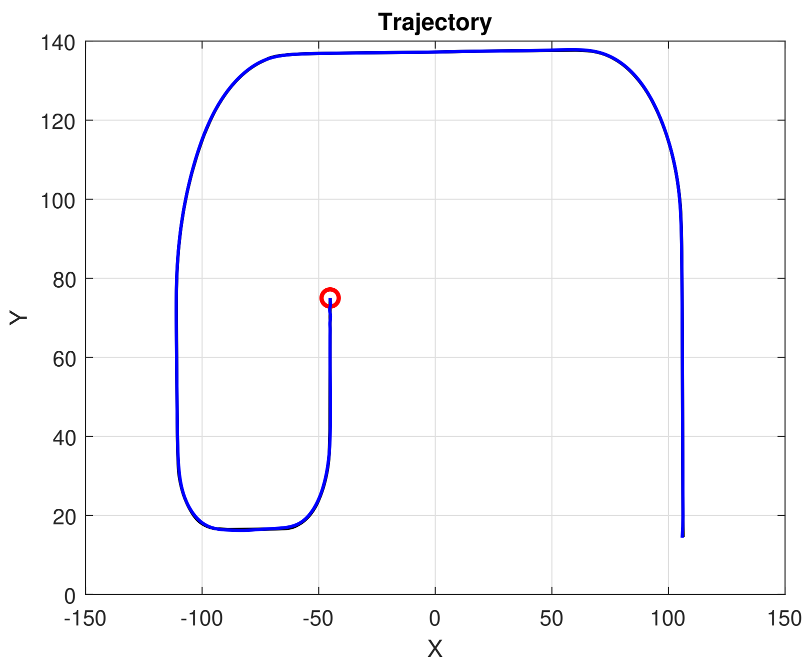


Figure 7.1: Indirect Left Turn

The scenery depicted in the image belongs to the Town10HD map of the CARLA simulator, a high-definition urban environment designed for realistic simulation of autonomous driving. Town10HD is one of CARLA's most detailed and superior maps, with improved graphic assets, more realistic materials, and enhanced lighting compared to previous versions. The urban environment features streets with complex intersections, traffic signals, vehicular and pedestrian traffic, and various urban elements such as buildings and vegetation.
The critical feature of this scenario is the 90-degree turns present in the turning maneuver; in the remaining part of the scenario, the trajectory is almost straight.
The results of two different simulations will be shown:

1. The first one in which the ego-vehicle presents a reference longitudinal speed of 30 km/h, and a comparison is made between the NMPC implemented in this thesis work and the PID autopilot of the CARLA simulator.

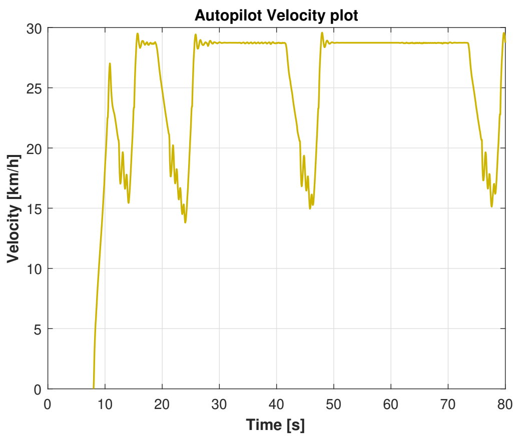
2. The second one in which the vehicle travels through the scenario at a reference speed of 50 km/h, considering that this is the speed limit for most Italian cities (70 km/h in particular conditions).

7.1.1 Indirect Left Turn at 30 km/h and Comparison with PID Autopilot

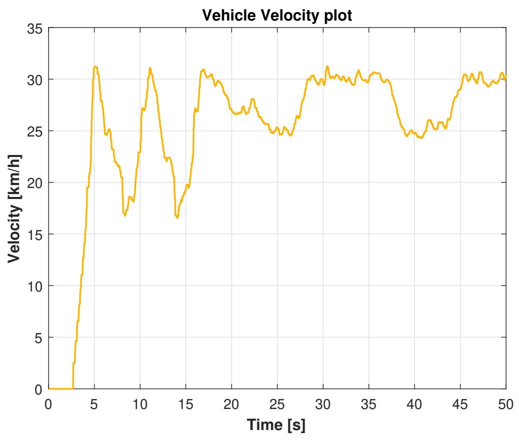
Regarding the first simulation with CARLA's autopilot controller comparison, the first specification to be verified is compliance with the prescribed reference speed. The graphs of the total velocities of the NMPC-controlled vehicle and CARLA's PID-based vehicles are shown below. The total velocities are evaluated as follows:

$$v = \sqrt{v_x^2 + v_y^2}$$

where v is the total velocity, v_x is the longitudinal velocity and v_y is the lateral speed.



(a) CARLA PID Vehicle Speed

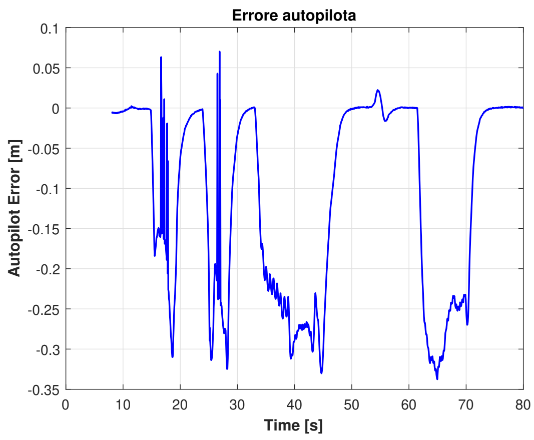


(b) NMPC Vehicle Speed

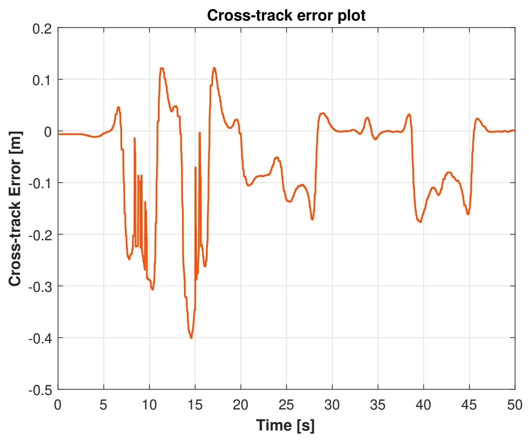
Figure 7.2: Comparison between Vehicle Speeds

The PID-controlled vehicle, once started, does not reach the reference speed in about 3 s, reaching about 27 km/h, and in the rest of the scenario it always remains at a speed slightly below 30 km/h. In the case of the NMPC-controlled vehicle, v0 = 26 km/h has been set (see chapter 6.3.1), and it reaches, even slightly exceeding, 30 km/h during

scenario tracking.
Another difference that can be seen, which also justifies the differences in tracking error, are the velocity throats, obtained due to the curvature variations in the scenario. For the vehicle with the autopilot, it can be seen that the vehicle slows down to 13-14 km/h, while the vehicle with the NMPC does not go below 16-17 km/h. This clearly shows that our vehicle can efficiently adapt to changes in trajectory and curvature, allowing for smoother path tracking.
A comparison between the cross-track error plots is shown below.



(a) CARLA PID based Autopilot Cross-track error



(b) NMPC controlled vehicle Cross-track error

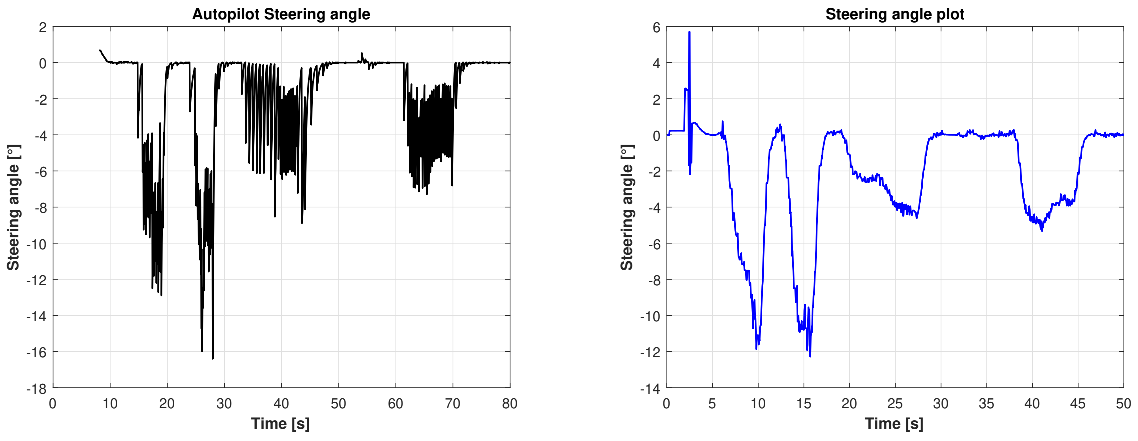
Figure 7.3: Comparison between Cross-track errors

Looking at the two graphs related to the cross-track error (e_{ct}, one can see significant differences between the behavior of the Autopilot and that of the Ego-vehicle.
In the first graph, which represents the absolute error of the Autopilot, one can clearly see how the error often remains at quite high values, with several peaks as high as 0.34 m. These peaks are mostly concentrated at curves, where the control system seems to have the most difficulty in maintaining the ideal trajectory. The error hardly ever goes to zero and has frequent and irregular oscillations, a sign that the Autopilot struggles more to correct deviations from the trajectory quickly.
The second graph, on the other hand, shows the error trend for the NMPC controlled vehicle. Here the situation looks much better: the maximum error reaches slightly larger values for the first curve (around ± 0.34 m), and smaller values for the second (± 0.18 m). The overall trend is more regular and stable. Again, peaks are noticeable at the curves, which are overall more moderate compared to the vehicle equipped with a PID-based autopilot. In addition, the error tends to remain closer to zero in straight sections, indicating greater accuracy in following the trajectory.

CONTROLLER	**RMS e_{ct} [m]**	**Max e_{ct} [m]**	**ASSESSMENT**
PID Autopilot	0.2184	0.3376	**Great**
NMPC	0.1146	0.4010	**Great**

In summary, comparing the two graphs, it is clear that the NMPC is able to maintain a lower and more consistent cross-track error than the Autopilot PID based. This results

in a more precise and stable ride, especially in curved sections where vehicle control is more challenging. Autopilot, in contrast, shows greater variability and more difficulty in handling curves, with higher and more frequent errors.
This analysis suggests that the control system implemented on the Ego-vehicle is more effective and reliable, providing better performance in terms of adherence to the desired trajectory, both in straights and curves. Moreover, as it can be seen from the table, the NMPC maximum cross track error is a bit higher than that of the PID based autopilot, but the RMS value of our controller is the half of that obtained with CARLA's controller. Overall, in numerical terms, both controllers provide great results in terms of path tracking and trajectory deviations.
Below there are the steering angles plot obtained from CARLA autopilot and the NMPC controlled vehicle respectively.

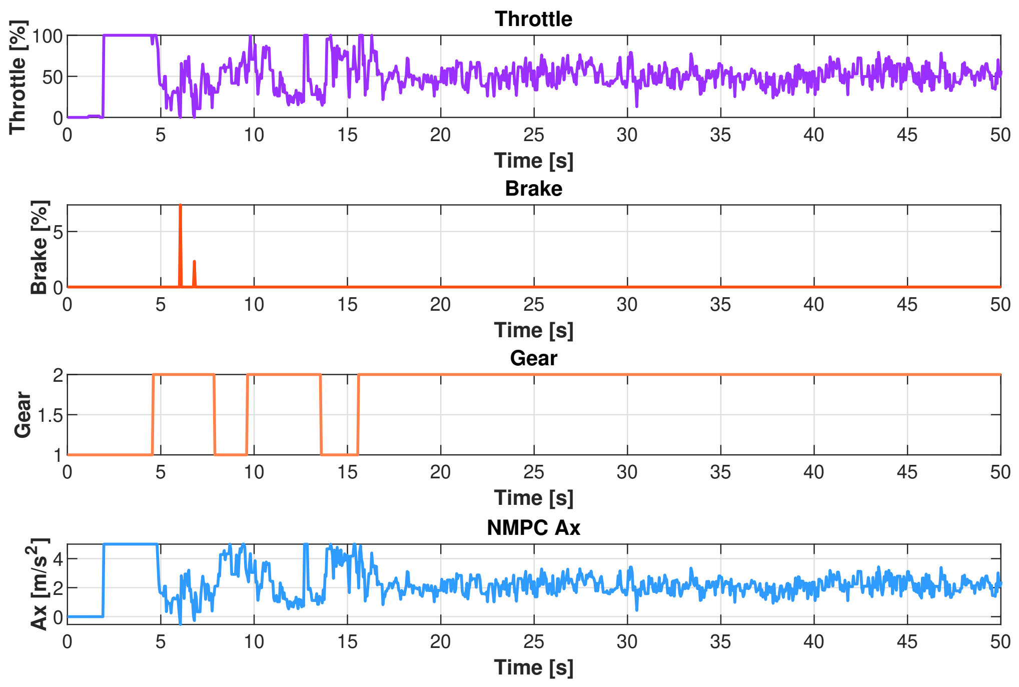


(a) CARLA Autopilot Steering Angle **(b)** NMPC controlled vehicle Steering Angle

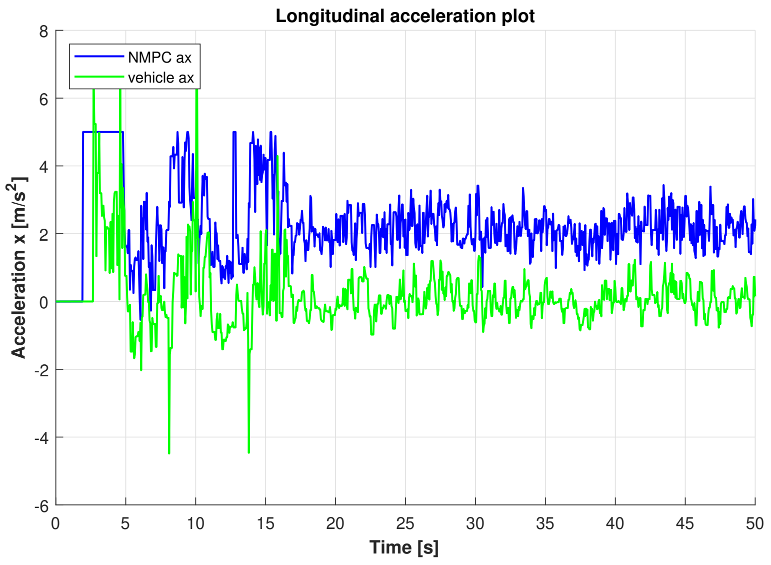
Figure 7.4: Comparison between Steering Angles

Looking at the two graphs, we immediately notice some significant differences in the behavior of the steering angle over time.
In the left plot, which represents the CARLA Autopilot steering angle, the signal is characterized by frequent and abrupt oscillations. The steering angle changes rapidly, with sharp transitions between positive and negative values. These fluctuations are particularly intense between 20 and 60 seconds, where the plot appears dense and irregular. This behavior suggests that the Autopilot is making constant, sudden corrections to the steering, resulting in a noisy and unstable control action. The values related to threads—the points where the vehicle negotiates curves—are numerous and closely spaced, indicating that the system reacts aggressively to changes in the road trajectory.
Conversely, the right plot shows the steering angle under NMPC control. Here, the steering angle varies in a much smoother and more continuous manner. The transitions between steering directions are gradual, and the overall signal is less noisy compared to the Autopilot. The threads are still present, marking the points where the vehicle takes curves, but they are more clearly defined and spaced further apart. This indicates that the NMPC controller handles curves with greater stability and predictability, making smoother adjustments rather than abrupt corrections.
Overall, the comparison reveals that the NMPC controller offers a more refined and stable steering behavior, especially when the vehicle is navigating curves. In contrast, the CARLA Autopilot demonstrates a more erratic steering pattern, with frequent and sharp

corrections that could lead to less comfortable or less safe driving dynamics.



(a) Throttle, Brake, Gear and NMPC Londitudinal Acceleration



(b) Comparison between NMPC and Real Accelerations

Figure 7.5: Throttle, Brake and Acceleration

Plot a): Control Variables

These four plots show the trend over time of several parameters related to the dynamics of a vehicle: accelerator (Throttle), brake (Brake), engaged gear (Gear), and longitudinal acceleration (NMPC Ax).
In the first graph, devoted to the throttle, it can be seen that in the very first seconds

the throttle control is at maximum, because the vehicle starts from standstill. After this initial phase, the throttle value decreases and begins to oscillate rather erratically, while remaining active throughout the observed period. This indicates that the vehicle continues to receive acceleration inputs, but with frequent variations, to adapt to road conditions and to balance the aerodynamics and rolling resisting forces.
The second plot shows the progress of the brake. Here we can see that the brake is applied only on very few occasions, all concentrated in the first 15 seconds. After these brief interventions, the value remains constantly at zero, a sign that the vehicle no longer needs to brake for the rest of the time considered.
The third graph shows the engaged gear. At first, it can be seen that the gear changes rapidly, shifting from one position to another in the first 15 seconds, since the vehicle accelerates from a standstill and increases speed. After this initial phase, the gear remains stable and there are no more changes for the rest of the period, suggesting that the vehicle has reached a cruising speed.
Finally, the fourth plot shows longitudinal acceleration (Ax), which represents how much the vehicle accelerates or decelerates along the direction of travel. Here again, higher and more variable values are observed in the first few seconds, corresponding to the start and acceleration phase. Thereafter, acceleration stabilizes at smaller values and oscillations become less pronounced, signaling a smoother ride. Acceleration presents the same variations as throttle and braking, being their combination.
In the figure below there is a comparison between the longitudinal acceleration commanded by the NMPC and the actual acceleration of the vehicle.

Plot b): NMPC and Real Accelerations Comparison
Looking at the graph, we see represented two curves that describe the trend of longitudinal acceleration (i.e., in the direction of travel) as a function of time. The blue curve ("NMPC ax") represents the acceleration that the predictive controller (NMPC) would like the vehicle to maintain: this is the ideal reference, calculated by a mathematical model that takes into account driving objectives and theoretical limitations. The orange curve ("vehicle ax"), on the other hand, shows the acceleration actually achieved by the vehicle during the test. At the beginning of the graph (first 0-15 seconds), both curves are quite erratic and have peaks and swings. These variation are caused by several reasons:

1. **NMPC ax** After the initial delay, there is, in the interval 3-5 s, a constant acceleration of 5 m/s^2 (the maximum one) because the vehicle starts from standstill. The other peaks and abrupt variations are due to gearshift.

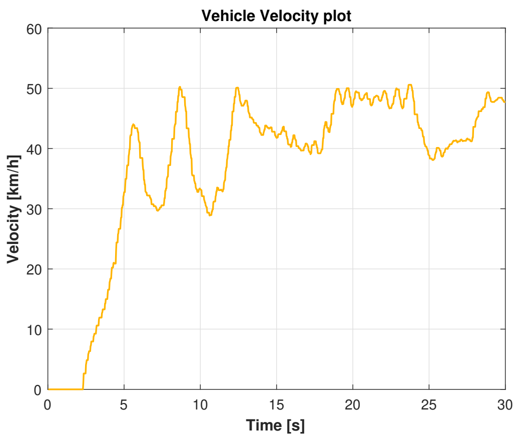
2. **vehicle ax** Changes in acceleration are sudden, like impulses, since, when CARLA's vehicle changes gears, it shifts to neutral before engaging the next gear. This aspect is not modeled in the dispaching.

After this initial phase, the curves stabilize: the blue curve remains higher on average, while the orange curve stays at lower values and still shows some oscillations.
A noticeable gap exists between these two signals, which is primarily due to differences in how throttle and brake commands are interpreted by the NMPC controller and the CARLA vehicle model.
In the NMPC framework, the desired acceleration is computed and then translated into throttle and brake commands, which are subsequently sent as inputs to the CARLA vehicle. However, the mapping between throttle values and the resulting acceleration differs significantly between the NMPC's internal model (the dispatcher) and the CARLA

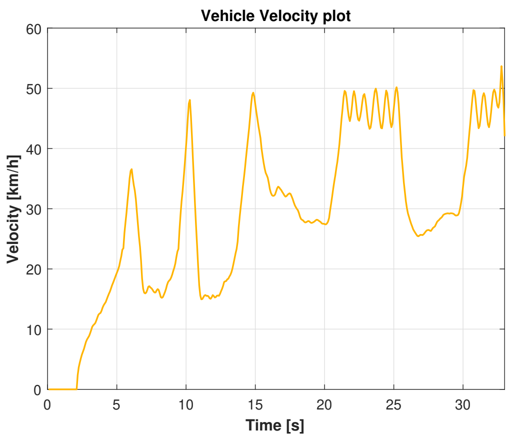
simulator. For example, in CARLA, a throttle value in the range of 0.4–0.55 (with the vehicle in second gear) results in an actual acceleration close to 0 m/s^2—essentially maintaining a constant speed. In contrast, the NMPC's internal model would associate the same throttle range with an acceleration of about 2 m/s^2.
As a consequence, to achieve constant velocity, the NMPC controller consistently requests an acceleration oscillating around 2 m/s^2. This command is converted to a throttle input of approximately 0.5 (or 50 %), which, according to the CARLA model, yields nearly zero acceleration. This explains why the blue NMPC acceleration signal is systematically higher than the green vehicle acceleration signal in the plot.
Despite this model mismatch between the NMPC dispatcher and the CARLA simulator, the overall system performs robustly. The vehicle is able to adapt to the reference speed and handle curves effectively. Moreover, the results in terms of cross-track error are excellent, with the vehicle maintaining low deviations from the desired trajectory. This demonstrates the controller's ability to compensate for differences in actuator modeling and still achieve precise and stable vehicle behavior.

7.1.2 Indirect Right Turn at 50 km/h

In the previous chapter, the vehicle tracking at 30 km/h was discussed. In this section, we will focus on path tracking at 50 km/h for the same scenario, which involves an indirect right turn. Specifically, we will compare two different approaches by varying the curvature parameter of the reference velocity: in the first case, the curvature parameter is kept at 10, while in the second case, it is increased to 30. Therefore, all the graphs corresponding to these two different approaches will be compared and analyzed.
Both the simulations are conducted with v0 = 43.



(a) Vehicle Speed - Kc = 10

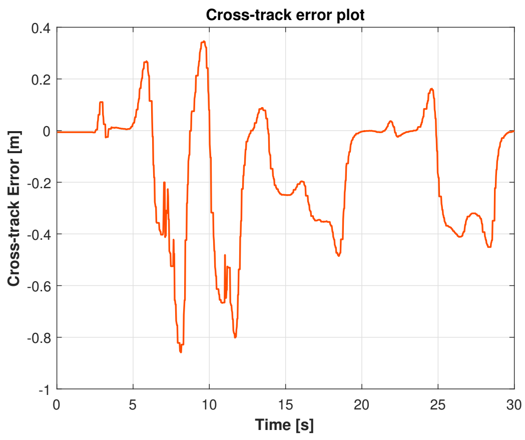


(b) Vehicle Speed - Kc = 30

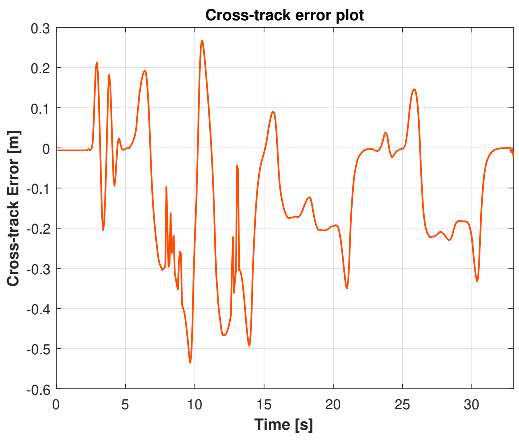
Figure 7.6: Comparison between Vehicle Speeds

When comparing the two vehicle speed plots for different curvature parameters, several differences become evident. In the first graph, where the curvature parameter Kc is set to 10, the vehicle's speed increases more gradually, and the overall profile appears smoother. The speed rises steadily and, after a few initial fluctuations, stabilizes closer to the reference value of 50 km/h. The oscillations in speed are relatively minor, indicating

a more controlled and stable response throughout the maneuver.
In contrast, the second graph with Kc=30 shows a less stable speed profile. Although the vehicle reaches higher speeds quickly, the speed fluctuates more sharply and frequently. Notably, during the curves, the vehicle experiences more significant speed reductions compared to the Kc=10 case. In fact, in the Kc=10 case the vehicle decelerate up to 30 km/h on the first curve, and up to 40 km/h on the second one; for the Kc=30 case, the reduction is higher: 15 km/h for the first curve, and about 25 for the second one. This indicates that with a higher curvature parameter, the vehicle slows down more aggressively when navigating turns, leading to larger speed drops in these sections.
So, increasing the curvature parameter from 10 to 30 results in more pronounced oscillations and greater speed reductions during curves.



(a) Cross-track Error - Kc = 10



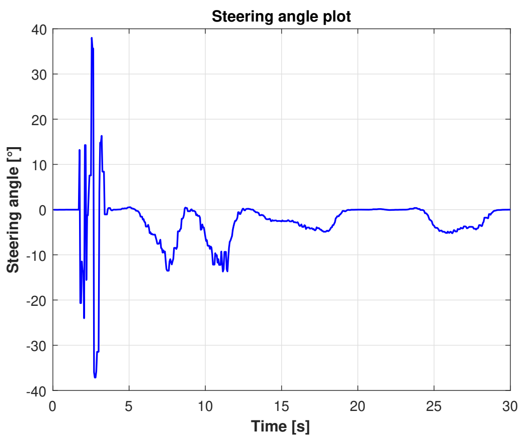
(b) Cross-track Error - Kc = 30

Figure 7.7: Comparison between Cross-track Errors

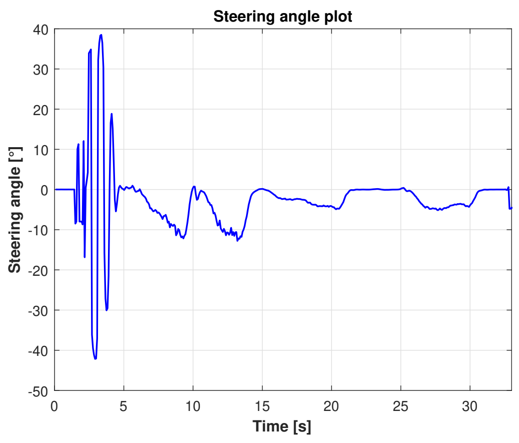
When comparing the cross-track error plots for the two different curvature parameters, several important differences can be observed.
For Kc=10 (left plot), the cross-track error fluctuates more significantly throughout the maneuver. The error values reach higher peaks, and the overall profile appears less stable, with more pronounced oscillations. This is confirmed by the table, where the root mean square (RMS) cross-track error is 0.2804 meters and the maximum error reaches 0.8585 meters. According to the assessment, this performance is considered "Acceptable," due to an high maximum value of the error on the first curve.
In contrast, for Kc=30 (right plot), the cross-track error is generally lower and the fluctuations are less severe. The error remains closer to zero for most of the maneuver, indicating better path tracking performance. The table supports this observation, showing a lower RMS error of 0.1975 meters and a reduced maximum error of 0.5354 meters. The assessment for this case is "Great," reflecting the improved accuracy and consistency of the vehicle's path tracking when the curvature parameter is increased. In fact the RMS value is excellent, and the maximum value a bit above 0.5 m represents also a good result in terms of accurancy.

Kc Value	**RMS e_{ct} [m]**	**Max e_{ct} [m]**	**ASSESSMENT**
10	0.2804	0.8585	**Acceptable**
30	0.1975	0.5354	**Great**

In this case increasing the curvature parameter from 10 to 30 leads to a significant improvement in cross-track error performance. The vehicle is able to follow the reference path more closely, with smaller deviations and less fluctuation, as demonstrated by both the plots and the numerical values in the table. This suggests that a higher curvature parameter enhances the vehicle's ability to stay on course, resulting in more precise and reliable path tracking.



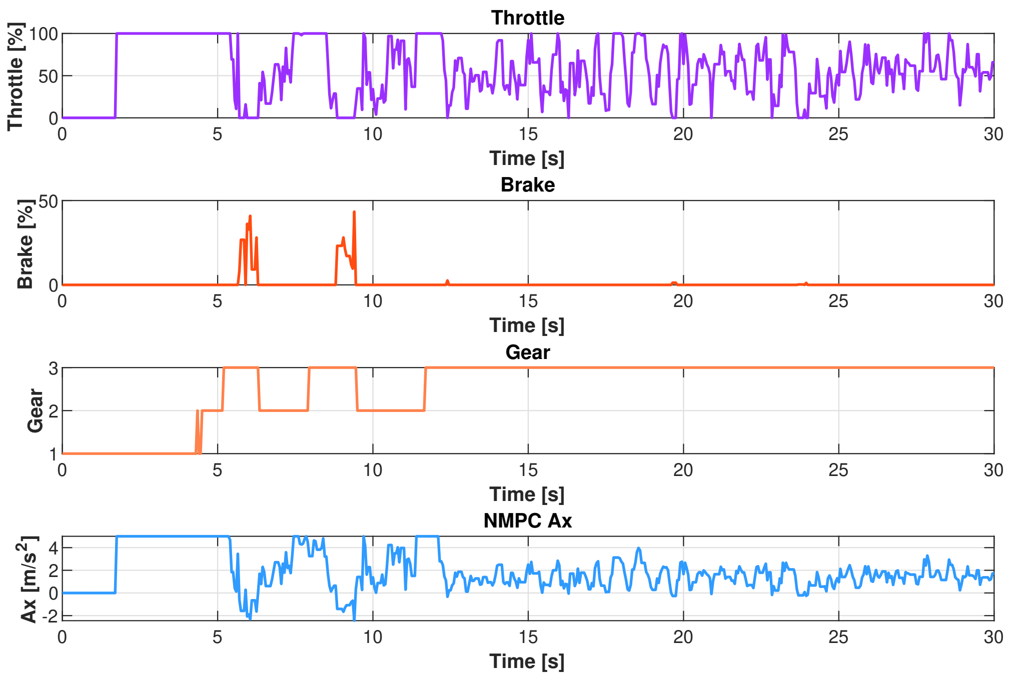
(a) Steering Angle - Kc = 10



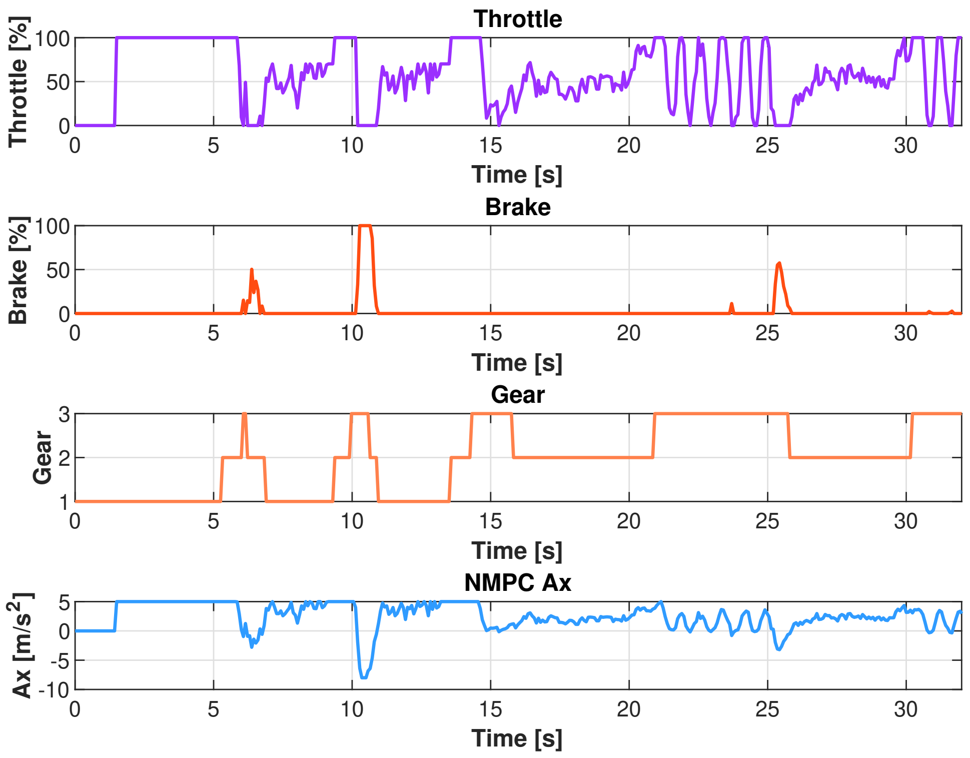
(b) Steering Angle - Kc = 30

Figure 7.8: Comparison between Steering Commands

The two steering angle plots provide a vivid illustration of how the choice of controller gain influences the dynamic behavior of a steering system. By examining these plots side by side, we can gain a deeper understanding of the subtle and not-so-subtle ways in which system responsiveness, stability, and control quality are affected by this parameter.
When a steering command is issued, the system's immediate reaction is crucial. With a lower gain (10), the system responds with a series of oscillations that, while noticeable, are relatively moderate in amplitude. The steering angle fluctuates above and below the desired value, as if the system is cautiously feeling its way toward the target. This cautiousness is reflected in the way the oscillations persist for a longer period, gradually diminishing as the system approaches equilibrium.
In contrast, increasing the gain to 30 transforms the system's character. The initial response becomes much more aggressive: the steering angle swings rapidly and with greater force, overshooting the target by a significant margin. These pronounced oscillations are a sign that the controller is reacting with urgency, eager to correct any deviation as quickly as possible. However, this urgency comes at a price—while the system reaches the steady state more quickly, the journey is marked by turbulence and instability in the early moments.
After these initial oscillations, both the systems maintain very low steering angles in correspondance of the curves, similar in values: about 12° for the first curve, and 5° for the second one. The plot with Kc=30 shows a bit lower values in correspondance of the first curve, due to the fact that the vehicles decelerate more when the curvature changes, and less oscillations are present.
In summary, at the beginning, Kc=30 causes sharper steering with respect to the other case; one stabilized, its behaviour on the curves is a little bit better, with less oscillations, in particular on the first curve.

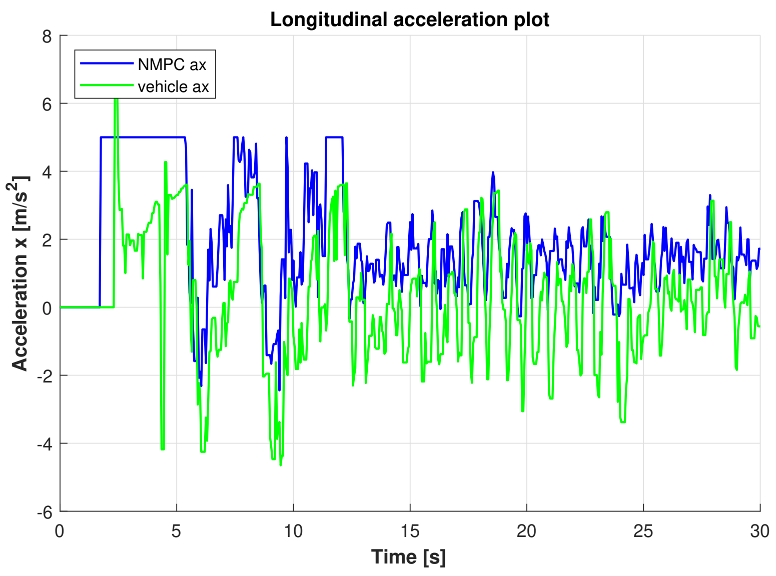


(a) Control Variables - Kc = 10

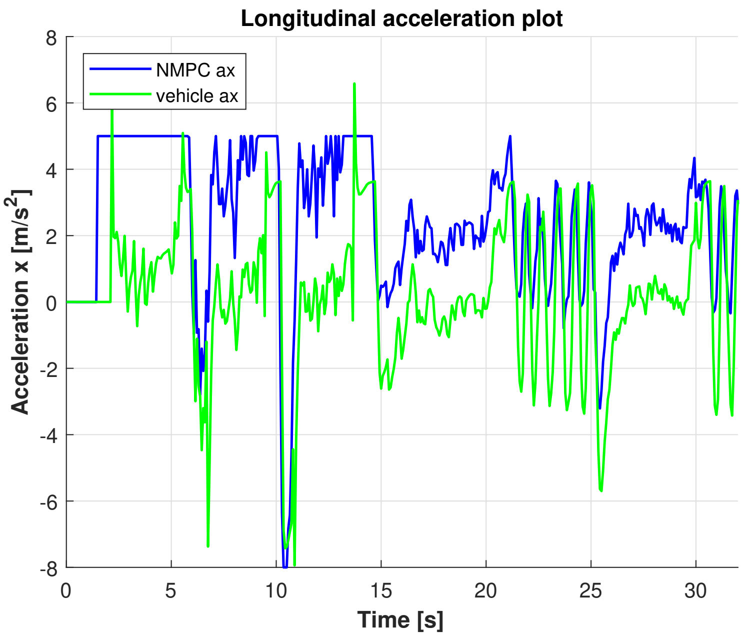


(b) Control Variables - Kc = 30

Figure 7.9: Comparison between Control Variables



(a) Accelerations - Kc = 10



(b) Accelerations - Kc = 30

Figure 7.10: Comparison between Accelerations - NMPC and Real

Figure 6.9: Control Variables plot

About *Throttle*, for both values of Kc, the throttle input exhibits significant variability, reflecting the controller's continuous effort to adapt to changing driving conditions. With Kc=10, the throttle transitions are relatively smoother, especially in the initial phase, with gradual increases and decreases. However, as time progresses, the controller becomes more active, leading to frequent oscillations in the throttle signal.

When the gain is increased to 30, the throttle profile changes noticeably. The transitions become more abrupt and frequent, particularly in the second half of the time window. This behavior indicates a more aggressive control strategy, where the controller reacts more promptly to the changes in curvature of the scenario. The higher gain amplifies the controller's sensitivity, resulting in sharper throttle commands that aim to adapt to the

curves more quickly.
The *Brake* input is generally sparse for both controller gains, as expected in a scenario where the primary objective is to maintain or increase speed.
For Kc=10, the brake is applied in short, isolated bursts, mainly during specific events such as deceleration or preparation for gear shifts. The magnitude of these brake applications is moderate, and their duration is brief.
With Kc=30, the brake interventions become even more distinct and isolated. The controller applies the brake in sharper, more pronounced peaks. This suggests that the higher gain leads to a more decisive braking strategy, where the controller opts for quick, strong interventions rather than prolonged, moderate braking, in particular when the vehicles encounters a curve.
Gear changes provide insight into how the controller manages the powertrain to optimize performance. For Kc=10, gearshifts occur less frequently throughout the simulation. The controller appears to be more cautious, making incremental adjustments to the gear ratio in response to changing speed and acceleration demands.
In contrast, with Kc=30, gear changes are more frequent and don't stabilize in the simulation, because, on the curves, the vehicle goes on 15 km/h (first gear) and 25 km/h (second gear), while the reference speed is 50 km/h, corresponding to the third gear. This higher curvature parameter causes more gearshifts with respect to the case with Kc=10.
The *predicted longitudinal acceleration* generated by the NMPC controller reveals important differences between the two gain settings. For Kc=10, the acceleration profile is characterized by higher variability and oscillations, especially in the early stages. The controller's moderate gain leads to a more cautious approach, with frequent corrections that manifest as fluctuations in the acceleration signal.
With Kc=30, the acceleration profile becomes more dynamic, with pronounced peaks and a tendency toward higher values. The controller's aggressive nature at this gain setting results in rapid changes in acceleration, reflecting its effort to quickly achieve the desired speed or trajectory. However, this increased responsiveness also introduces the risk of overshooting and greater oscillations, particularly during rapid transitions.

Figure 6.10: Comparison of Predicted and Actual Accelerations
For the lower gain, the NMPC and actual vehicle accelerations generally follow similar trends, with both signals rising and falling in tandem. However, there are noticeable discrepancies, particularly in the magnitude and timing of the peaks. The mismatch is especially evident during phases of high acceleration or sudden deceleration, where the real system's dynamics diverge from the controller's predictions. This due to the fact that the throttle and brake commands result in different accelerations between the dispatcher and the CARLA vehicle model.
Increasing the gain to 30 improves the synchronization between the NMPC and actual accelerations in certain regions. Both signals display sharper and more synchronized peaks, indicating that the controller is more effective at tracking the desired acceleration profile. However, the higher gain also amplifies the differences during rapid transitions. The NMPC prediction often overshoots compared to the real acceleration, and the oscillations become more pronounced. This behavior highlights the trade-off between responsiveness and stability: while a higher gain enables quicker corrections, it can also lead to aggressive commands that the real vehicle cannot always follow perfectly, resulting in transient mismatches.

Final Remarks and Discussion on the Curvature Parameter Choice

The comparative analysis between different values of the curvature parameter Kc clearly shows how the choice of this parameter profoundly influences the vehicle's dynamic behavior during path tracking. A lower value of Kc, such as 10, tends to produce a smoother and more gradual control action, with less abrupt changes in speed and steering. However, this comes at the cost of slightly reduced accuracy in following the desired trajectory. Conversely, a higher value, like 30, enhances the system's responsiveness, improving the precision in tracking the path but introducing more pronounced oscillations and sharper speed variations.
This dynamic reflects a classic trade-off between stability and responsiveness in autonomous vehicle control: an overly aggressive control strategy can lead to instability and discomfort, while a too conservative approach may limit the vehicle's ability to quickly adapt to changes in the path and environmental conditions. Moreover, the need to tune the parameter Kc for specific scenarios represents a significant limitation in terms of control system generalization. In real-world contexts, where vehicles must handle a wide variety of situations—ranging from tight curves to straight roads, and from low to high speeds—the ability to rely on a single, stable, and robust set of parameters is essential to ensure reliable and safe operation without the need for constant manual adjustments.

7.2 Suburban Junction

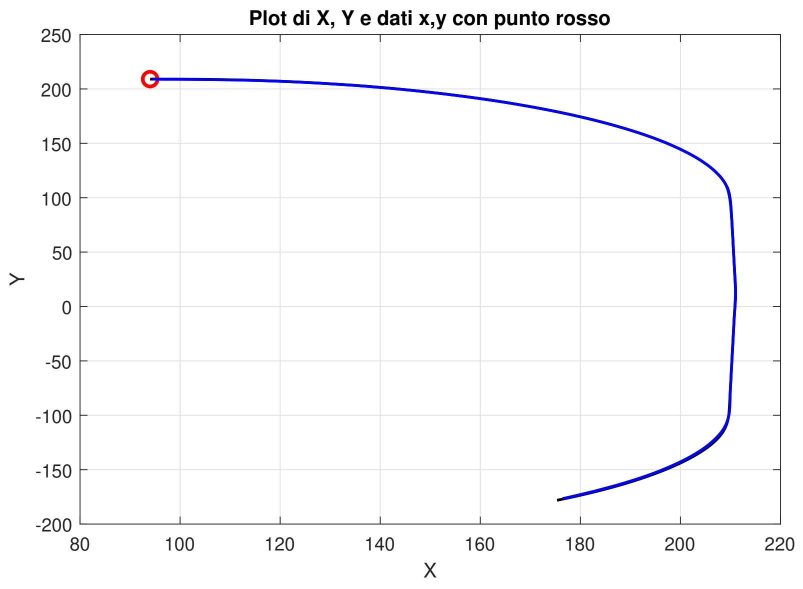


Figure 7.11: Suburban Junction with a wide curve

This scenario takes place in Town 04 of the CARLA simulator, a small town characterized by a picturesque mountain landscape with snow-capped mountains in the background and conifers. The special feature of this environment is its unique road configuration: a multi-lane road that encircles the town following a “figure of 8” layout.
Specifically, the scenario depicted in the image represents a portion of a suburban interchange road, and 2 simulations are conducted, with different conditions:

1. **Curve Behavior Without Traffic** The first simulation is conducted in a controlled environment without the presence of other vehicles. The objective is to analyze the behavior of the autonomous vehicle as it negotiates curves at a speed of 50 km/h. This setup allows evaluating the stability of the vehicle in curves and checking the accuracy of the navigation system along the predefined route.
2. **Overtaking Maneuver** The second simulation introduces an element of increased complexity with the addition of another vehicle in the track. The main objective is to test the autonomous vehicle's ability to perform a safe and smooth overtaking maneuver. This simulation is particularly challenging because:
   - Requires dynamic evaluation of the traffic situation.
   - Needs real-time lane-change decisions.
   - Must ensure safety by maintaining appropriate distances from other vehicles.

7.2.1 Curve Behavior Without Traffic

The graphs that will be shown in this simulation are of the same variables that were represented in the previous scenarios.
The first graph that is shown is the total speed that the vehicle reaches in path tracking this part of the road. The value of the parameter v0 is set to 44 km/h. At the beginning

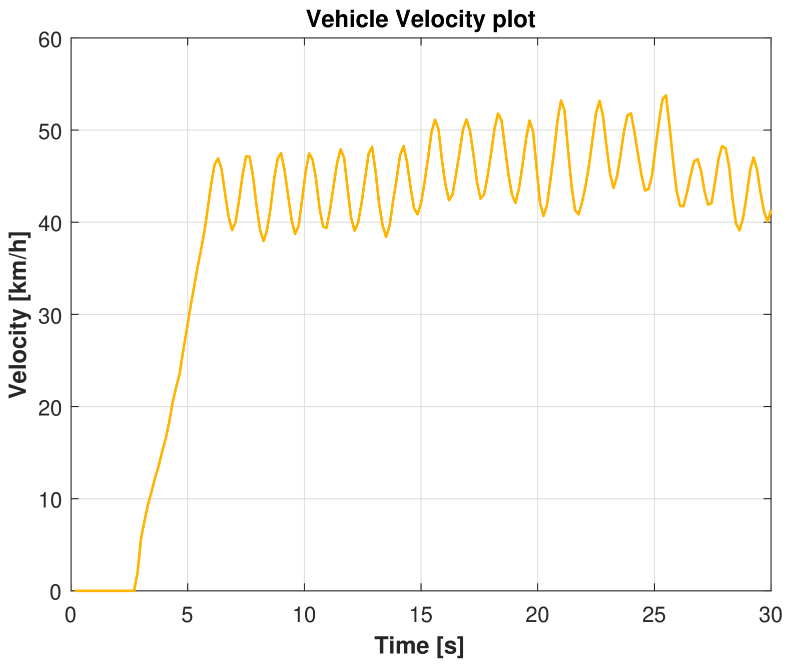


Figure 7.12: Vehicle Speed - Suburban Junction

of the simulation, in the first 5 seconds or so, the vehicle starts from a standstill and accelerates rapidly, reaching a speed of about 45 km/h almost immediately. This phase corresponds to the initial section with a gentle curvature, where the vehicle can increase speed without any particular hindrance.
After this initial acceleration, the speed stabilizes around 40-45 km/h, but the graph shows regular and rather marked oscillations. These oscillations are typical of a course characterized by frequent turns and changes of direction. Under these conditions, the

vehicle is forced to slow down slightly and then accelerate again, thus generating the speed variations seen in the graph.
Around 15 s, it can be seen that the oscillations become slightly larger, due to the almost abrupt change in the curvature of the path (change from gentle curve to almost straight track), and the maximum speed reached approaches 50 km/h, a sign that the vehicle is trying to maintain the target speed but must continuously adapt to the path conditions. After 25 s, the vehicle exits the straight section, traveling through a gentle curvature, similar to the initial section of road, and consequently, the value and trend of the speed returns similar to the initial section of road.
The plot below is that of the Cross-track error. At the beginning (0-3 s), the error is

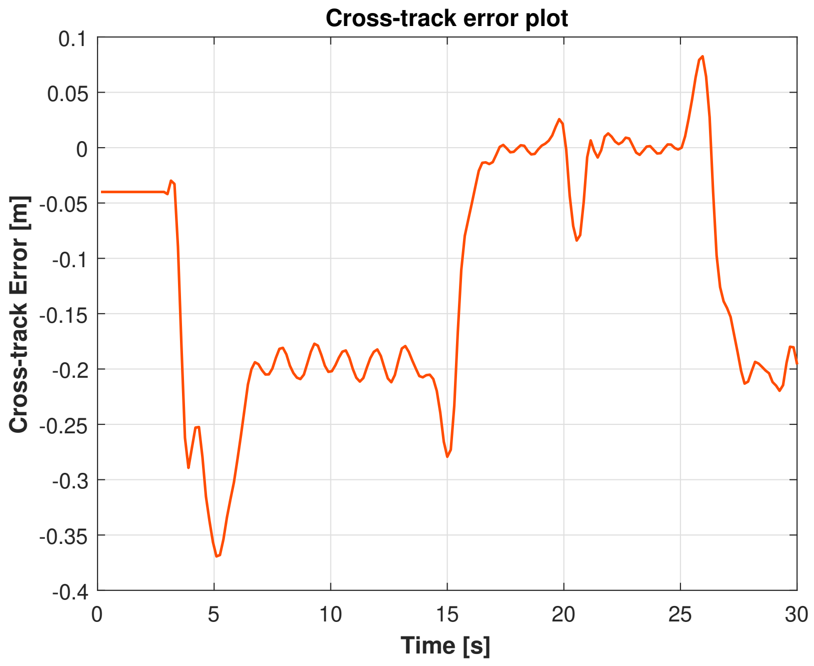


Figure 7.13: Cross-track Error

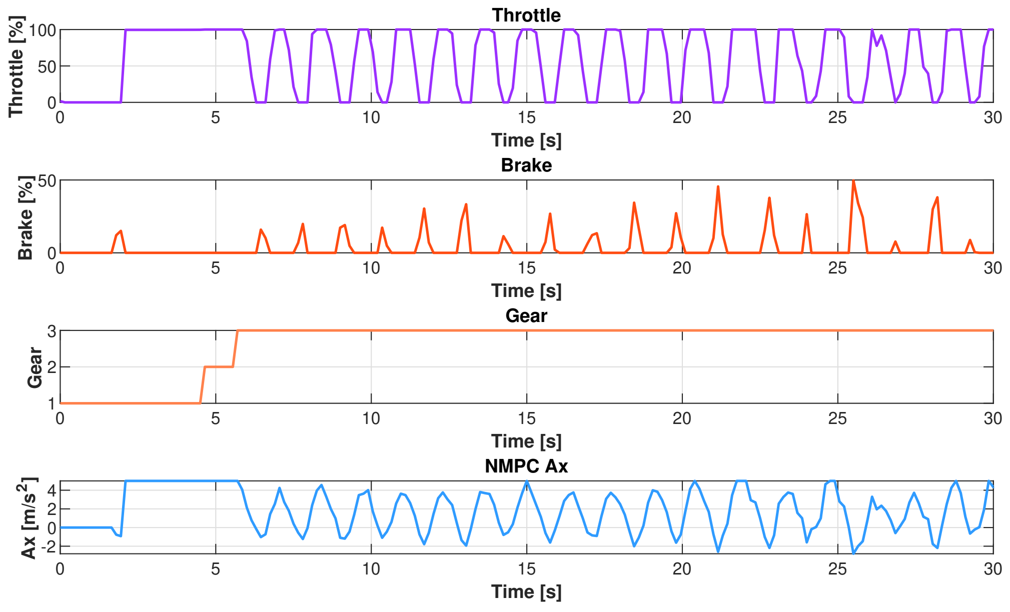
close to zero, then (3-6.5 s) it rapidly decreases to about -0.37 m. This indicates that the vehicle, immediately after departure, deviates to the left from the ideal trajectory as it aligns with the path.
In the time interval (6.5-14) s, after the initial correction, the error fluctuates, but generally remains within -0.2 m. These oscillations are normal on such a challenging course: the vehicle must constantly correct its position to follow the trajectory, especially in curves. On the 15 s, the vehicle arrives in the section with abrupt change of curvature, generating an almost instantaneously corrected error of about -0.27 m, and as it travels along a nearly straight section, the error fluctuates around 0.
Toward the end, some larger oscillations are noticeable, with the error even reaching positive values (up to about 0.08 m), a sign that the vehicle shifts slightly to the right to travel the abrupt straight-curve change of trajectory. The control system always manages to bring the vehicle back close to the center of the lane and adapt dynamically to the curvature changes.
The RMS (Root Mean Square) value of the lateral error represents the mean square deviation of the vehicle from the ideal trajectory throughout the simulation. A value of 0.1636 m indicates that, on average, the vehicle deviates about 16 cm from the ideal

RMS e_{ct} [m]	**Max e_{ct} [m]**	**ASSESSMENT**
0.1636	0.3693	**Great**

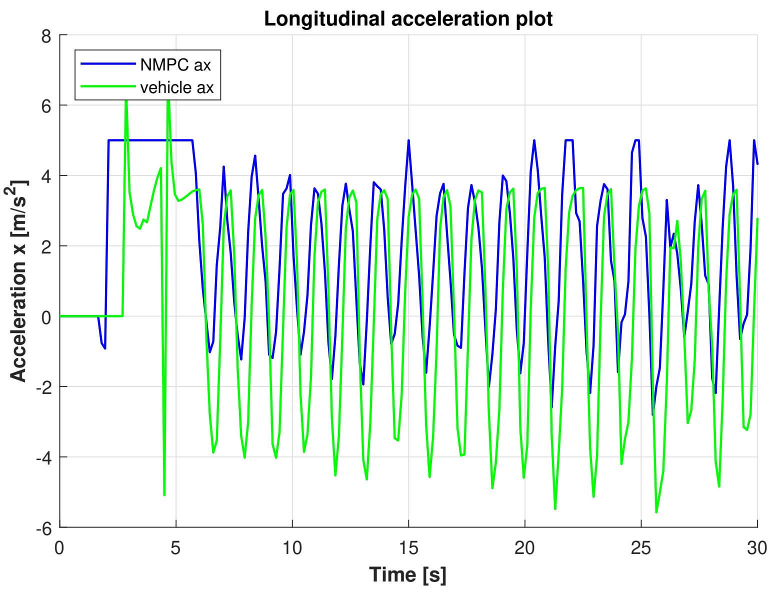
trajectory-a very good result, especially considering the complexity of the track.
The value of 0.3693 m (about 37 cm) as maximum error is also very small, especially with tight turns and frequent changes of direction.
It represents a great result, almost excellent, considering that the track is characterized by some abrupt changes of curvature and the reference speed of 50 km/h (it would be excellent if the maximum cross-track error was below 0.3 m).

Plot a): Control Variables
The graph shows how the vehicle's control system manages throttle, brake and gear shifting to follow a desired longitudinal acceleration profile.
The first subgraph shows the percentage of throttle opening over time. At the beginning, the throttle is quickly raised to full throttle and held high for a few seconds, probably to start the vehicle. Thereafter, a very regular oscillatory pattern is observed, with rapid and repeated opening and closing cycles. This indicates that the control system is modulating the throttle continuously to adjust its speed according to changes in curvature.
The second subgraph shows the use of the brake. In the first few seconds, the brake is used only sporadically, then it begins to be activated cyclically, corresponding to the throttle oscillations. This pattern suggests that the control rapidly alternates between acceleration and braking to maintain the required pattern.
The third subgraph shows the gear engaged. At first, the vehicle starts in first gear, then quickly shifts to second and finally to third, where it remains for the rest of the simulation. This behavior is typical of a standing start followed by a higher speed driving phase, where it is no longer necessary to shift gears frequently.
The last subgraph shows the longitudinal acceleration profile calculated by the Nonlinear Model Predictive Control (NMPC) controller. A rapid initial growth can be seen, followed by a series of regular oscillations between positive and negative values. These oscillations correspond to the acceleration and braking cycles observed in the first two subgraphs, indicating that the controller is commanding rapid changes in acceleration to follow a certain dynamic target.

Plot b): Comparison between NMPC and Real Accelerations
At the beginning (first 5 seconds or so), both accelerations increase rapidly, with some initial peaks and swings due to the start-up phase and the beginning of the cornering section. In particular, the NMPC ax has a maximum acceleration (for starting from a standstill), while the vehicle ax has 3 acceleration peaks: the first due to starting from a standstill, the second (negative) due to shifting into neutral gear to change from first to second gear, and the third due to shifting into second gear.
After the first few seconds, both curves show a smooth oscillatory pattern, with the blue line (NMPC ax) following a slightly smoother and more regular profile than the orange line (vehicle ax). The orange line (vehicle ax) shows more pronounced oscillations and lower minimum values than the blue line, indicating that the actual vehicle undergoes greater changes in acceleration with respect to the controller's ones, due to differences in the schematization of the longitudinal dynamics principally.
As we saw in the first scenario, there are differences between NMPC and Real Accelerations values. The dispacher function allows maximum negative accelerations in third gear



(a) Control variables



(b) NMPC Acceleration versus Real Acceleration

Figure 7.14: Throttle, Brake and Acceleration

equal to -5.46 m/s^2, so 0.3-0.4 of Brake correspond to about -2 m/s^2. From the red plot (vehicle accelerations) we can see that, for those brake values, we obtain about -5-6 m/s^2, generating a more oscillatory behaviour.

The following plot represents the variation of the steering angle during the scenario tracking.
In the first interval (0-5 s), the steering angle shows some rather pronounced oscillations, with both positive and negative peaks. This behavior is typical of the vehicle start-up phase, when the control system is probably trying to stabilize the trajectory or correct

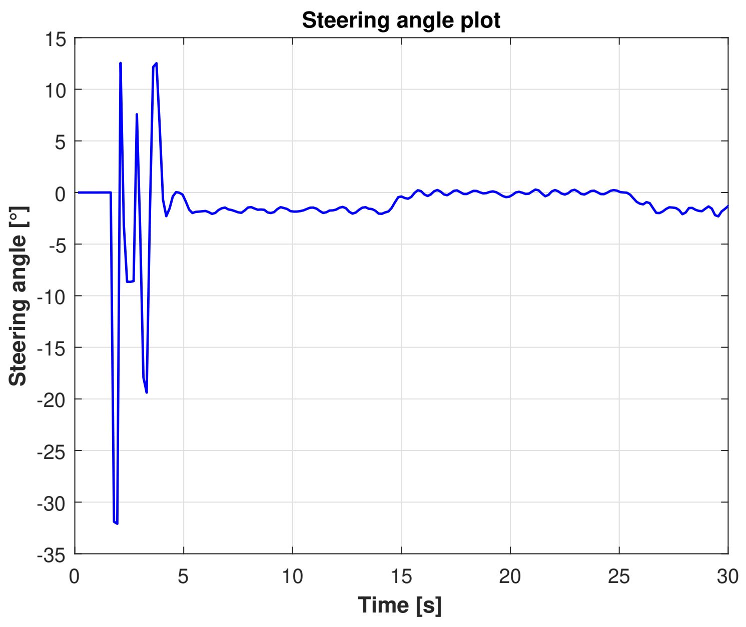


Figure 7.15: Steering Angle

any initial deviations from the desired course and to reactions to intense acceleration and braking inputs, as seen in the other graphs.
After the initial (5-30)s phase, the curve stabilizes: the steering angle remains very close to zero, with only small oscillations. This indicates that the vehicle is proceeding almost in a straight line, with no need for major directional corrections.
This behavior is consistent with what was observed in previous graphs, where the focus was mainly on longitudinal control (acceleration and braking), while the lateral component (steering) is still well controlled.

7.2.2 Overtaking

The overtaking maneuver is the action by which a vehicle overtakes another vehicle, animal or pedestrian moving or stationary on the roadway, moving sideways to position itself in front of the overtaken vehicle in the same direction of travel. It is generally performed to the left, requiring careful consideration of space and safety to avoid collisions, especially with oncoming vehicles from the opposite direction.
The maneuver is especially important in suburban scenarios because these roads, being outside population centers, have higher speeds (up to 110 km/h on major suburban roads) and less dense traffic, making overtaking a frequent maneuver to maintain traffic flow and traffic safety. However, overtaking is one of the most dangerous maneuvers, especially in suburban environments where visibility may be limited and the road may be two-way, so it requires attention and compliance to be performed safely.
The maneuver was implemented on the same road seen in the previous simulation, with the addition of a vehicle, placed 30 m ahead of the ego vehicle, equipped with autopilot traveling at a speed of 30 km/h (there are many maximum speed limits of 30 km/h in that scenario).
To better handle this maneuver, a Matlab Function, called "*check_overtaking*," was implemented that initiates the overtaking maneuver under certain conditions. The script of that function is placed below.

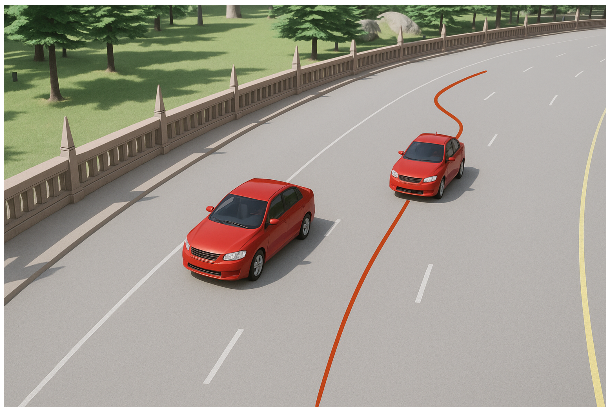


Figure 7.16: Overtaking

Listing 7.1: Funzione check_overtaking in MATLAB

```
function [flag,dist] = check_overtaking(ze2, ze)

flag=0; % inizializzazione del flag
x=ze(1);
y=ze(2);
vx=ze(4);
x2=ze2(1);
y2=ze2(2);
vx2=ze2(4)

% Sorpasso
dist=vecnorm(ze2(1:2)-ze(1:2));
if dist<15 && vx >= vx2
    flag=1;
end
end
```

At first, the function takes as input the state of the vehicle in front (ze2) and the state of its own vehicle (ze). The positions and speeds of the two vehicles are then extracted. The heart of the function is checking the conditions for overtaking:

- The distance *dist* between the two vehicles is calculated.
- A check that your vehicle is close enough to the one in front is done (less than 15 meters), that it is still behind or nearly aligned.
- If all these conditions are true, the function signals that the overtaking should be done (*flag=1*).

The ax variable calculated using this function is given as input to the *dispatcher* block, and used to evaluate throttle and brake commands.
Therefore, the *flag* variable is linked to the *ref_gen* path planning function. Next, if the overtaking flag is active, the reference trajectory (*ref_tr*) is changed to simulate the lane change: the yaw angle *psi* of the trajectory is calculated and the trajectory is shifted laterally by 3.5 meters so that the vehicle is brought into the overtaking lane.
The following plots represent the variation in time of the variables during this maneuver. The graph titled "Vehicle Velocity plot" provides a clear visualization of how a vehicle's speed changes during an overtaking maneuver, as shown over a 15-second interval. At the

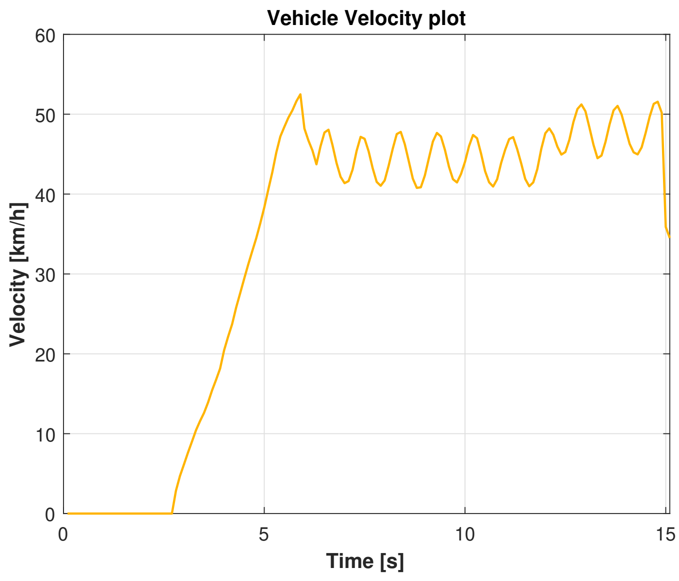
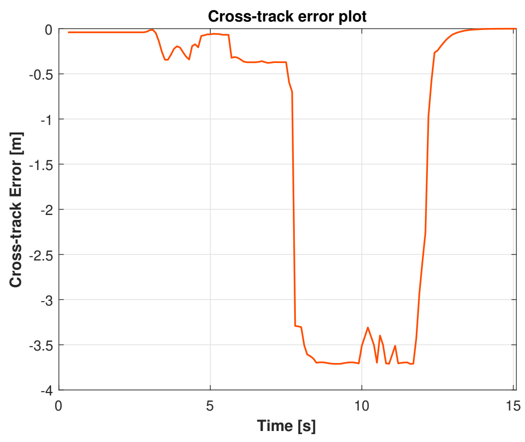


Figure 7.17: Overtaking Vehicle Speed

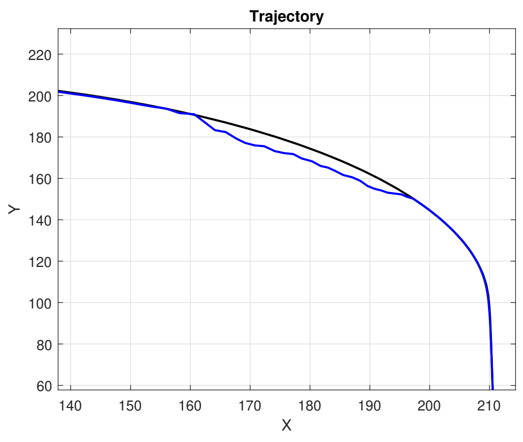
start, the vehicle is stationary, but it quickly accelerates, reaching a speed of about 50 km/h within the first 5 seconds.
After reaching its peak speed, the graph shows a series of oscillations in the velocity, fluctuating between roughly 40 and 50 km/h. We can observe a decrease in speed at the very moment the overtaking maneuver begins. This reduction in velocity, as well as the oscillations that follow, are mainly due to a significant change in the curvature of the path. In other words, as the vehicle adopts a new reference trajectory for the overtaking maneuver, the road's curvature increases, requiring the control system to adjust the speed accordingly. The oscillations in the velocity are a direct consequence of these continuous adjustments as the vehicle adapts to the new, more complex trajectory. Despite the lower speed and the fluctuations caused by the increased curvature, the vehicle is still able to successfully complete the overtaking maneuver. This demonstrates that even with these dynamic challenges, the system remains effective and the overtaking is achieved safely.
Towards the end of this time period, there is an increase in speed, which likely marks the completion of the overtaking process.

Plot a): Cross-track Error
Looking at the curve, we can see that the error remains fairly constant and contained for most of the time, but at some point there is a more pronounced deviation due to the occurrence of the overtaking maneuver: the error drops sharply, reaching a negative peak (indicating that the vehicle deviates to the left of the reference trajectory), and then quickly returns to its previous values. The vehicle moves to the center of the left passing lane, 3.5 m away from the center of the lane where the other vehicle is. However, the NMPC system manages to quickly correct the trajectory and return the vehicle close to the ideal path.

Plot b): Trajectory Deviation
The second graph, on the other hand, represents the deviation from the reference trajectory. Here it shows how the vehicle, during its path, deviates from the ideal trajectory it should follow. The curve shows the trend of the position relative to the reference trajectory over time or along the traveled space. In this case, the deviation appears quite regular and



(a) Cross-track error



(b) Trajectory deviation

Figure 7.18: Deviations from the trajectory

progressive: there are no sudden peaks or swings, but rather a gradual change. This type of graph is useful for understanding how closely the vehicle follows the programmed trajectory and whether there are systematic errors or tendencies to deviate consistently from it.

The success of this maneuver shows that the implemented NMPC system is able to complete even complex maneuvers that require interaction with other road agents.

Chapter 8

Complex Scenarios

In the context of autonomous driving, handling lane merging scenarios is one of the most complex challenges, especially when dealing with urban and suburban environments characterized by high traffic density and unpredictable dynamics. The analysis carried out focused on three main scenarios: urban lane merging, suburban lane merging, and multi-vehicle traffic roundabout merging, with the objective of evaluating the performance of the ego-vehicle under realistic traffic conditions and stringent safety and driving comfort constraints.
These scenarios are scarcely discussed in the literature, given their complexity, and are also developed entirely on MATLAB/Simulink.

8.1 Sensors and Other Instruments

In the simulations of those scenarios, the assumption of instant-by-instant knowledge of the trajectories and speeds of other vehicles, even without direct communication between vehicles, can be justified by the combined use of advanced sensors, location systems, and on-board computational tools.
In order to reconstruct the position and speed of other vehicles in real time, AD systems rely on a suite of sensors that work synergistically:

1. **Radar**: enables the measurement of the relative distance and speed of surrounding objects, proving particularly reliable even in low visibility conditions.

2. **LiDAR**: provides high-resolution three-dimensional mapping of the environment, enabling accurate detection of the location of other vehicles on the road.

3. **Cameras**: using computer vision techniques, allow vehicles to be identified, tracked and classified, estimating their trajectory and speed by comparing successive frames.

A key role in trajectory reconstruction is played by satellite tracking systems:

- **GPS**: provides the absolute position of the vehicle with an error on the order of a few meters. This data, when integrated with information from on-board sensors, allows the trajectory traveled to be estimated.

- **Differential GPS (DGPS)**: improves the accuracy of traditional GPS by reducing the error to a few centimeters through the use of ground reference stations. In complex scenarios and with multiple vehicles equipped with DGPS, it is possible to obtain a very accurate estimate of the trajectories of each vehicle, even without direct communication between them.

In addition, another tool assumed to be on board the controlled vehicle is a computing unit, typically a **microcontroller** or advanced control unit. All data acquired from the sensors and the tracking system are processed in real time by this device, which also performs data fusion (sensor fusion) and state estimation algorithms, such as Kalman filters or multi-object tracking techniques, which allow the trajectory and velocity of each vehicle detected in the surrounding environment to be continuously reconstructed.
In this thesis work, this comprehensive knowledge of the trajectories and speeds of other vehicles was implemented using MATLAB's *Automated Driving Toolbox*, by using **"Driving Scenario Designer"** app, present in the MATLAB database.

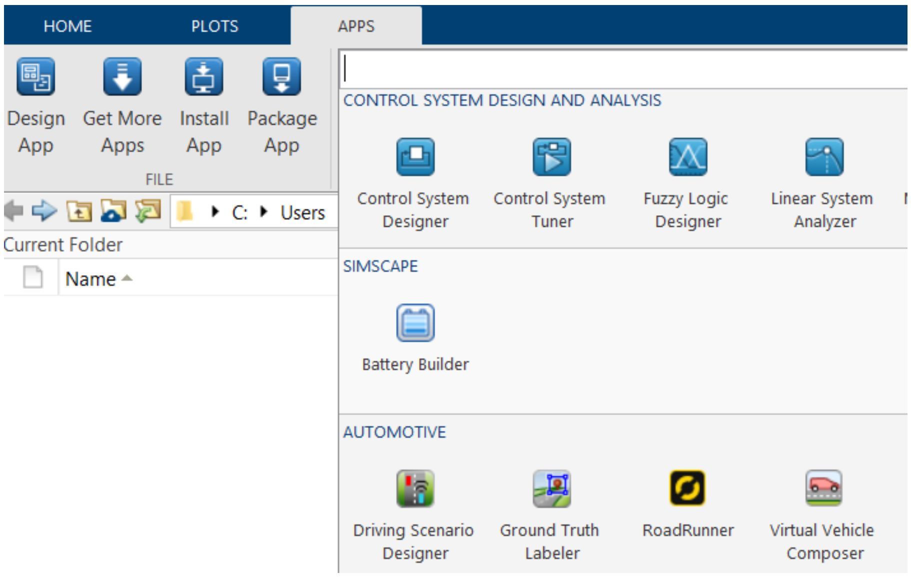


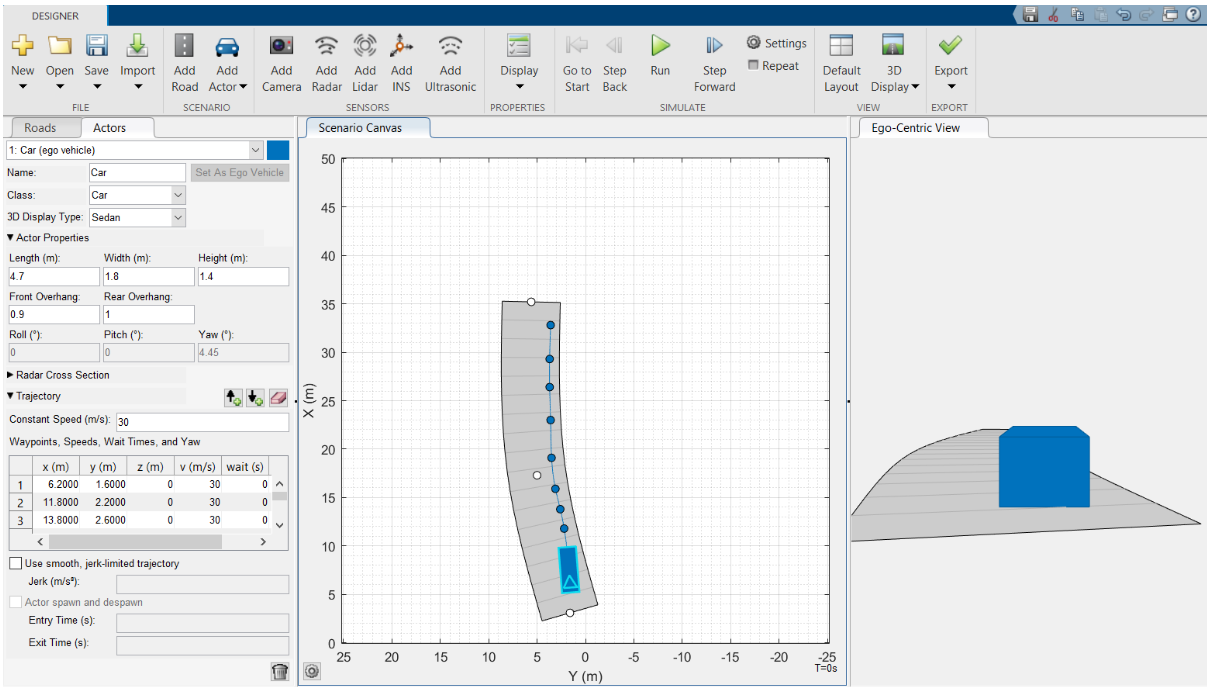
Figure 8.1: MATLAB's APPS Database

Real roads were used, on which the trajectory points of all vehicles involved in the scenario were drawn. Then, the trajectories were then interpolated based on the speed of each vehicle and temporally synchronized, so that traffic behavior could be realistically simulated and strategies for route planning and ego-vehicle control could be evaluated. This procedure mirrors what would happen in a real system, where the trajectories of other vehicles are continuously estimated based on data provided by sensors, GPS/DGPS, and calculations performed by the on-board microcontroller, without the need for inter-vehicle communication. In this way, the simulation maintains a high degree of realism and enables reliable testing of the control and management logic of complex lane merging scenarios.

8.2 Obtaining Trajectories: Methodology and Implementation

After introducing the sensors used for environmental perception and estimation of the trajectories of other vehicles, these semi-chapters describe in detail the methodology adopted for obtaining the trajectories, the function of constraints, and the choice of interpolation techniques, with particular reference to the implementations made through

MATLAB's Automated Driving Toolbox.



1: Car (ego vehicle)

Name: Car | Set As Ego Vehicle

Class: Car

3D Display Type: Sedan

▼ Actor Properties

Length (m): 4.7 | Width (m): 1.8 | Height (m): 1.4

Front Overhang: 0.9 | Rear Overhang: 1

Roll (°): 0 | Pitch (°): 0 | Yaw (°): 4.45

▶ Radar Cross Section

▼ Trajectory

Constant Speed (m/s): 30

Waypoints, Speeds, Wait Times, and Yaw

	x (m)	y (m)	z (m)	v (m/s)	wait (s)
1	6.2000	1.6000	0	30	0
2	11.8000	2.2000	0	30	0
3	13.8000	2.6000	0	30	0

☐ Use smooth, jerk-limited trajectory

Jerk (m/s³):

☐ Actor spawn and despawn

Entry Time (s):

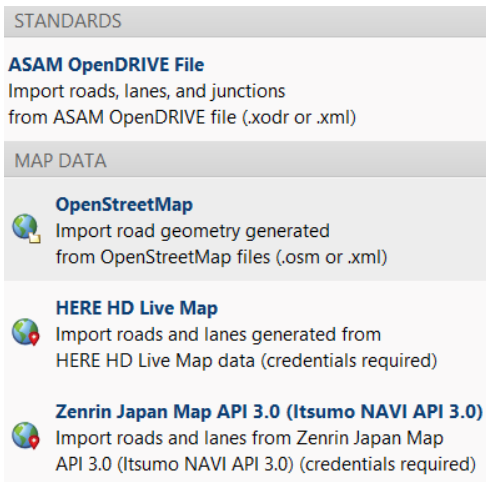
Exit Time (s):

Figure 8.2: Driving Scenario Designer interface

The attached image shows the interface of MATLAB's Driving Scenario Designer. In the left panel ("Actors"), a vehicle can be added (e.g., "Car (ego vehicle)") by specifying its geometric properties (length, width, height, overhang, yaw, etc.). The "Trajectory" section allows to define the vehicle's trajectory by a sequence of waypoints (imposed by us), i.e., points in space (X, Y, optionally Z) that the vehicle is to follow. Each waypoint can be associated with a speed and a waiting time. To decide the waypoints, it is needed to click on the vehicle, and select "Add Forward Waypoints" or "Ctrl+F" command.
Instead of compose manually the trajectory, the waypoint coordinates can be also added, as the speed at each point and the wait time ("wait (s)"). The buttons next to the table allow to add, remove or edit points.
In the "Constant Speed (m/s)" section, you can set a constant speed for the entire section or specify different speeds for each segment.
The blue dots on the map represent waypoints, while the line connecting them is the trajectory the vehicle will follow.
To create scenarios on real roads, real map data can be imported from OpenStreetMap (OSM). This is the procedure, step-by-step, to be followed:

1. **Download OSM data**: go to «*openstreetmap.org*», select the area of interest, and download the .osm file.

2. **Import OSM data**: using the command "Import –> OpenStreetMap", the selected road in osm format can be imported into the interface.


STANDARDS

ASAM OpenDRIVE File
Import roads, lanes, and junctions
from ASAM OpenDRIVE file (.xodr or .xml)

MAP DATA

OpenStreetMap
Import road geometry generated
from OpenStreetMap files (.osm or .xml)

HERE HD Live Map
Import roads and lanes generated from
HERE HD Live Map data (credentials required)

Zenrin Japan Map API 3.0 (Itsumo NAVI API 3.0)
Import roads and lanes from Zenrin Japan Map
API 3.0 (Itsumo NAVI API 3.0) (credentials required)

Figure 8.3: Import - CreateDrivingScenario

3. **Adapting and defining trajectories**: once imported the map, you can place vehicles on real roads and define their waypoints following the real roadway geometry. You can then edit the waypoints for each vehicle so that they faithfully follow the real roadway.

After defining vehicles' trajectories, the scenario can be saved into other MATLAB scripts or into Simulink using the "Export" command.
Because the imported or manually defined waypoints may be few or irregular, interpolation of the trajectories is performed to obtain a smoother, sampled path with the desired resolution.

Listing 8.1: Trajectories generation

```
waypoints = [
    ];
Ts = 0.05;
dx = diff(waypoints(:,1));
dy = diff(waypoints(:,2));
ds = sqrt(dx.^2 + dy.^2);
ls = sum(ds); %lunghezza strada
vx_ref = 90/3.6;
T = ls / vx_ref; %tempo per percorrere la strada

% Calcolo passo temporale e distanza per campionamento
s = [0; cumsum(ds)];
step_dist = vx_ref * Ts; % Distanza percorsa a ogni Ts
distanze_campionate = 0:step_dist:ls; % Punti di interpolazione

% Interpolazione delle coordinate X e Y (linear)
Xr = interp1(s, waypoints(:,1), distanze_campionate, 'linear');
Yr = interp1(s, waypoints(:,2), distanze_campionate, 'linear');

% Interpolazione delle coordinate X e Y con makima
Xr = makima(s, waypoints(:,1), distanze_campionate)';
Yr = makima(s, waypoints(:,2), distanze_campionate)';

% Time Synchronization
min_length = min([length(Xr), length(Xr1), length(Xr2), length(Xr3)]);
Xr = Xr(1:min_length);
```

```
Yr = Yr(1:min_length);
Xr1 = Xr1(1:min_length);
Yr1 = Yr1(1:min_length);
Xr2 = Xr2(1:min_length);
Yr2 = Yr2(1:min_length);
Xr3 = Xr3(1:min_length);
Yr3 = Yr3(1:min_length);

% Calcolo del tempo di simulazione
Tfin = Ts*min_length;

% Salva le traiettorie in un file .mat
save('Trajectory_name.mat', 'Xr', 'Yr','Xr1', 'Yr1','Xr2', 'Yr2','Xr3', '
    Yr3');
```

The first step is to calculate the distances *ds* between each pair of consecutive waypoints, using the Euclidean distance formula. Adding up all these distances gives the total length *ls* of the trajectory to be traveled by the vehicle. Knowing the desired reference speed (e.g., 90 km/h converted to meters per second), the total time T required to travel the entire trajectory can be calculated.
At this point, you define the time step *Ts* of the simulation, that is, every how long you want to update the position of the vehicle (e.g., every 0.05 seconds). Multiplying the velocity by the time step yields how many meters the vehicle moves at each time interval. Thus, a vector of sampled distances is constructed, representing all the positions along the trajectory where you want to know the position of the vehicle.
To obtain the exact coordinates (X and Y) corresponding to these distances, interpolation is used. Basically, you “fill in” the missing points between the known waypoints, generating a continuous, smooth trajectory. Different interpolation methods can be chosen:

- **Linear**: ideal for straight stretches or with sharp changes of direction, providing simplicity and fidelity to the original data.
  Linear interpolation, on the other hand, connects points with straight segments, generating discontinuities in the first derivative of the interpolated function. This can lead to abrupt changes in the target velocity, making Kc (curvature parameter, discussed in the next chapters) tuning less effective or less “sensitive.” By increasing Kc, the system response may be less smooth and require more in-depth parameter tuning to avoid undesirable behavior such as sudden or insufficiently marked decelerations.

- **Makima**: produces smoother trajectories than linear, avoiding spurious oscillations typical of splines, and is therefore preferred for curvilinear stretches or in suburban scenarios.
  This type of interpolation provides more regularity and continuity of the first derivative than linear interpolation. This results in a more gradual and realistic variation of the target speed along the curvature profile. As a result, the tuning effect of the Kc parameter is more noticeable and controllable: by increasing it, the vehicle decelerates consistently and predictably in areas of high curvature.

When there are multiple vehicles in the scenario, each with its own trajectory, it is important that all trajectories have the same length and are temporally synchronized. For this reason, we cut all the vectors of the trajectories to the same minimum length, so that each vehicle is updated at the same time instant during the simulation.
After that, the total simulation time *Tfin* is calculated by multiplying the number of

trajectory points by the chosen time step. This ensures that the simulation is consistent and that all vehicles move synchronously along their respective trajectories.
Finally, the so-obtained trajectories are saved in a file .mat, imported to run the scenario. This procedure yields realistic and well-sampled trajectories, which are essential for simulation and validation of autonomous driving algorithms on real roads.

8.3 MATLAB/Simulink Full System

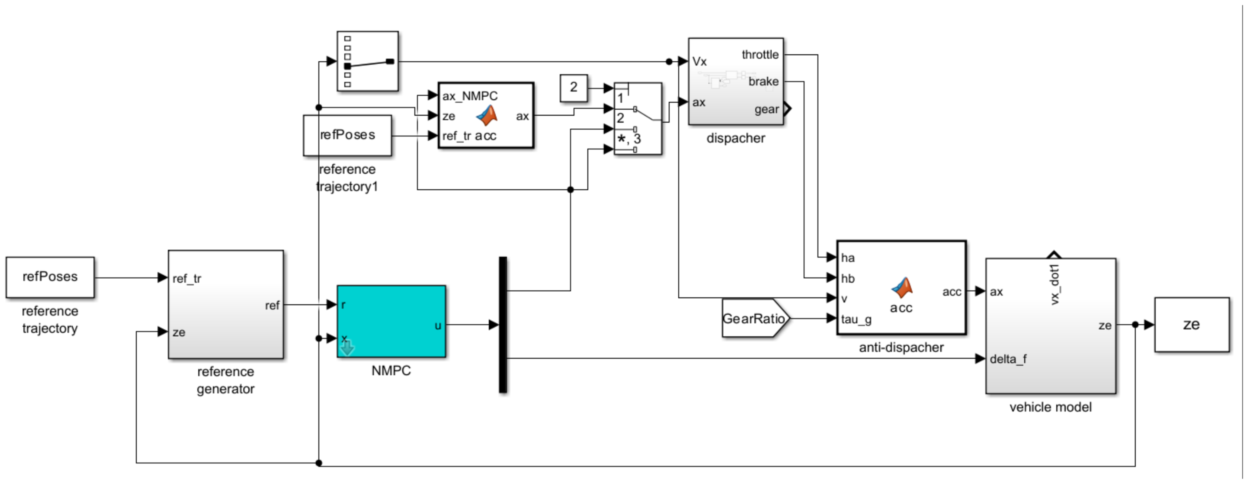


Figure 8.4: Simulink System - Complex Scenarios

8.3.1 "dispatcher" and "anti-dispatcher" Blocks

Regarding the Simulink subsystem called the "*dispatcher*", it is the same one that was described in section 5.3.4. It consists of 2 Matlab Functions: one that decides the gear to be used in real-time, depending on the longitudinal velocity *Vx* of the vehicle, and the second that converts the NMPC longitudinal acceleration *ax* into throttle *ha* and brake *hb* commands, then saturated in the range [0, 1], and given as input to the "anti-dispatcher" block.

Listing 8.2: "Anti-Dispatcher" block

```
function acc  = acc(ha,hb,v,tau_g)

% Parameters (SEAT LEON)
m = 1318;                  % kg
Jw = 1.4;                  % kg*m^2 (moment of inertia of wheels)
Jt = 0.02;                 % kg*m^2 (assumed, as not provided)
Je = 1;                 % kg*m^2 (assumed, as not provided)
Te_max = 468;              % N*m
Tb_max = 4*600;            % N*m (maximum brake torque per wheel*4)
eta = 0.85;                 % \ (assumed efficiency)
tau_d = 2.6;               % \ (assumed)
Cx = 0.300;                % \ (drag coefficient)
Cr = 0.014;                 % \ (rolling resistance coefficient, assumed)
rho = 1.225;               % kg/m^3 (air density)
n = 2;                     % \ (number of driven wheels)
n_nd = 2;                  % \ (number of non-driven wheels)
r = 33.5 / 100;            % m (wheel radius)
```

```
A = 2.66;                 % m^2 (frontal area, stimed by SW)

%1st model of long. dynamics
me_tr = m + Je*(tau_g*tau_d/r)^2 + Jt*(tau_d/r)^2 + (n+n_nd)*Jw/r^2;
me_bk = m + (n+n_nd)*Jw/r^2;
Fres = 0.5*rho*Cx*A*v^2 + Cr*m*9.81;
if ha > 0
     acc = (ha*Te_max*eta*tau_d*tau_g/r - Fres)/me_tr;
elseif hb > 0
          acc = -(hb*Tb_max/r + Fres)/me_bk;
else
          acc = 0;
end
```

The main innovation in this Simulink system lies in the block called the anti-dispatcher, which plays a fundamental and complementary role to the traditional dispatcher.
Whereas the dispatcher typically takes as input a desired longitudinal acceleration and converts it into throttle and brake commands, the anti-dispatcher does the reverse: it takes as input the throttle and brake signals and converts them into a real longitudinal acceleration to be provided to the vehicle dynamic model. The data of the vehicle are the same as the dispatcher, and a logic if-then-else is implemented according to ha and hb values:

- if **ha $>$ 0** (and consequently hb $= 0$): acceleration would be > 0, so the inverse formula of ha is used to evaluate ax values.

- if **hb $>$ 0** (consequently ha $= 0$): the vehicle is braking, so the negative acceleration is evaluated as the inverse of the hb expression.

- if **both ha and hb $=$ 0**: the vehicle is at standstill or it is traveling at constant speed, so ax $= 0$.

This reconversion is crucial for two main reasons.
The former is the consistency with real vehicle dynamics; the anti-dispatcher calculates the longitudinal acceleration taking into account the physical and mechanical limitations of the vehicle, such as mass, inertias, maximum engine and brake torques, and aerodynamic and rolling resistance. In this way, the resulting acceleration is more closely matched to reality and reflects the actual maximum and minimum limits that the vehicle can achieve under all driving conditions.
The latter is represented by the exceeding NMPC model limits; in traditional NMPC (Nonlinear Model Predictive Control) models, a fixed limit is often imposed on maximum acceleration, e.g., 5 m/s^2, regardless of the gear engaged. This can lead to unrealistic situations, such as imposing too high a maximum acceleration for higher gears, where the actual acceleration capacity decreases.
The "anti-dispatcher" solves this problem because, starting directly from the throttle and brake controls, it calculates the actual acceleration that the vehicle can actually develop in that gear and condition, avoiding artificial saturations and making the model more realistic and reliable.

8.3.2 "vehicle model" Subsystem

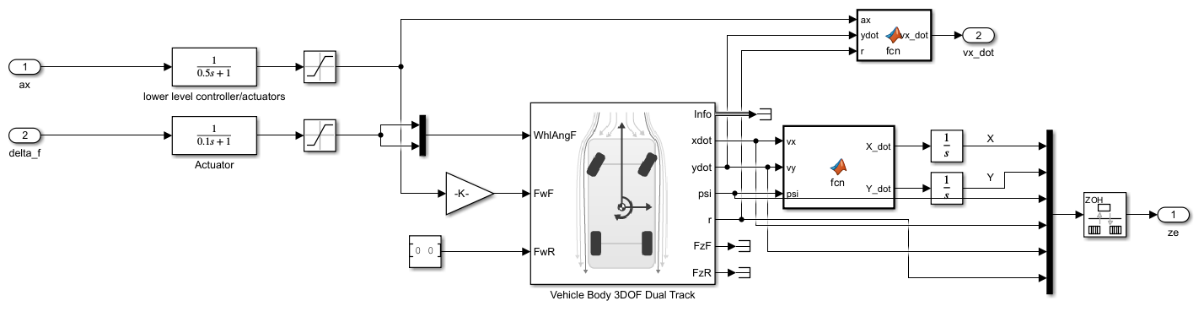


Figure 8.5: "vehicle model" Subsystem

In this subsystem, there are several blocks, each with a specific role for vehicle dynamics: actuators, the vehicle block (3DoFs dual track model), and 2 Matlab Functions that convert the variables output from the vehicle into useful states and variables for vehicle dynamics analysis.
The subsystem inputs are the Steering angle *delta_f*, coming from the NMPC, and the longitudinal acceleration *ax*, coming from the "anti-displacement" block. The system outputs are the *ze* states of the vehicle and the actual longitudinal acceleration of the vehicle *vx_dot*.

Actuators Blocks

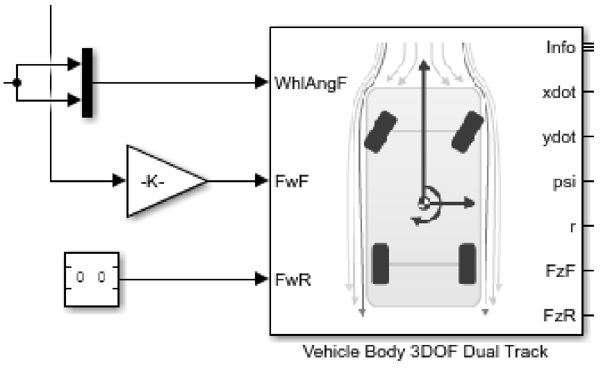
Actuators are critical because they transform abstract commands (required acceleration and steering angle) into realistic physical actions, taking into account the limitations and dynamics of real actuators. This makes it possible to simulate real vehicle behavior, where commands are not applied instantaneously but follow a certain dynamic due to mechanical and electronic inertias. For this reason, they are modeled using first-order LPFs (low pass filters) that simulate the dynamic response of the real actuators (throttle and steering).
Regarding the values used for the time constants used for LPFs:

- $\tau = \mathbf{0.5}$ **s** for *ax* represents the inertia of the traction system; when the accelerator or brake pedal is pressed, the vehicle does not respond instantaneously. The delay is due to various factors: engine inertia, electronic control system response, transmission, rotating mass inertia, etc.
  Studies and literature on vehicular modeling (e.g., "Vehicle Dynamics and Control" by Rajesh Rajamani) report time constant values between 0.3 and 1 second for longitudinal response of road vehicles, so the choice of 0.5 s is a good compromise between realism and simplicity.

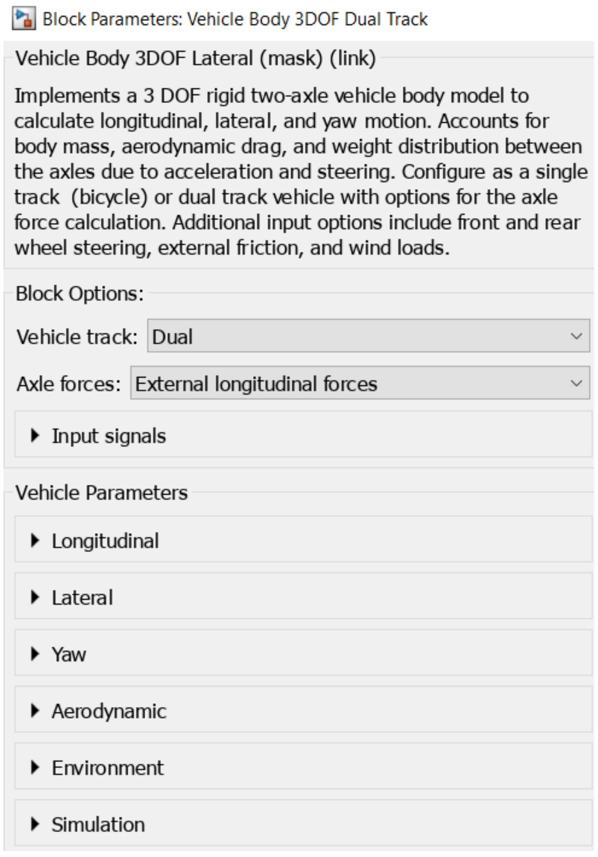
- $\tau = \mathbf{0.1}$ **s** for the steering angle to give the vehicle a quicker response; the steering system, especially in modern vehicles with electric power steering, is designed to respond quickly to driver commands. However, even here there is a small delay due to mechanical inertia, electronics, and speed limits of the electric power steering motor.
  Technical literature and experimental data show that electronic steering systems have time constants between 0.05 and 0.2 seconds. Therefore, 0.1 s is a standard value used in many simulators and advanced control models.

In the Simulink system using CARLA's vehicle, these LPF Actuators are not used. The main reason is that the inclusion of LPF filters for the actuators results in a significant increase in the time required to perform and complete the simulation. In particular, LPF filters require finer temporal resolution to be simulated correctly, thus increasing the number of computational steps and, consequently, the total simulation time.

"Vehicle Body 3DOF Dual Track" Block



(a) Model Block


Block Parameters: Vehicle Body 3DOF Dual Track

Vehicle Body 3DOF Lateral (mask) (link)

Implements a 3 DOF rigid two-axle vehicle body model to calculate longitudinal, lateral, and yaw motion. Accounts for body mass, aerodynamic drag, and weight distribution between the axles due to acceleration and steering. Configure as a single track (bicycle) or dual track vehicle with options for the axle force calculation. Additional input options include front and rear wheel steering, external friction, and wind loads.

Block Options:

Vehicle track: Dual

Axle forces: External longitudinal forces

(b) Dual Track Model block options

Figure 8.6: "Vehicle Body 3DOF Dual Track" Block

The block used can be obtained through the *Automated Driving Toolbox*. There are several options to select:

- **Vehicle track –> Dual**, setting Dual Track as the model
- **Axle forces –> External longitudinal forces**. With the assumption that the vehicle is front-wheel drive, the external forces acting on the rear axle *FwR* are set equal to 0, while those on the front axle *FwF* are set as m*ax/2 (equal for both the front wheels), via the Gain block. In this way, the second principle of dynamics is met, i.e.:

$$ma_x = \sum_{i=1}^{n} F_{x_i}$$

  where i is the number of wheels, and F_{x_i} is the longitudinal force acting on the generic wheel i.

- **Input signals –> Front wheel steering**. Thus I impose that only the front wheels are steered. The steering angle is the one calculated by the NMPC, the same for both front wheels.
- **Vehicle Parameters**: various parameters are required, such as track, wheelbase, number of both driving and non-driving wheels, moment of inertia around vehicle z axis, and so on. These parameters are the same as the CARLA vehicle. Some of them that are unknown (e.g. Longitudinal velocity tolerance, Relative wind angle vector, etc.) are set as default.

The outputs of this vehicle block are: *xdot* and *ydot* (longitudinal and lateral speeds calculated in the vehicle CoG, respectively), *psi* (yaw angle), *r* (yaw rate), *FzF* and *FzR* (vertical loads at the front and rear axles, respectively).

Matlab Functions

Listing 8.3: X_dot and Y_dot Calculus

```
function [X_dot,Y_dot] = fcn(vx,vy,psi)
X_dot = vx.*cos(psi) - vy.*sin(psi);
Y_dot = vx.*sin(psi) + vy.*cos(psi);
end
```

This function is used to convert vehicle speeds from the local reference system (i.e., fixed on the vehicle itself) to the global reference system (i.e., the map or road plane). These expressions are the equations (3.18) and (3.19). These are the *inputs* of this function:

- **vx**: longitudinal velocity of the vehicle (i.e., along the front-rear axis of the vehicle).
- **vy**: lateral velocity of the vehicle (i.e., along the transverse axis of the vehicle).
- **psi**: yaw angle, i.e., the orientation of the vehicle with respect to the overall X axis.

The function calculates *X_dot* (the vehicle speed along the global X-axis) and *Y_dot* (the velocity of the vehicle along the global Y axis).
This is critical because as the vehicle moves and rotates, its local velocities must be "projected" onto the global system to understand how it actually moves on the map. Then, two *Integrator* blocks are used to obtain the X and Y coordinates of the vehicle in the global reference frame. In these blocks, initial coordinates *X0* and *Y0* are put.
In practice, this function allows the global (X, Y) position of the vehicle to be updated from its local velocities and orientation.

Listing 8.4: vx_dot Calculus

```
function vx_dot = fcn(ax,ydot,r)
vx_dot = ydot.*r + ax;
end
```

This function calculates the real longitudinal acceleration of the vehicle, that is, how the vehicle's forward velocity changes over time. The inputs of the function are:

- **ax**: longitudinal acceleration applied to the vehicle (e.g., by accelerator or brake).
- **ydot**: lateral velocity of the vehicle.
- **r**: yaw angular velocity (how fast the yaw angle changes over time).

The formula used takes into account both the applied acceleration and an additional term due to the combination of lateral velocity and vehicle rotation.
This second term $ydot \cdot r$ represents an internal "Coriolis" effect within the vehicle system: as the vehicle rotates and moves laterally, there is a contribution that affects the change in longitudinal velocity.

8.3.3 "reference generator" Subsystem

This subsystem consists of 2 blocks: a block that generates the trajectory that the ego vehicle should follow, and a second block that calculates the cross track error (same block described in section 5.3.3).

Listing 8.5: ref_gen Function - 1

```
function [ref, ref_pose, v] = ref_gen(tr1, tr2, tr3, ref_tr, ze, vx_ref)

% Parametri
NS = 50; % numero punti di riferimento fissi
Kc = 10.2;

N = size(ref_tr,1);
% --- Trova punto traiettoria più vicino ---
dis = vecnorm(ref_tr(:,1:2)' - ze(1:2)*ones(1,N), 2, 1);
[~, c] = min(dis);
ref_pose = ref_tr(c,:)'; % 2x1
```

The *ref_gen* function that takes as input the positions of external vehicles (*tr1, tr2, tr3*), the reference trajectory *ref_tr*, the current state of vehicle ze (specifically its position), and a nominal reference longitudinal velocity *vx_ref*. The function returns:

- *ref*: the XY reference points to follow.
- *ref_pose*: the position of the current reference point.
- *v*: the adjusted reference velocity

Two important parameters are set: *NS*, indicating how many fixed reference points (50) you want to consider for the future trajectory, and *Kc* a coefficient used to adjust the speed according to the curvature of the road (value chosen according to the type of scenario). This last parameter is obtained by tuning, and it varies according to the vehicles' velocities, relative distances and the change in curvature between the immission and the main roads. Next, the distance between the vehicle's current position (ze(1:2)) and all the points on the reference trajectory *ref_tr* are calculated. The nearest point (*c*) is then found and its position is assigned to *ref_pose*. This point will be the current reference for driving.

Listing 8.6: ref_gen Function - 2

```
% --- Calcolo curvatura ---
dX = diff(ref_tr(:,1));
dY = diff(ref_tr(:,2));
ds = sqrt(dX.^2 + dY.^2);
psir = atan2(dY, dX);
dpsir = diff(psir);
curv = dpsir ./ ds(1:end-1);

% --- Calcolo velocità adattiva ---
```

```
v_adapt = vx_ref / (abs(curv(c)) * Kc + 1);

if c < ct
    v = v_adapt;
else
    v = vx_ref;
end

% —— Calcolo Np adattivo ——
Np = round(v * 3);
```

Next there is the calculation of road curvature and adaptive speed.
The reference speed used before the input is calculated using the following formula:

$$v = \frac{vx\_ref}{Kc * |curv| + 1}$$

This means that on sharper curves, where the curvature is higher, the reference speed decreases to ensure vehicle stability and safety. Conversely, when the road is perfectly straight and the curvature is zero, the reference speed equals the original desired speed. It is important to note that the value of K_C varies depending on several factors related to the driving scenario. These factors include the number of vehicles present, their speeds, and their relative positions on the road. Adjusting K_C based on these conditions allows the system to adapt the reference speed dynamically, maintaining safe and efficient vehicle behavior under varying traffic and road conditions.
The decision to adapt vehicle speed to the curvature of the road stems from two main reasons, both closely related to both methodological consistency with what has been developed in the integration between Simulink and CARLA and to the characteristics of the complex scenarios faced. On the one hand, the approach of making speed a function of curvature has also been adopted in the integrated Simulink-CARLA system, where vehicle dynamics are simulated in realistic environments and speed is continuously adapted according to the radius of curvature of the trajectory. This ensures uniformity and consistency in the design and evaluation criteria of the different simulation environments. On the other hand, in all the complex scenarios analyzed, such as entering a traffic circle or a main road, a significant change in curvature is always observed right at the merging point. In these situations, the geometry of the road imposes a change in the radius of curvature that, if not properly managed from a speed perspective, can lead to loss of grip, increased cross-track error and reduced safety. Adapting speed to curvature therefore allows the vehicle to maintain control even in the most critical phases of the maneuver, improving stability, reducing oscillations, and ensuring compliance with the dynamic constraints imposed by the vehicle and the road itself.
ct is an index that indicates a particular reference point along the trajectory (e.g., the entry point into a traffic circle or road).
The differences between consecutive points in the trajectory are calculated to obtain the distance between them (ds) and the angle of the trajectory ($psir$). The curvature $curv$ is calculated as the change in angle per unit distance, that is, how much the trajectory changes direction locally, using the following formula:

$$curv_i = \frac{d\psi_i}{ds_i}$$

where $d\psi_i$ is the derivative (so the variation) of the curvature between 2 generic points i and $i+1$, and ds_i is the distance between the same 2 consecutive points.

The reference velocity is adjusted inversely to the curvature: if the curve is tight (high curvature), the velocity decreases.
If the current point *c* is before the input point *ct*, the adapted velocity is used; otherwise the nominal velocity vx_ref is kept.
Finally, future reference points *Np* are calculated, depending on the current velocity. The 3 by which the velocity is multiplied for calculation is the *Prediction Horizon*.
Further modifications of this function will be described specifically in the next chapter, describing the results of simulations of the complex scenarios.

8.3.4 NMPC Simulation Parameters

In the present work, the implemented nonlinear predictive control (NMPC) system is characterized by a simpler structure than that used in the CARLA simulator. In particular, the adopted cost function is defined as:

$$J(u(t:t+T_p)) \doteq \int_t^{t+T_p} \left(\|\tilde{y}_p(\tau)\|_Q^2 + \|u(\tau)\|_R^2 \right) d\tau + \|\tilde{y}_p(t+T_p)\|_P^2 \tag{8.1}$$

where the term related to the variation of inputs, represented by the matrix R_{sd} is not included, nor is there a term related to the collocation nodes ncp typical of more complex formulations.
The following is the table containing the numerical values of the NMPC parameters:

NMPC Parameters	**Values**
T_s	0.05
T_p	3
n	6
R	$\begin{bmatrix} 0.06 & 0 \\ 0 & 1 \end{bmatrix}$
Q	$\begin{bmatrix} 25 & 0 \\ 0 & 25 \end{bmatrix}$
P	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
ub	[5, 0.4]
lb	[-8, -0.4]

Table 8.1: NMPC Parameters' Values

This simplification means that the values of the weight matrices Q and R are also different from those used in the CARLA system. This choice, although it reduces computational and modeling complexity, is advantageous in that the simplified model and the smaller number of parameters still yield good results in terms of path tracking, robustness, and fit. Moreover, this configuration leaves ample room for future control improvements and optimizations.
Another distinctive aspect concerns the limits imposed on the steering angle: in the current system the limit is set at, in absolute values, 0.4 rad (about 23°), lower than CARLA's limit of 50°. This reduction to less than half of the maximum steering limit was adopted as the optimal compromise between the three scenarios considered (lane merging, traffic roundabout, urban environment with stop). Functionally, a more restrictive limit on steering angle can promote greater vehicle stability and control in complex scenarios

such as lane merging, traffic roundabouts, and urban driving with stops. Limiting the maximum steering angle reduces the possibility of abrupt and sudden maneuvers, facilitating smoother and more predictable vehicle behavior. This helps improve stopping power and vehicle dynamics management, especially in heavy traffic situations or tight spaces, where excessive steering could compromise safety and stability.

8.3.5 "nlcon" NMPC Function

The function **nlcon(x,y)** is designed to calculate safety constraints (constraints) between the controlled vehicle and three other vehicles in the scenario. These constraints are used to ensure that the vehicle maintains a safe distance from other vehicles during simulation or trajectory planning.

Listing 8.7: nlcon Function

```matlab
function F=nlcon(x,y)

    %Load Trajectories
    traj_data = load('traj.mat');     % More efficient loading

    % Extract trajectories
    Xr = traj_data.Xr; Yr = traj_data.Yr;
    psir=[atan2(diff(Yr),diff(Xr));0];
    Xr1 = traj_data.Xr1; Yr1 = traj_data.Yr1;
    Xr2 = traj_data.Xr2; Yr2 = traj_data.Yr2;
    Xr3 = traj_data.Xr3; Yr3 = traj_data.Yr3;

    %Time Synchronization
    min_length = min([size(x(1,:)), length(Xr), length(Xr1), length(Xr2),
   length(Xr3)]);
    x = x(:,1:min_length);

    %Safety Parameters
    v3 = ...;       % Other vehicles' speed
    v2 = ...;
    v1 = ...;
    vx = x(4,1:min_length);
    dist_sec = ceil(vx);
    k = 3;
    gap1 = dist_sec + k*abs(vx-v1);
    gap2 = dist_sec + k*abs(vx-v2);
    gap3 = dist_sec + k*abs(vx-v3);

    %Calculate Constraints
    F = zeros(3, min_length); % Initialize constraint matrix

    % Vehicle 1 constraint
    F(1,:) = gap1 - vecnorm(x(1:2,:) - [Xr1(1:min_length)'; Yr1(1:
   min_length)']);

    % Vehicle 2 constraint
    F(2,:) = gap2 - vecnorm(x(1:2,:) - [Xr2(1:min_length)'; Yr2(1:
   min_length)']);

    % Vehicle 3 constraint
    F(3,:) = gap3 - vecnorm(x(1:2,:) - [Xr3(1:min_length)'; Yr3(1:
    min_length)']);
```

First there is the loading of vehicle trajectories. Here the function loads a file called 'traj.mat' that contains the trajectories (X and Y positions) of three outer vehicles (Xr1,Yr1, Xr2,Yr2, Xr3,Yr3) and the main vehicle (Xr, Yr). It also calculates the yaw angle psir of the lead vehicle, using the difference of the coordinates. To make sure that all trajectories and state of the controlled vehicle have the same time length (same number of samples), the minimum length among all vectors is taken.
The x-matrix (whose rows correspond to the 6 states and whose columns represent the values of the states instant by instant) is reduced to this minimum length (the number of columns) to avoid dimension errors.
Next, the velocities of the other three vehicles (*v1, v2, v3*) are defined.
vx is the current velocity of the controlled vehicle, extracted from the state vector *x*.
dist_sec is the base safety distance, calculated as the speed of the controlled vehicle rounded up (considering the reaction time equal to 1).
k is a coefficient that increases the safety distance as a function of the speed difference between the controlled vehicle and the other vehicles. It was obtained by tuning.
The variables *gap1, gap2* and *gap3* then represent the minimum required safety distance for each of the three outside vehicles, taking into account both the current speed of the controlled vehicle and the speed difference with the others.
An F-matrix is created that will contain the values of the constraints for each of the three vehicles, for each instant of time. For each external vehicle, the Euclidean distance between the current position of the controlled vehicle (*x(1:2,:)* are the X and Y coordinates) and the position of the external vehicle (*Xr1, Yr1*, etc.) is calculated. The constraint is defined as the difference between the required safety distance (gap) and the calculated actual distance.

- **If F(i,j) $>$ 0**, the constraint is met (sufficient distance).
- **If F(i,j) $<$ 0**, the constraint is violated (too close to another vehicle).

Chapter 9

Complex Scenarios - Simulation Results

9.1 Extraurban Lane Merging

In the suburban scenario, the input occurs on a high-speed main road. The chosen road is a part of the East Ring Road of Milan. In this case, the merging point is defined as the entry point (parameter $ct = 44$).
The trajectories were calculated, as described in the previous chapter in section 8.2, that is, with the *Driving Scenario Designer* app.

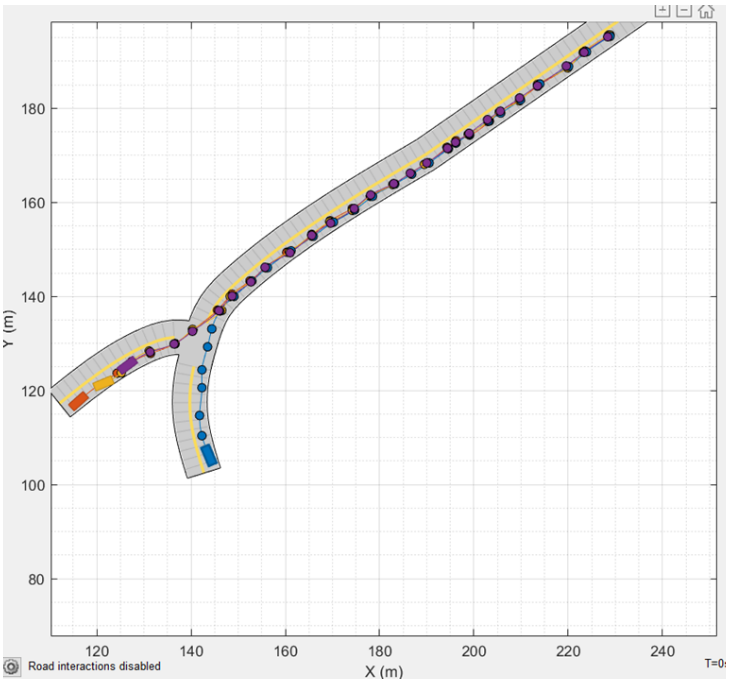


Figure 9.1: Tangenziale Est - Milan

The part of the imported road was obtained from *OpenStreetMap*.
There are 4 vehicles in the scenario:

- the **ego-vehicle**, controlled by the NMPC (blue vehicle in the figure). It has a variable reference speed: 65 km/h before entering the bypass, and 80 km/h after entering the bypass.

- **Other 3 vehicles** to simulate traffic, lined up according to their speed: the vehicle in front has a speed of 100 km/h, the one in the middle goes at 90 km/h, and the last one in the back goes at 70 km/h.

The scenario was developed with the Worst Case Approach: the ego-vehicle should collide with one of the other vehicles traveling on the road. Therefore, thanks to the *nlcon* function and speed adaptation to the curvature, the vehicle slows down, and gives way to other vehicles before merging.
In this case, the interpolation method used is “makima,” to have a more predictable behavior as the parameter Kc changes (see chapter 8.2).

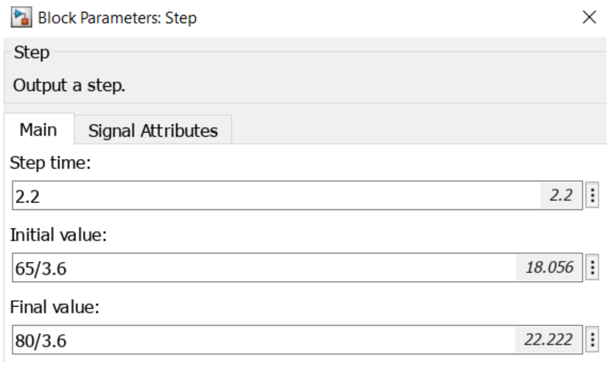
9.1.1 "acc" Function

Listing 9.1: "acc" block

```
function ax = acc(ax_NMPC, ze, ref_tr)
    % ze(1:2) posizione attuale (x,y)
    pos_attuale = ze(1:2)'; % 1x2

    % Prendi tutte le posizioni della traiettoria Nx2
    pos_traiettoria = ref_tr(44,1:2);

    % Calcola distanze euclidee tra posizione attuale e tutti i punti della
     traiettoria
    dist = vecnorm(pos_traiettoria - pos_attuale, 2, 2); % Nx1

    % Trova indice del punto più vicino
    [~, k] = min(dist);

    if ze(2) <= pos_traiettoria(k,2)
        % Prima di superare il punto target, usa accelerazione NMPC
        ax = ax_NMPC;
    else
        % Dopo aver superato il punto target, calcola frenata per
    raggiungere v_des
        ax = 5;
    end
end
```

This MATLAB code block is used to handle the longitudinal acceleration of an autonomous vehicle during an input maneuver on a reference trajectory, taking advantage of a Nonlinear Model Predictive Control (NMPC) controller.
The first step is to extract the current position of the vehicle and all positions of the reference trajectory. Next, the code calculates the Euclidean distance between the current position and all points in the trajectory, thus identifying the point in the trajectory closest to the vehicle at that time. At this point, a check is made: if the vehicle has not yet passed the entry point of the suburban road (i.e., the current y-coordinate is less than or equal to that of the target point), then the acceleration calculated by the NMPC is used. On the other hand, if the vehicle has passed the target point, a maximum acceleration (ax = 5) is forced. This is to make the vehicle accelerate as quickly as possible, ignoring the command from the NMPC. This logic is adopted to solve a problem in this entry maneuvers: after completing the entry, the vehicle tends to slow down (to reduce deviation

from the target trajectory), risking being caught up and rear-ended by the following vehicle. This is caused by the fact that the Q matrix (about reduction of error) has higher values with respect to the matrix R (about commands' intensity), giving higher weight to the reduction of cross-track error, at the expense of reduction of *ax*.
So, this function forces maximum acceleration after the target point ensures that the vehicle accelerates quickly, reducing the risk of rear-end collision.
This function takes as inputs the reference trajectory of the vehicle ref_tr, the acceleration commanded by the NMPC ax_NMPC and the states of the vehicle *ze*. This new acceleration is imposed to the "dispacher" function, and converted into throttle and brake.
An important aspect of this logic is the management of the reference speed (vx_ref), which is not constant but follows a stepped profile: up to a certain instant *Ti*, the reference speed is set at 65 km/h, and after *Ti*, the reference speed is increased to 80 km/h.


Block Parameters: Step

Step

Output a step.

Main | Signal Attributes

Step time:
2.2 (2.2)

Initial value:
65/3.6 (18.056)

Final value:
80/3.6 (22.222)

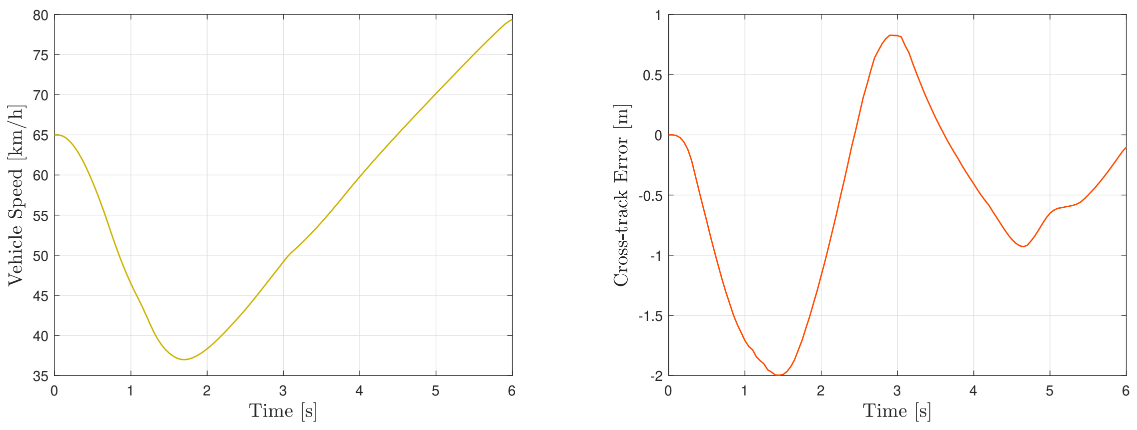
Figure 9.2: Reference speed - Extraurban Merging

Ti is calculated as follows, obtaining 2.2 s:

$$T_i = T_{fin} \cdot \frac{ct}{\text{min\_length}}$$

where *Tfin* is the total simulation time, *ct* is the index of the entry point (target point) on the trajectory and *min_length* is the total length of the reference trajectory (obtained by time synchronization with other trajectories).
This means that after completing the input (i.e., after the target point), the vehicle must accelerate to reach the new higher reference speed. The code block, therefore, forces the vehicle to apply maximum acceleration at this very stage, so as to quickly reach the desired speed of 80 km/h.
It is important to note that the acceleration provided by the NMPC does not necessarily represent the actual acceleration that the vehicle can achieve.

9.1.2 Simulation Results - Extraurban Merging



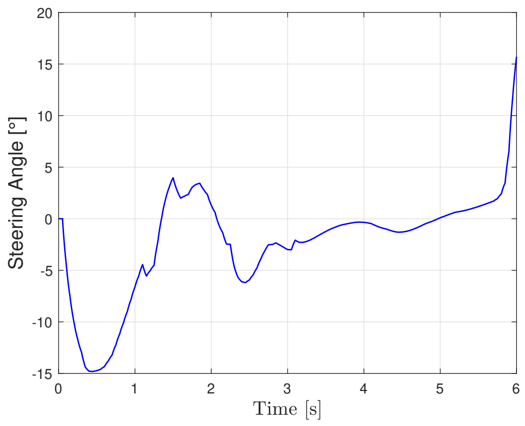
(a) Vehicle Speed **(b)** Cross-track Error

Figure 9.3: Speed and Error Plots - Extraurban Merging

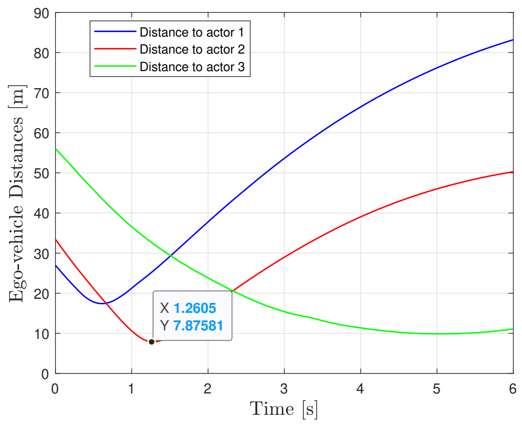
Plot a): Vehicle Speed
At first, the speed is the reference speed of 65 km/h. In the first two seconds, however, the vehicle speed decreases significantly, dropping to about 37 km/h due to the *nlcon* function and the increase in curvature forcing the ego-vehicle to slow down, sensing one or more vehicles in the vicinity, in correspondance of the entry in the main road.
After this abrupt deceleration, the vehicle speed undergoes a sharp increase, rising from 65 to 80 km/h. This change is due to two reasons: first, the reference speed increases, and the vehicle accelerates to reach it; second, there is the effect of the "acc" function: the NMPC imposes maximum acceleration, allowing the vehicle to reach the new reference speed in a few seconds. This delay is physiological, due to the dynamic limits of the vehicle in terms of maximum longitudinal acceleration and the need to ensure a safe and comfortable maneuver.

Plot b): Cross-track Error
At the beginning, the vehicle is practically on the reference trajectory (error close to zero). In the first few seconds, however, the error quickly becomes negative, reaching a minimum value around -2 m: this means that the vehicle moves laterally to the left from the ideal lane, to start entering the main road. Such a high error is due to the high value of the curvature at the entry.
After this minimum, the curve rises very rapidly, passing zero and reaching a positive peak near +0.8 m around 3 seconds (shifted laterally to the right). This is because the vehicle has entered the tangential, where the road is almost straight.
After the peak, the error decreases again, oscillating but with decreasing amplitude, a sign that the vehicle is stabilizing on the new trajectory. These oscillations are normal and represent the small adjustments needed to realign perfectly to the new lane.

RMS e_{ct} [m]	**Max e_{ct} [m]**
0.9552	1.9974



(a) Steering Angle plot



(b) Distances plot

Figure 9.4: Steering Angle and Distances Plots - Extraurban Merging

Plot a): Steering Angle

At the beginning, there is a sharp steering to the left (negative angle, down to -15°), which is necessary to initiate the lane change maneuver. Immediately thereafter, the steering angle rapidly returns toward zero and then reaches +5°. This second peak is obtained just before 2 s, at the minimum value of speed assumed in this maneuver. Then, shortly after entering, a new local minimum of -5° is obtained, probably due to the abrupt change in curvature between the entry ramp and the main road. These oscillations indicate the continuous adjustments that the driver (or the control system) must make to maintain the desired trajectory during merging.
The vehicle then maintains 0° of steering, due to the fact that the main road is almost straight.
Toward the end of the maneuver, the steering angle increases again, this time in a positive direction, until it reaches about +15°; probably, due to excessive speed the vehicle should deviate from the reference trajectory, and so the controller applies some steering angle to reduce the cross-track error.

Plot b): Distances to other vehicles

The graph shows the trend of distances between the ego-vehicle and three different actors (actor 1, actor 2, actor 3) as a function of time, expressed in seconds. Distances are represented in meters on the y-axis.

1. The blue curve represents the distance between the ego-vehicle and actor 1, at speed v1 = 100 km/h. Initially this distance decreases, indicating that the ego-vehicle is reaching the immission point, and then it increases steadily.

2. The red curve shows the distance to actor 2, traveling at speed v2 = 90 km/h. This distance also initially decreases, reaching a minimum shortly after 1 s of about 7.88 m; the ego-vehicle, due to the combination of the increase in road curvature and the function of constraints **nlcon**, the vehicle slows down and gives way to actor 2.

3. The green curve indicates the distance to actor 3, at speed v3 = 70 km/h. In this case, the distance starts at a high value, decreases rapidly to about 10 m around 2 s, and then stabilizes at low values (about 10 m) for the rest of the time interval. This is because the vehicle, after slowing down to 37 km/h, restarts by accelerating to

full speed, but its speed is less than that of vehicle 3, thus decreasing their relative distance. Once v3 speed is reached and exceeded, their relative distance increases slightly.

The plot is useful for analyzing the distance dynamics between an autonomous vehicle and other subjects in the surrounding environment, identifying possible situations of risk or close interaction as a function of time. In particular, the minimum highlighted on the red curve represents a critical point for possible safety maneuvers.

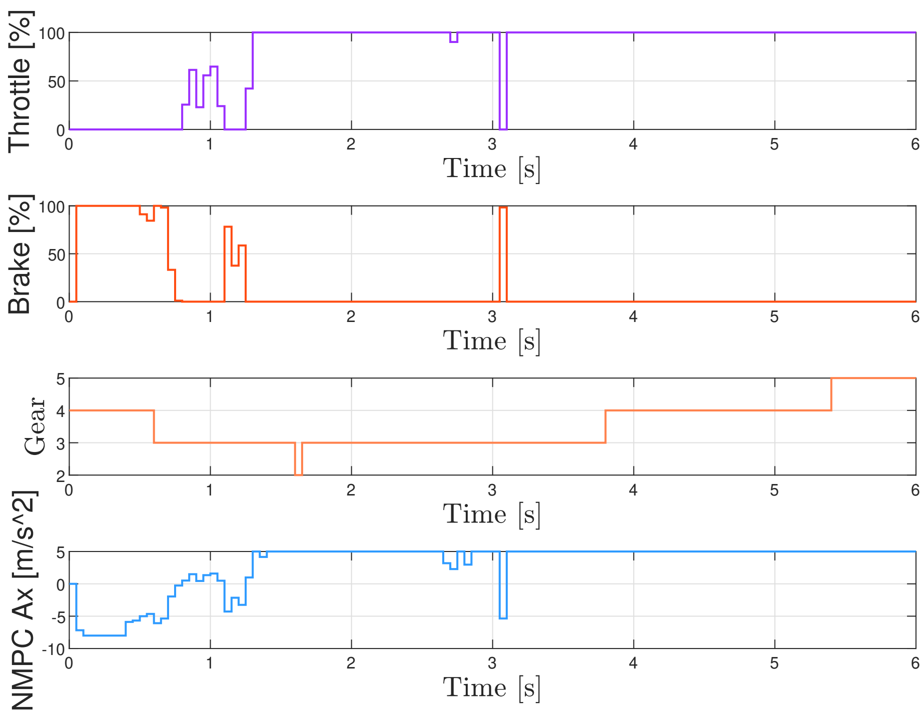


Figure 9.5: Throttle, Brake, Gear, NMPC Acceleration - Extraurban Merging

These 4 plots represent 4 important control variables: throttle, brake, gear, and longitudinal acceleration imposed by the NMPC respectively.
It can be seen that the last plot (longitudinal acceleration) is a combination of the first 2 plots (throttle and brake), and that when one of them is different from 0, the other is equal to 0. At the beginning of the simulation, it can be seen that the brake is almost always at 100 %, as the ego-vehicle, sensing other vehicles on the main road that are passing near the entry point, decelerates sharply. After 1 s, the vehicle re-accelerates bringing itself close to the entry, and slows down again as the vehicle passes at 90 m/h.
The vehicle starts in gear 4, downshifting to gear 3 in the time interval described so far.
Next, once vehicle 2 has been given the right of way, the vehicle begins to accelerate to reach the new reference speed of 80 m/h. Although the vehicle is accelerating at full-throttle, for a very brief moment of time, the vehicle goes into gear 2, due to the actuator-induced delay. At that point, the total speed of the vehicle is 37 km/h, while the longitudinal speed is 35 km/h (below the longitudinal speed threshold of gear 3 equals 36 km/h).
Shortly after the entry point, the vehicle decelerates again for a very short instant of time, because the cross-track error has increased (+0.8 m), and by decelerating, the vehicle aligns more closely with the trajectory.

After that, the vehicle maintains maximum constant acceleration until the end of the simulation; this is due to the "*acc*" function (see chapter 8.1.1). As for the gear, the vehicle also reaches gear 5 toward the end of the simulation.

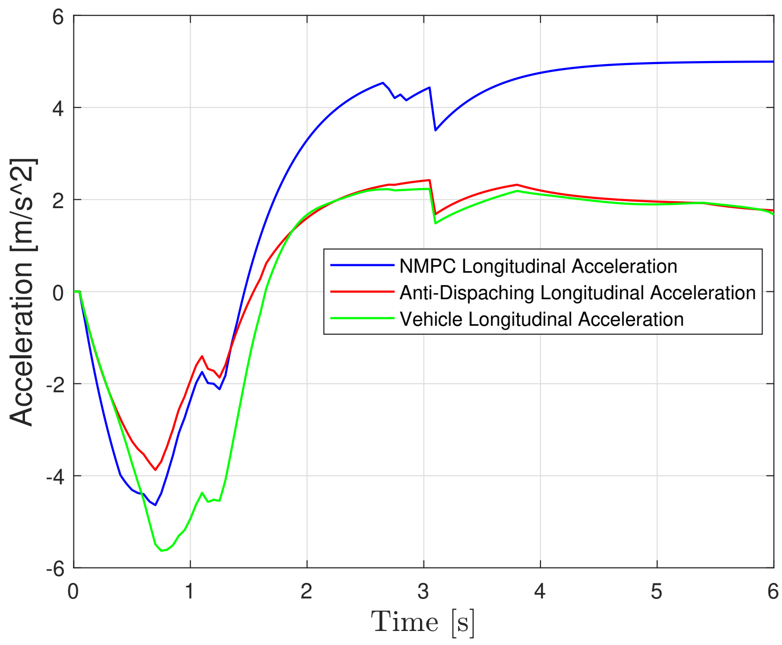


Figure 9.6: Accelerations - Extraurban Merging

In this plot, three longitudinal accelerations are compared: the one imposed on the vehicle by the NMPC (blue curve), the one calculated by the "anti-dispatcher" block (red curve), and the actual acceleration of the vehicle (green curve).
In the first part of the simulation, there is a deceleration for all three curves: the red one shows less negative values in absolute terms compared to the blue curve, due to the saturation of values caused by the reconversion of throttle and brake into acceleration, according to the vehicle's limits. The green curve shows the greatest deceleration, due to the additional term that considers the vehicle's yaw rate and lateral velocity, causing the vehicle to slow down more than what was predicted by the NMPC. As explained regarding the control variable plot, there is an acceleration and a new deceleration at about 1 s.
Afterwards, the vehicle accelerates at maximum, as previously explained, and the curves follow different trends: the blue curve has an almost constant acceleration of 5 m/s^2 (maximum imposed by the NMPC), while the other two follow the physical limits of the vehicle (5 m/s^2 is the maximum longitudinal acceleration in gear 1, and this decreases as the gear increases).

9.2 Urban Lane Merging

The urban scenario analyzed is based on a real situation: the entry from a suburban interchange (SS7) onto an urban road (Via Dante Alighieri, Matera). In this context, the ego-vehicle must compulsorily stop at a stop sign before it can enter the main flow. The traffic consists of four vehicles, with speeds of 36, 41.4, 46.8 km/h. The ego-vehicle has a reference speed of 50 km/h, higher than the other ones because, in the simulation, the ego-vehicle enters from a suburban interchange onto an urban road, and its initial speed reflects that typical of a suburban stretch, while other vehicles already on the urban road maintain lower speeds, in line with the limits imposed by the city context.

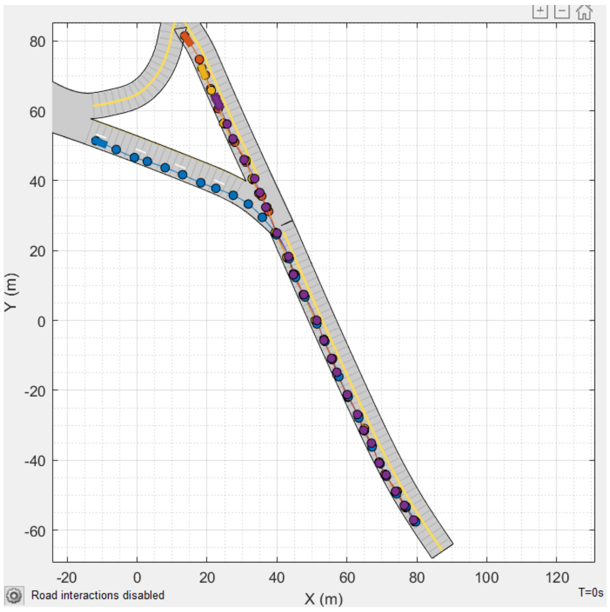


Figure 9.7: Urban Entry

Trajectories were generated using the "Driving Scenario Designer" app and considering the worst-case scenario, i.e., the most critical situation in terms of safety. Trajectories were calculated by linear interpolation of waypoints, with time synchronization between vehicles to ensure consistency of simulations. In this scenario, linear interpolation ensures that the vehicle stops at the stop for smaller values of Kc than would be obtained with “makima” interpolation.
Road length, travel time and interpolation points were calculated as a function of reference speed and sampling time, while synchronization of the trajectories ensures that all vehicles share the same time base for analyzing the results.

9.2.1 Trajectory Planning

Listing 9.2: "ref_gen" block

```
% —— Calcolo curvatura ——
dX = diff(ref_tr(:,1));
dY = diff(ref_tr(:,2));
ds = sqrt(dX.^2 + dY.^2);
psir = atan2(dY, dX);
dpsir = diff(psir);
curv = dpsir ./ ds(1:end-1);

Kc = 10.2;
v_adapt = vx_ref / (abs(curv(c)) * Kc + 1);

%% —— Logica di stop —— (scenario urbano)
index_stop = 60; % indice punto stop nella traiettoria
stop_point = ref_tr(index_stop, 1:2)'; % 2x1

```

```
dis_stop = vecnorm(ref_tr(index_stop,1:2)' - ze(1:2)*ones(1,N), 2, 1);
[~, c_stop] = min(dis_stop);

pos_veicolo = ze(1:2,c_stop); % 2x1 posizione attuale veicolo

% Posizioni veicoli esterni al punto stop
pos1 = tr1(c_stop, 1:2)';
pos2 = tr2(c_stop, 1:2)';
pos3 = tr3(c_stop, 1:2)'; % Distanze euclidee
dist1 = norm(pos1 - pos_veicolo);
dist2 = norm(pos2 - pos_veicolo);
dist3 = norm(pos3 - pos_veicolo);

% Soglia sicurezza (esempio funzione di velocità)
soglia = ceil(v_adapt);

% Condizione per fermarsi: veicolo troppo vicino e davanti
ferma = (dist1 < soglia && pos_veicolo(1) >= pos1(1)) || ...
(dist2 < soglia && pos_veicolo(1) >= pos2(1)) || ...
(dist3 < soglia && pos_veicolo(1) >= pos3(1)) || ...
(norm(stop_point - pos_veicolo) < 4); % ferma anche vicino allo stop

if ferma
v = 0; %per avere una velocità più bassa possibile
else
v = v_adapt;
end

% —— Calcolo Np adattivo ——
Np = round(v * 3);
```

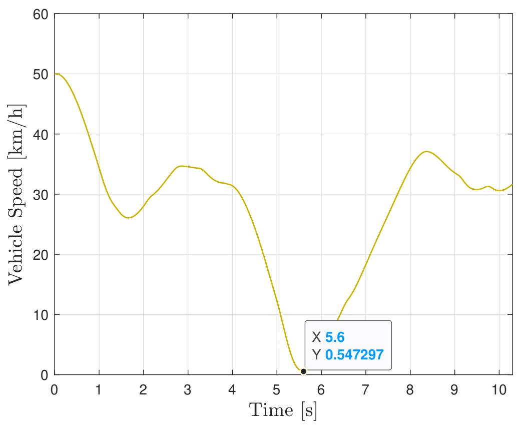
The first part of the code, which deals with the calculation of curvature and adaptive velocity, is the same as for the other complex merging scenarios. What is new in the trajectory planning of this scenario is the logic to handle the stopping of the vehicle at the STOP signal, before entering the city street.
The system identifies a specific point on the trajectory where it is necessary to stop; this point is selected via an index on the reference trajectory (60) and its coordinates are extracted.
The current position of the vehicle is determined by finding the point on the trajectory closest to the stop point, using the Euclidean distance between the vehicle and the stop point.
The system also tracks the position of other vehicles that are traveling along the city street. For each of these vehicles, the distance to the lead vehicle is calculated.
A dynamic safety threshold is established, which depends on the adaptive speed of the vehicle. This threshold represents the minimum safe distance the vehicle must maintain from other vehicles to avoid collisions.
The vehicle must stop if at least one of the following conditions occurs:

- Another vehicle is in front (i.e., along the direction of travel) and at a distance less than the safety threshold.
- The vehicle is very close to the stop point (in this case at a distance of less than 4 m). These conditions are evaluated simultaneously: if at least one is true, the vehicle sets the speed v = 0. It should be clarified that Setting v = 0 in an NMPC is not the same as stopping exactly where we want to, because the system considers the

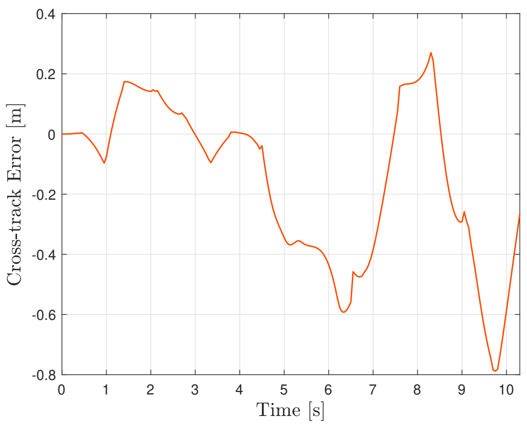
actual dynamics of the vehicle and the NMPC works on finite time windows and favors continuity and regularity of trajectory rather than an exact stop.

If any of the stop conditions are met, the reference speed v is set to zero to ensure maximum safety. If no stop condition is met, the vehicle continues to move at the previously calculated adaptive speed.

9.2.2 Simulation Results - Urban Merging



(a) Vehicle Speed



(b) Cross-track Error

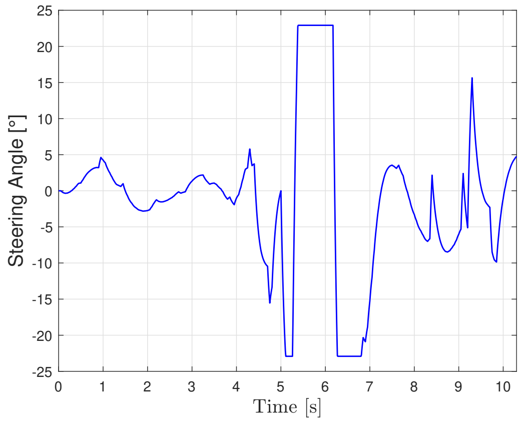
Figure 9.8: Speed and Error Plots - Urban Merging

Plot a): Vehicle Speed
The vehicle starts at 50 km/h (reference speed), then slows down for the first time due to the constraints function *nlcon* (as it detects vehicles approaching the merging point from a certain distance). Then it accelerates again once the first two vehicles have passed, and, as the third vehicle approaches (traveling at 36 km/h), the ego-vehicle slows down once more. At the stop sign, there is an increase in road curvature, so the vehicle's deceleration is emphasized due to the speed adapting to the curvature. The vehicle does not come to a complete stop, but reaches a speed of 0.55 km/h (0.15 m/s), which is low enough to be considered stopped (in a real-world scenario, below a certain speed—and therefore below a certain engine RPM, the ICE does not deliver torque, preventing the vehicle from moving).
Once the third vehicle has also passed, the ego-vehicle starts moving again and completes the merging maneuver. After reaching the minimum speed, it accelerates again, indicating that the vehicle has exited the stopping phase and is accelerating to merge into urban traffic. After restarting, following a speed peak above 36 km/h, the vehicle decelerates and maintains a speed around 30–35 km/h, thus keeping a certain distance from the vehicle ahead.

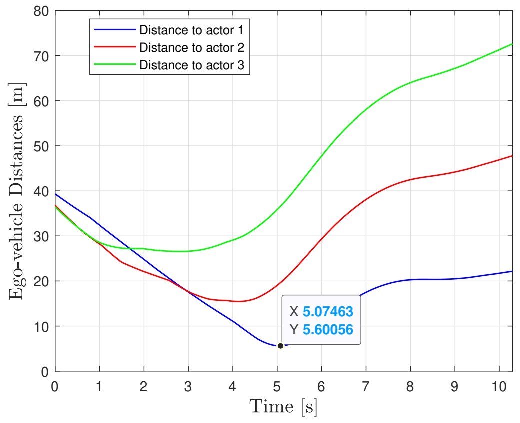
Plot b): Cross-track Error
During the initial deceleration, the vehicle deviates slightly to the left from the reference trajectory (-0.1 m). When the speed increases, the cross-track error becomes positive, indicating a deviation to the right. As the vehicle slows down again, its cross-track error decreases and becomes negative once more. When the vehicle comes to a stop, the error

is about -0.4 m.
Then, as it accelerates to start moving and merge, the error continues to increase in absolute value, reaching a first trough of -0.6 m as it successfully completes the merging maneuver.
When the vehicle reaches its local maximum speed, there is a new peak in cross-track error of +0.2 m (the error passes from negative to positive values since the vehicle is trying to align itself with the reference trajectory). In the final stretch, where the vehicle maintains a speed of around 30–35 km/h, the maximum error value of 0.8 m is reached.

RMS e_{ct} [m]	**Max e_{ct} [m]**
0.3031	0.7880



(a) Steering Angle plot



(b) Distances plot

Figure 9.9: Steering Angle and Distances Plots - Urban Merging

Plot a): Steering Angle
In the first 5 seconds, during which the vehicle decelerates and accelerates, the steering angle oscillates between -5 and 5°. When the vehicle decelerates again and then comes to a stop and restarts to enter the main road, there are sharp steering inputs, reaching the steering saturation value of ±23° between 5 and 7 seconds. This value likely would have been exceeded if it weren't for the steering limits imposed by the NMPC.
After merging, the steering angle stabilizes, oscillating around zero, except for a peak of about 15° when the vehicle accelerates again and reaches the velocity local maximum above 36 km/h.
The oscillations present in this plot are typical of the adjustments needed to maintain the desired trajectory and to stabilize the vehicle, in particular during the main maneuver.

Plot b): Distances to the other vehicles
This graph shows the evolution of the distances (in meters) between the analyzed vehicle and three other vehicles (referred to as actor 1, 2, and 3).
At the beginning, the distances decrease, consistent with approaching the STOP sign and the other vehicles already present. The distances to actors 2 and 3 are not critical; the most critical distance is reached around 5 seconds (about 5.6 meters) with actor 1 (v1 = 36 km/h), when the ego-vehicle stops and the other vehicle is passing by. After yielding

to the other vehicles, the distances increase, as the vehicle begins to merge at a very low speed after stopping at the stop sign.

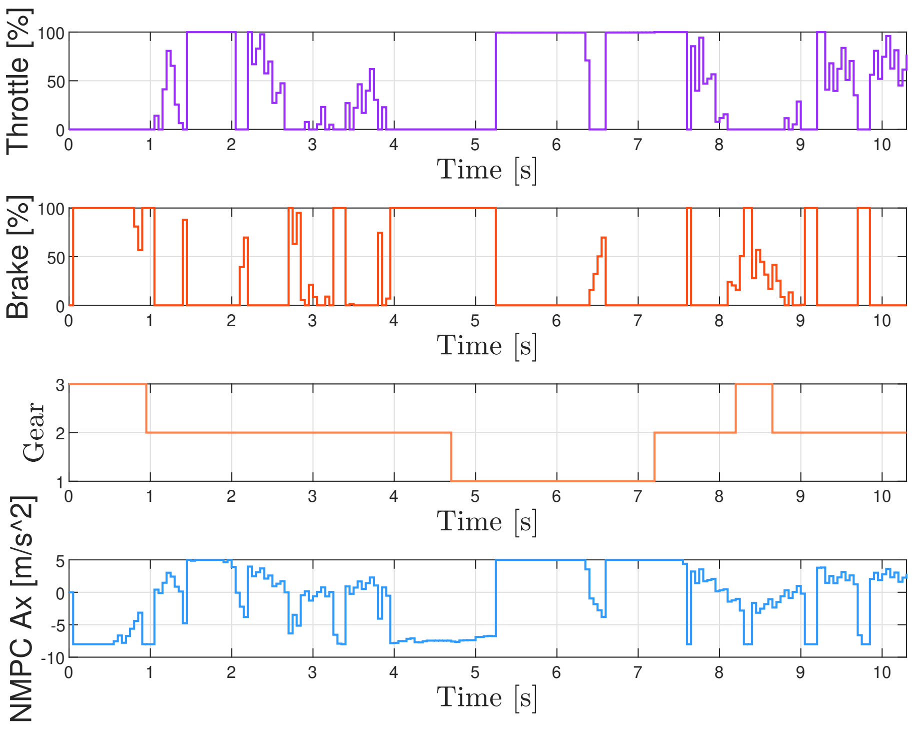


Figure 9.10: Throttle, Brake, Gear, NMPC Acceleration - Urban Merging

Control Variables plot

In the first second of the simulation, the vehicle applies almost maximum braking: this behavior is evident in the graphs by the brake peak close to 100 % and the rapid drop in longitudinal acceleration, indicating that the car is responding to the presence of vehicles passing on the main road at that moment. During this phase, the transmission quickly shifts down from third to second gear, as shown in the third panel of the first graph.
Next, as the vehicle approaches the stop sign, there is a marked alternation between throttle and brake: the throttle is modulated with even high peaks, while the brake is activated intermittently. In this phase, the throttle prevails, and as a result, the vehicle's speed increases progressively until about 4 seconds, as can also be seen from the longitudinal acceleration (*NMPC Ax*) showing positive values.
Between 5 and 6 seconds, the vehicle brakes at maximum: the brake command again reaches values close to 100 %, the longitudinal acceleration drops to its minimum, and the transmission shifts down to first gear (gear 1), indicating that the vehicle is completing the stopping phase near the stop sign.
After this phase, the car starts again with strong acceleration (throttle at 100 %), showing that the vehicle is merging onto the main road. Shortly after 7 seconds, the transmission shifts back up to second gear (gear 2), accompanied by a rapid increase in longitudinal acceleration.
In the final part of the simulation, after merging, there is a sequence of sharp accelerations and brakings: throttle and brake alternate rapidly, the transmission briefly shifts into third gear (gear 3) before stabilizing again in second gear until the end of the simulation. This behavior reflects the need to dynamically adapt to traffic and maintain control in

the phases following the merge.
This sequence of events clearly shows how the NMPC control system reactively manages the different phases of the urban merging maneuver, alternating decisive braking, rapid acceleration, and gear changes to ensure safety and responsiveness to traffic conditions.

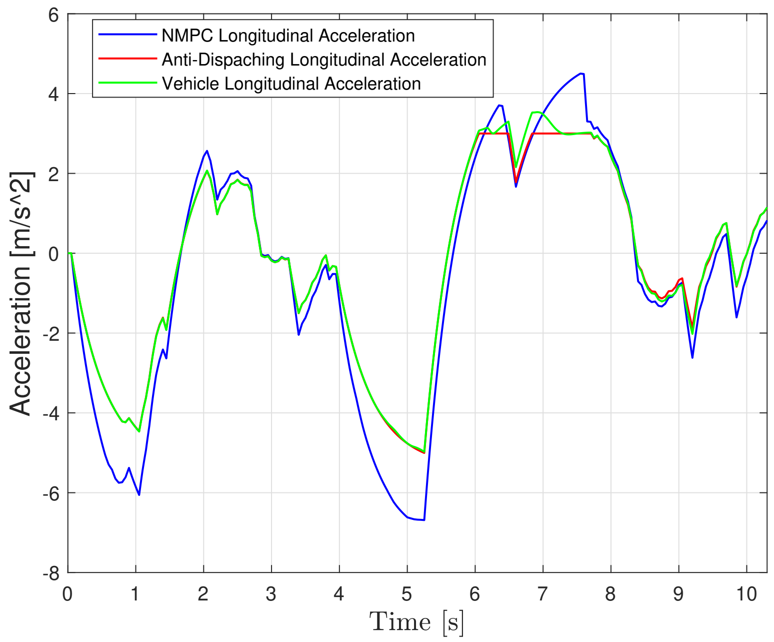


Figure 9.11: Accelerations - Urban Merging

Accelerations plot
Regarding the Anti-Dispatching acceleration, it was saturated at 3 m/s^2 instead of 5 m/s^2. This prevents a sudden restart after the stop sign and also leads to a reduction in the cross-track error, which would otherwise reach 2 m during the merging maneuver.
The three curves follow a similar trend, with rapid changes between positive values (acceleration) and negative values (braking), reflecting the different phases of the maneuver: intense deceleration near the stop, followed by strong acceleration for merging, and then further decelerations and accelerations to adapt to traffic after restarting. The most pronounced peaks of the blue curve (NMPC) are slightly smoothed out in the red curve (Anti-Dispatching), since the latter, by converting throttle and brake commands into acceleration, respects the physical limits of the vehicle.
The green curve (Vehicle real longitudinal acceleration) is very similar to the red curve. The only differences occur when the Anti-Dispatching acceleration is saturated, and in those cases, the green curve reaches slightly higher values.

9.3 Roundabout Merging

The multi-vehicle merging scenario with traffic circle represents one of the most complex situations from a planning and control perspective for autonomous vehicles. In this context, the ego-vehicle must merge into a traffic circle, interacting with other vehicles traveling on the same road infrastructure, each with different speeds and trajectories. Managing this

maneuver requires simultaneous consideration of dynamic, safety, and comfort constraints, as well as trajectory planning that minimizes collision risk and maximizes traffic flow.
This scenario was created with the "Driving Scenario Designer" app, importing from "OpenStreetMaps" a traffic circle present on the street of Via Dante Alighieri - Matera.

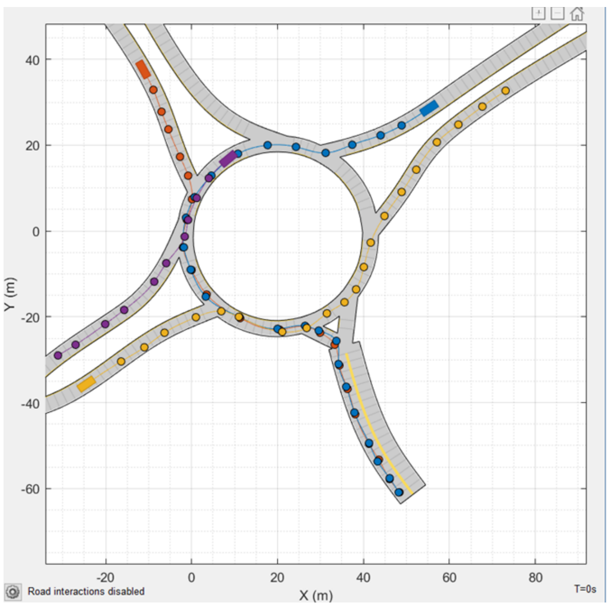
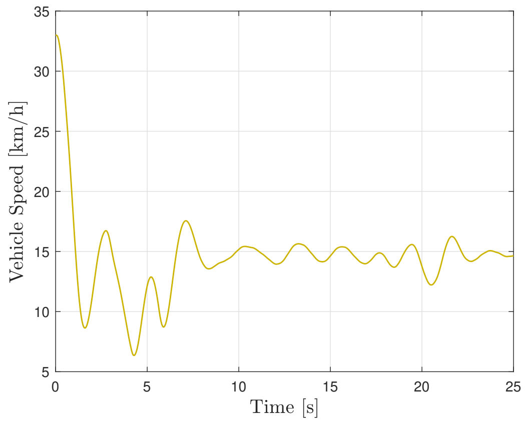


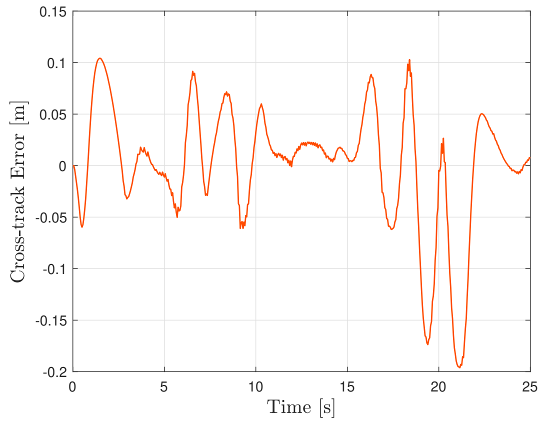
Figure 9.12: Roundabout Scenario

The interpolation used is of the "makima" type, in order to obtain the smoothest possible trajectory and to have a more controlled behavior by changing the curvature parameter Kc. In fact, as that parameter increases, the vehicle slows down more at the entry point, distancing itself more from other vehicles. Moreover, before the point of entry $(ct = 50)$ the reference speed is the curvature adaptive speed v_adapt; once entered, the vehicle follows the reference longitudinal speed vx_ref, since, if it continued to use v_adapt with $Kc = 10.2$, the vehicle would stop at every change of curvature in the traffic circle, resulting in an unrealistic behavior.
The ego-vehicle follows a reference speed of 33 km/h, and the other three are placed in a line according to their speed in the traffic circle: the first two ahead are traveling at 20 km/h, and they are already in the traffic circle, so the controlled vehicle must give them the right of way. The other vehicle is initially on an urban road at 30 km/h, then enters reaching a speed of 15 km/h, thus being behind the ego-vehicle, having entered after it.

9.3.1 Simulation Results - Roundabout Merging



(a) Vehicle Speed - Roundabout



(b) Cross-track Error

Figure 9.13: Speed and Error Plots - Roundabout

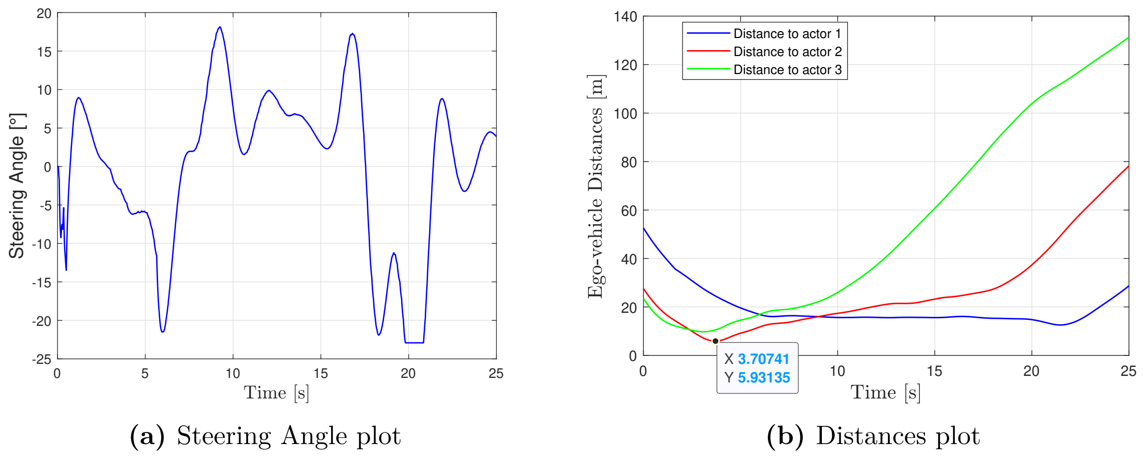
Plot a): Vehicle Speed
This plot describes the trend of vehicle speed during the maneuver.
The vehicle starts from 33 km/h and decelerates abruptly, causing 3 throats in the plot: the first as the vehicle perceives at some distance the passing 20 km/h vehicles; the second, after an acceleration made by the vehicle to get to the entry point, is due to both the increase in the curvature of the road and the excessive proximity with the second vehicle in the rear; and finally, the third due to the abrupt change in curvature between the traffic circle and the main road.
After these 3 threads, the vehicle reaches the entry point of the traffic circle, the reference speed becomes vx_ref, so the vehicle does not slow down too much at the curvature changes, but hovers around 15 km/h.
Around 20 s there is a new speed thread, and it is obtained at the time when the vehicle has to exit the traffic circle and enter the city road. Although the reference speed is not v_adapt, the NMPC still adapts to the change in curvature and slows down, preventing too high a deviation from the trajectory.

Plot b): Cross-track Error
In the graph, it can be seen that the error fluctuates around zero, with variations up to about +0.1 meters and -0.05 meters. The fluctuations indicate that the control system is constantly trying to correct the trajectory, but cannot completely cancel the error, probably due to the fact that the trajectory is circular, and the vehicle has to continuously adapt to each change in curvature, deviating to the right and left of the trajectory.

RMS e_{ct} [m]	**Max e_{ct} [m]**
0.0592	0.1961

The error overall is very low, since in this scenario the vehicle is proceeding at very very low speeds.
The highest error values are obtained at the 20 s, when the vehicle has to exit the traffic

circle. There is a really high curvature change that forces the vehicle to deviate more from the trajectory, causing an error of -0.2 m.



(a) Steering Angle plot **(b)** Distances plot

Figure 9.14: Steering Angle and Distances Plots - Roundabout

Plot a): Steering Angle
The graph shows how the vehicle continuously changes the steering angle to follow the desired trajectory in the traffic circle. Frequent variations and oscillations can be seen, with both positive and negative peaks. This indicates that the vehicle is dealing with curves and changes of direction typical of a traffic circle, constantly adjusting the steering to maintain the correct trajectory and probably to avoid obstacles or other vehicles. In general, the values fluctuate between -23° and +18°. The highest steering angle values are obtained:

1. after 5 s, with -20° of steering as the vehicle is entering a traffic circle, and there is a not inconsiderable change in road curvature;

2. At 20 s, when the vehicle exits the traffic circle, and the vehicle reaches the steering limit imposed by the NMPC of about -23° for about 1 s.

Plot b): Distances to other vehicles
At first, all distances are decreasing, which suggests that the vehicle is approaching the actors (the ego-vehicle is going toward the entrance of the traffic circle). The lowest distance is 5.93 m from the second vehicle in line at 20 km/h.
As time passes, the distances tend to increase, especially for the second and third actors (red and green), which reach very high values (over 100 m for the third actor). This may indicate that the vehicle is gradually moving away from these actors, leaving them behind in the traffic circle, and simply because their trajectories diverge.
The distance to the first actor (blue), on the other hand, remains relatively constant and low, suggesting that this actor is close to the vehicle throughout the maneuver, perhaps because it is following it. The distance, after the entry is about constant at 17-18 m (the vehicles both have a speed around 15 km/h), until 20 s, where the ego vehicle slows down too much because of the change in curvature between traffic circle and city street, at the exit of the roundabout. Then the distances increase, as the ego-vehicle exits the traffic

circle.

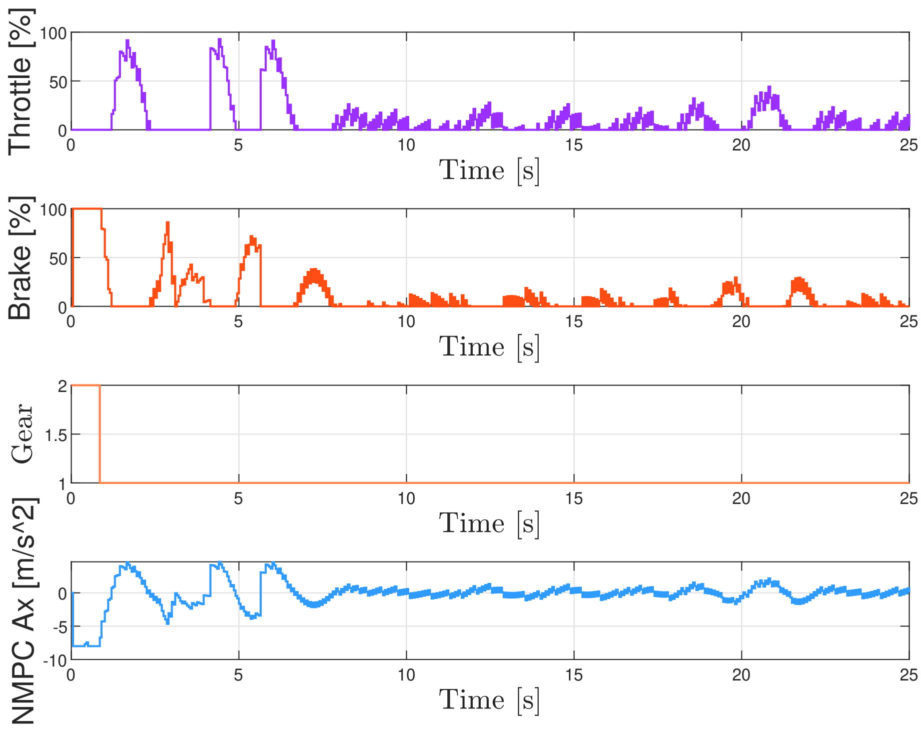


Figure 9.15: Throttle, Brake, Gear, NMPC Acceleration - Roundabout

Control Variables plot

This group of plots represent, in their entirety, the longitudinal behavior of a vehicle controlled by a Nonlinear Model Predictive Control (NMPC) system while driving through a traffic circle. The four plots are temporally coordinated (x-axis: time in seconds) and show how the vehicle handles throttle, brake, gear, and longitudinal acceleration to deal with the maneuver safely and efficiently.

In the initial phase (0-5 s), obvious peaks are observed in both throttle (Throttle) and brake (Brake) control. This indicates that the vehicle rapidly alternates between acceleration and deceleration, typically to adapt speed to traffic circle entry and to respond to safety and precedence constraints. In this phase, the gear (Gear) remains constant at 1, consistent with low speed and frequent changes in pace. The longitudinal acceleration required by the NMPC (NMPC Ax) shows fluctuations between positive and negative values, reflecting precisely these rapid changes between acceleration and braking.

In the intermediate phase (5-15 s), throttle and brake commands become more intermittent and less intense: the vehicle alternates between short accelerations and moments of release, while the brake is used less frequently. The gear remains unchanged, confirming that the vehicle is proceeding at a low speed, typical of traffic circle travel. NMPC acceleration stabilizes at more moderate values, with fewer oscillations, a sign that the vehicle's longitudinal dynamics are more controlled and less prone to abrupt changes.

In the final phase (15-25 s), the behavior remains similar: accelerator and brake are finely modulated, with small adjustments to maintain speed and safety until exiting the traffic circle. The gear does not change, and the NMPC acceleration remains in a narrow range, a sign that the vehicle has reached a condition of dynamic equilibrium.

The alternating and coordinated use of accelerator and brake, together with the choice to maintain a low gear, shows how the NMPC system manages the maneuver conservatively, constantly adapting speed to the conditions of the traffic circle and any safety constraints.

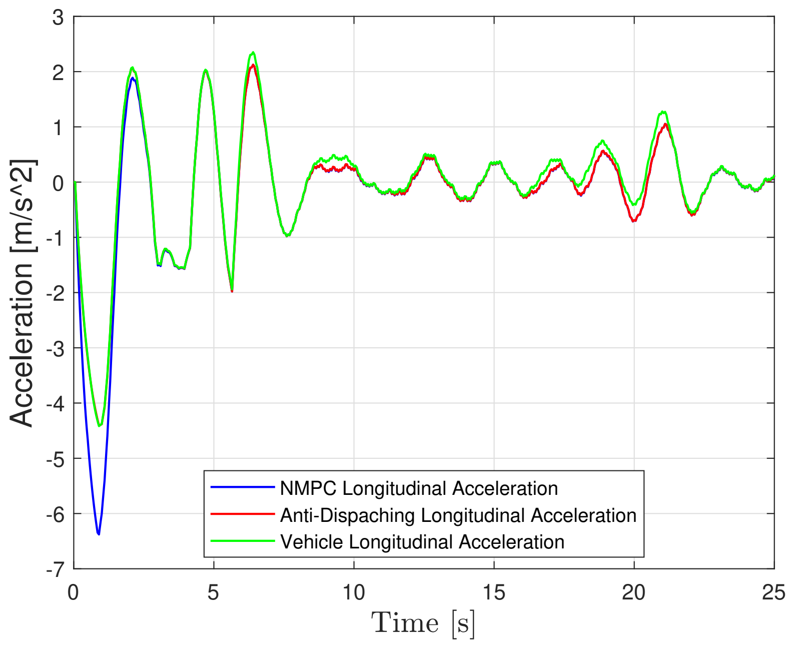


Figure 9.16: Accelerations - Roundabout

Acceleratons plot

In the initial phase (0-5 seconds), strong oscillations and both positive and negative peaks are observed in all three curves. This indicates that the vehicle is making rapid changes in acceleration and deceleration, typical of the phase of entering a traffic circle, where it is necessary to reduce speed and adapt to driving conditions. The presence of negative peaks corresponds to braking or deceleration, while positive peaks correspond to short accelerations.
In the intermediate phase (5-15 seconds), the curves begin to overlap more and the oscillations smooth out. This means that the vehicle is stabilizing its longitudinal dynamics, more closely following the indications of the NMPC controller and the anti-displacement system. The difference between required and actual acceleration is reduced, a sign of good vehicle control and response.
In the final phase (15-25 seconds), acceleration is maintained at smaller values with small variations, indicating a smoother and steadier ride, typical of the phase of exiting the traffic circle and returning to stable speed.
The correspondence between the three curves in the middle and final part of the graph indicates that the acceleration set by the NMPC, and consequently the Throttle and Brake controls, are never saturated; therefore, the acceleration set by the NMPC accurately reflects the physical limits of the vehicle.

Chapter 10

Conclusions

10.1 Final Remarks

In this thesis, a comprehensive Nonlinear Model Predictive Control (NMPC) framework for autonomous vehicles was developed, integrated, and validated within the CARLA simulation environment. The proposed control architecture demonstrates strong performance in both standard and highly complex driving scenarios, such as urban and extra-urban merging and roundabout navigation. The modular design of the Simulink blocks and the robust MATLAB–CARLA interface enable flexible experimentation and rapid prototyping of advanced control strategies. Simulation results confirm the effectiveness, robustness, and real-time feasibility of the NMPC approach for trajectory tracking and decision-making in dynamic, uncertain environments. The dispatching function, which translates NMPC acceleration commands into actuator signals while considering simplified longitudinal and gear-shifting models, further enhances the practical applicability of the system. Overall, the work provides a solid foundation for future research and development in the field of autonomous vehicle control systems.

10.2 Problems and Limitations

While the proposed control system has demonstrated robust performance across a variety of scenarios, it is important to acknowledge several limitations that currently affect its generality and adaptability.

First, the speed imposed by the controller is inherently tied to the structure of the prediction horizon. Specifically, the controller aims to reach the furthest reference point provided within the prediction horizon T_P, which is determined by the variable N_P. This means that the maximum achievable speed is not always an independent variable but is instead a function of the horizon's length and the way reference points are distributed. Such a dependency can limit the system's flexibility, as the optimal speed should ideally be selected dynamically based on the specific driving situation and actively tracked by the controller, rather than being dictated by fixed horizon parameters.

Another notable limitation concerns the implementation itself. Many of the Simulink blocks and functions were developed in an ad hoc manner, tailored specifically for the scenarios and vehicle model considered in this work. While this approach enabled effective control and facilitated scenario-specific tuning, it also reduces the generalizability of the system. Applying these blocks in different environments or with alternative vehicle models

would likely require significant redesign or parameter adjustment, thereby limiting the scalability of the current solution.
The performance of the NMPC controller is also highly sensitive to parameter tuning, particularly the weights in the cost function. These parameters were meticulously adjusted to achieve a balance between tracking accuracy, robustness, and computational efficiency for the specific scenarios under study. However, this fine-tuning process is scenario-dependent and may necessitate substantial re-calibration when transferring the controller to new contexts or vehicles.
The dispatching model we're using is quite simplified, and the gearshift logic is also very basic. One key limitation is that it doesn't differentiate between upshifts and downshifts. In reality, shifting up and shifting down are very different processes, each affecting the vehicle's behavior in unique ways, especially regarding engine response and driving comfort. Moreover, the model doesn't take into account the engine's RPM (revolutions per minute) or the torque output. This omission means it cannot accurately simulate scenarios involving slopes—both uphill and downhill—where engine speed and torque play a critical role in maintaining performance and safety. Because of these simplifications, the simulated vehicle behavior may diverge from what would happen in real life, particularly in dynamic driving situations or on roads with varying gradients. To improve the model's accuracy, it would be necessary to incorporate a more realistic gearshift logic that considers engine RPM, torque, and the different dynamics of upshifting versus downshifting. Additionally, a more detailed dispatching model would help better capture the vehicle's longitudinal dynamics under varying conditions. While these simplifications are common in many control and simulation frameworks—often to reduce computational complexity or improve robustness—they do limit the fidelity of the simulation, especially when dealing with complex driving scenarios like hill climbs or descents.
Additional limitations emerged during the implementation and validation of complex scenarios, such as lane merging and roundabout navigation. For instance, the gain parameter K_c required careful tuning for each scenario, and a single value could not be found that was optimal across all conditions. This necessitated compromises and highlighted the need for scenario-specific calibration. Furthermore, distinct differences were observed between scenarios: in the extra-urban lane merging case, it was necessary to introduce an additional block to enforce maximum acceleration immediately after merging, as well as a step change in the reference speed. In contrast, the urban merging scenario required the use of an adaptive or zero speed profile, rather than an adaptive or fixed reference speed, and imposed an extra constraint on maximum acceleration to ensure safety and comfort in lower-speed, stop-and-go conditions.
Finally, the simulation environment itself requires careful management to ensure realistic and safe interactions. For example, when a vehicle merges at low speed, the following vehicle must adapt its speed appropriately to avoid a frontal collision with the ego vehicle. This aspect is not yet fully automated in the current setup and highlights the need for more advanced interaction protocols and environment monitoring to guarantee safety in all possible traffic configurations.
Addressing these limitations in future work will be essential to enhance the robustness, adaptability, and general applicability of the control system to a broader range of vehicles and driving scenarios.

10.3 Improvements and Future Works

Building upon the results and limitations identified in this work, several promising directions can be pursued to further enhance the robustness, flexibility, and applicability of the proposed NMPC-based control system for autonomous vehicles.

One of the most immediate and impactful improvements would be the migration of the entire control framework from MATLAB/Simulink to Python. Python is rapidly becoming the standard in robotics and autonomous systems, thanks to its powerful open-source libraries, extensive community support, and seamless integration with platforms like CARLA. By transitioning to Python, the system would benefit from improved scalability, easier integration with machine learning modules, and the possibility to leverage advanced optimization and perception libraries. Moreover, the direct Python API for CARLA would simplify the simulation pipeline and enable more efficient real-time testing and deployment.

Another crucial area for future work is the schematization and realistic modeling of sensors. The current implementation uses idealized sensor models, which do not fully capture the challenges encountered in real-world perception, such as noise, latency, limited field of view, or occlusions. Future developments should focus on integrating detailed models of LiDAR, radar, camera, and GNSS/IMU sensors within the simulation environment. This would allow for the testing and validation of perception algorithms under more realistic conditions and facilitate the development of robust sensor fusion strategies. Additionally, simulating sensor faults or degraded performance would support the creation of fault-tolerant control architectures, a key requirement for safe autonomous driving.

Improving the dispatching and gear-shifting logic represents another important development path. The current dispatching function, which translates NMPC acceleration commands into throttle, brake, and gear signals, relies on simplified vehicle models and fixed rules. In future work, this module could be enhanced by adopting adaptive or learning-based strategies that better reflect the complexities of real powertrains, including electric and hybrid systems. More accurate models would allow for smoother gear transitions, improved energy efficiency, and greater passenger comfort. Additionally, the dispatching logic could be extended to dynamically adjust to varying road conditions, traffic scenarios, and vehicle states.

From a control perspective, further refinement of the NMPC model itself is warranted. This could involve incorporating more detailed lateral and longitudinal vehicle dynamics, as well as adaptive cost function weights that automatically tune themselves to changing scenarios. Such enhancements would improve the controller's ability to handle highly dynamic maneuvers, sharp turns, and complex interactions with other vehicles. Moreover, the integration of data-driven or AI-based modules could enable the controller to learn from past experiences and continuously improve its performance over time.

Finally, expanding the system's capability to handle multi-agent and cooperative scenarios would be a significant step forward. By enabling communication and coordination between multiple autonomous vehicles (V2V) or between vehicles and infrastructure (V2X), the control framework could support advanced maneuvers such as platooning, cooperative merging, and intersection management. This would not only improve traffic efficiency but also enhance safety in dense and complex urban environments.

In summary, the future development of this work should focus on transitioning to more flexible and scalable software platforms, increasing the realism and robustness of perception and actuation modules, and enhancing the intelligence and adaptability of the control algorithms. These steps will be essential to bring the proposed system closer to practical deployment in real-world autonomous vehicles and to address the ever-growing challenges of modern mobility.

Bibliography

[1] *What Are the Six Levels of Vehicle Automation?* Accessed: 2025-04-22. n.d. URL: https://www.ptolemus.com/what-are-the-six-levels-of-vehicle-automation/ (cit. on p. 6).

[2] Vittorio Ravello. *Lecture 23: Electric Architecture and ePWT Components for SAE Level 4 and 5 Autonomous Electric Vehicles.* E-Powetrain Components course, 2023-2024. 2024 (cit. on p. 7).

[3] Luisa Andreone. *Lecture 20, Stellantis / CRF.* Communication and Network System course. 2024 (cit. on pp. 8, 18).

[4] *What is ADAS? Levels, Working and More.* Accessed: 2025-04-22. n.d. URL: https://www.carblogindia.com/what-is-adas-levels-working/ (cit. on p. 8).

[5] *List of Advanced Driver Assistance Systems (ADAS) and their Function.* Accessed: 2025-04-22. n.d. URL: https://www.acsysteme.com/en/multimedia-resources/list-of-advanced-driver-assistance-systems-adas-and-their-function/ (cit. on p. 9).

[6] *Types of ADAS Sensors.* Accessed: 2025-04-22. n.d. URL: https://dewesoft.com/blog/types-of-adas-sensors (cit. on p. 10).

[7] *What is LiDAR? An Introduction to the Technology.* Accessed: 2025-04-22. n.d. URL: https://quantum-systems.com/blog/2024/12/09/what-is-lidar/ (cit. on p. 10).

[8] *How to Choose a LiDAR.* Accessed: 2025-04-22. n.d. URL: https://store.clearpathrobotics.com/blogs/blog/how-to-choose-a-lidar (cit. on p. 11).

[9] *What cameras are installed in ADAS and AD systems? - Groups of devices that contribute to the enhanced performance of equipment -.* Accessed: 2025-04-22. n.d. URL: https://industrial.panasonic.com/ww/ds/ss/technical/ap2 (cit. on p. 11).

[10] *Automotive Ultrasonic Sensors.* Accessed: 2025-04-22. n.d. (Cit. on p. 11).

[11] Guido Albertengo. *17 - Autonomous Driving.* Communication and Network System course. 2024 (cit. on p. 12).

[12] *Waymo: dove avanza e dove rallenta la società di auto a guida autonoma di Google.* Accessed: 2025-04-22. n.d. URL: https://www.economyup.it/automotive/self-driving-car/waymo-dove-avanza-e-dove-rallenta-la-societa-di-auto-a-guida-autonoma-di-google/ (cit. on p. 12).

[13] Bouchard Frederic, Sedwards Sean, Czarnecki Krzysztof. «A Rule-Based Behaviour Planner for Autonomous Driving». In: *Rules and Reasoning (2022)* 263-279 (2024). URL: https://arxiv.org/abs/2407.00460 (cit. on p. 13).

[14] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, Karol Zieba. «End to End Learning for Self-Driving Cars». In: (2016). URL: `https://arxiv.org/abs/1604.07316` (cit. on p. 13).

[15] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, Gigel Macesanu. «A Survey of Deep Learning Techniques for Autonomous Driving». In: *Journal of Field Robotics, Online ISSN:1556-4967, 2019* Article ID: ROB21918 (2016). URL: `https://arxiv.org/abs/1910.07738` (cit. on p. 14).

[16] *Veicoli autonomi: 20 milioni in strada entro il 2025 per un mercato globale stimato 65 miliardi di dollari*. Accessed: 2025-04-22. n.d. URL: `https://www.key4biz.it/veicoli-autonomi-20-milioni-in-strada-entro-il-2025-per-un-mercato-globale-stimato-65-miliardi-di-dollari/385400/` (cit. on p. 14).

[17] Rongliang Zhou a, Haotian Cao b, Jiakun Huang a, Xiaolin Song a, Jing Huang a, Zhi Huang a. «Hybrid lane change strategy of autonomous vehicles based on SOAR cognitive architecture and deep reinforcement learning». In: *Neurocomputing, Volume 611* 128669 (2025). URL: `https://www.sciencedirect.com/science/article/abs/pii/S0925231224014401` (cit. on p. 14).

[18] *In-Vehicle Computing | NVIDIA*. Accessed: 2025-04-22. n.d. URL: `https://www.nvidia.com/it-it/self-driving-cars/in-vehicle-computing/` (cit. on p. 15).

[19] Samuel Mbugua, Julia N. Korongo, Samuel Mbuguah. *On Software Modular Architecture: Concepts, Metrics and Trends*. 2022. URL: `https://www.researchgate.net/publication/359922034_On_Software_Modular_Architecture_Concepts_Metrics_and_Trends` (cit. on p. 15).

[20] Vito La Vecchia. *Le principali architetture dei sistemi distribuiti*. Accessed: 2025-04-24. n.d. URL: `https://vitolavecchia.altervista.org/le-principali-architetture-dei-sistemi-distribuiti/` (cit. on p. 16).

[21] Guido Albertengo. *Lecture 09: WAVE and ITS-G5*. Communication and Network System course. 2024 (cit. on p. 17).

[22] *The V-Cycle for Automotive Vehicle Development*. Accessed: 2025-04-24. n.d. URL: `https://www.researchgate.net/figure/The-V-cycle-for-automotive-vehicle-development_fig45_30513529` (cit. on p. 18).

[23] *Software Engineering - SDLC V-Model*. Accessed: 2025-04-24. n.d. URL: `https://www.geeksforgeeks.org/software-engineering-sdlc-v-model/` (cit. on p. 19).

[24] *V-Model Software Development*. Accessed: 2025-04-24. n.d. URL: `https://x-engineer.org/v-model-software-development/` (cit. on p. 19).

[25] *Model-in-the-loop*. Accessed: 2025-04-24. n.d. URL: `https://it.wikipedia.org/wiki/Model-in-the-loop` (cit. on p. 19).

[26] *Software-in-the-loop*. Accessed: 2025-04-24. n.d. URL: `https://it.wikipedia.org/wiki/Software-in-the-loop` (cit. on p. 20).

[27] *Hardware-in-the-loop*. Accessed: 2025-04-24. n.d. URL: `https://it.wikipedia.org/wiki/Hardware-in-the-loop` (cit. on p. 20).

[28] *Driving simulators*. Accessed: 2025-07-07. n.d. URL: https://www.abdynamics.com/driving-simulators/#:~:text=Driver-in-the-Loop%20%28DIL%29%20simulation%20brings%20all%20the%20elements%20of,new%20vehicles%2C%20components%2C%20active%20safety%20systems%20and%20more (cit. on p. 20).

[29] *Traversing the V-Cycle: A Single Simulation Application for the Renault 1.5 dCi Passenger Car Diesel Engine*. 2013. URL: https://saemobilus.sae.org/papers/traversing-v-cycle-a-single-simulation-application-renault-15-dci-passenger-car-diesel-engine-2013-01-1120 (cit. on p. 21).

[30] *Automotive Simulation Models*. Accessed: 2025-04-24. n.d. URL: https://www.dspace.com/en/pub/home/products/sw/automotive_simulation_models.cfm (cit. on p. 21).

[31] *Hardware-in-the-loop simulation*. Accessed: 2025-04-24. n.d. URL: https://en.wikipedia.org/wiki/Hardware-in-the-loop_simulation (cit. on p. 21).

[32] M. Bertozzi, A. Broggi, M. Felisa, and A. Tibaldi. «V-Cockpit: A Platform for the Design, Testing, and Validation of Car Infotainment Systems through Virtual Reality». In: *MDPI* (2021). Accessed: 2025-07-07. URL: https://www.mdpi.com/1424-8220/21/4/1234 (cit. on p. 21).

[33] *CARLA Release 0.9.14*. Accessed: 2025-07-07. 2022. URL: https://carla.org/2022/12/23/release0.9.14/ (cit. on p. 21).

[34] *Vladlen Koltun*. Accessed: 2025-07-07. n.d. URL: https://en.wikipedia.org/wiki/Vladlen_Koltun (cit. on p. 21).

[35] *CARLA Explained*. Accessed: 2025-07-07. n.d. URL: https://paperswithcode.com/lib/carla (cit. on p. 22).

[36] *CARLA Documentation*. Accessed: 2025-07-07. n.d. URL: https://carla.readthedocs.io/ (cit. on p. 22).

[37] *SuperAnnotate*. Accessed: 2025-07-07. n.d. URL: https://www.superannotate.com/ (cit. on p. 23).

[38] *LGSVL Simulator*. Accessed: 2025-07-07. n.d. URL: https://repo.valu3s.eu/tools/external-not-improved-tool/lgsvl (cit. on p. 23).

[39] *LGSVL Simulator on GitHub*. Accessed: 2025-07-07. n.d. URL: https://github.com/lgsvl/simulator (cit. on p. 23).

[40] *DeepAI*. Accessed: 2025-07-07. n.d. URL: https://deepai.org/ (cit. on p. 24).

[41] Guodong Rong and Byung Hyun Shin and Hadi Tabatabaee and Qiang Lu and Steve Lemke and Martins Mozeiko and Eric Boise and Geehoon Uhm and Mark Gerow and Shalin Mehta and Eugene Agafonov and Tae Hyung Kim and Eric Sterner and Keunhae Ushiroda and Michael Reyes and Dmitry Zelenkovsky and Seonman Kim. «LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving». In: *IEEE Intelligent Transportation Systems Conference (ITSC)* (2020) (cit. on pp. 24, 25).

[42] Matthias Müller, Vincent Casser, Jean Lahoud, Neil Smith, Bernard Ghanem. «Sim4CV: A Photo-Realistic Simulator for Computer Vision Applications». In: *International Journal of Computer Vision (IJCV)* (2018) (cit. on p. 26).

[43] *Sim4CV: A Photo-Realistic Simulator for Computer Vision Applications.* Accessed: 2025-07-07. 2018. URL: https://www.youtube.com/watch?v=SqAxzsQ7qUU (cit. on p. 26).

[44] *Yonohub meets Microsoft AirSim.* Accessed: 2025-07-07. 2022. URL: https://www.linkedin.com/pulse/yonohub-meets-microsoft-airsim-azure-g-alexander-kolbai (cit. on p. 27).

[45] *AirSim announcement: This repository will be archived in the coming year.* Accessed: 2025-07-07. n.d. URL: https://github.com/microsoft/AirSim (cit. on p. 27).

[46] *AirSim APIs.* Accessed: 2025-07-07. 2021. URL: https://microsoft.github.io/AirSim/apis/ (cit. on p. 27).

[47] *Aerial Informatics and Robotics Platform.* Accessed: 2025-07-07. n.d. URL: https://www.microsoft.com/en-us/research/project/aerial-informatics-robotics-platform/ (cit. on p. 27).

[48] *Using SITL with AirSim.* Accessed: 2025-07-07. n.d. URL: https://ardupilot.org/dev/docs/sitl-with-airsim.html (cit. on p. 27).

[49] *Microsoft AirSim.* Accessed: 2025-07-07. n.d. URL: https://www.helicomicro.com/2017/02/16/microsoft-airsim/ (cit. on pp. 27, 28).

[50] *Sensors in AirSim.* Accessed: 2025-07-07. n.d. URL: https://microsoft.github.io/AirSim/sensors/ (cit. on p. 27).

[51] *CarMaker.* Accessed: 2025-07-07. n.d. URL: https://www.ipg-automotive.com/en/products-solutions/software/carmaker/ (cit. on pp. 28, 29).

[52] *Transfer a Simulink Plug-in Model from CarMaker/Office to CarMaker for Simulink?* Accessed: 2025-07-07. n.d. URL: https://www.ipg-automotive.com/en/support/support-request/faq/transfer-a-simulink-plug-in-model-from-carmaker-office-to-carmaker-for-simulink-151/ (cit. on p. 28).

[53] *Add-ons.* Accessed: 2025-07-07. n.d. URL: https://www.ipg-automotive.com/en/products-solutions/software/add-ons/ (cit. on p. 28).

[54] Yannik Weber and Stratis Kanarachos. «The Correlation between Vehicle Vertical Dynamics and Deep Learning-Based Visual Target State Estimation: A Sensitivity Study». In: *Sensors* (2019) (cit. on p. 29).

[55] *SCANeR 2024.2 Release Note.* Accessed: 2025-07-07. n.d. URL: https://www.avsimulation.com/en/scaner-2024-2-release-note/ (cit. on pp. 29, 30).

[56] *SCANeR.* Accessed: 2025-07-07. n.d. URL: https://www.avsimulation.com/en/scaner/ (cit. on p. 29).

[57] *Albero di trasmissione.* Accessed: 2025-07-07. n.d. URL: https://www.chimica-online.it/download/albero-di-trasmissione.htm (cit. on p. 32).

[58] *Cos'è la coppia motore?* Accessed: 2025-07-07. n.d. URL: https://www.qualesceg liere.it/trapano/coppia-motore/ (cit. on p. 35).

[59] Nicola Amati. *Longitudinal Dynamics.* Driver Assistance System Design B course. 2023-2024 (cit. on p. 36).

[60] Vittorio Ravello and Shailesh Hegde. *Practice 02 - Simplified Vehicle Dynamics and Standard Driving Cycles.* E-Powetrain Components course. 2023-2024 (cit. on pp. 37, 38).

[61] Carlo Novara. *Lane Keeping - part 1.* Driver Assistance System Design A course. 2023-2024 (cit. on pp. 38, 55, 56).

[62] Nacer Kouider M'Sirdi, Abdelhamid Rabhi, Nabila Zbiri, Y. Delanne. «Vehicle–road interaction modelling for estimation of contact forces». In: *n.d.* (2005) (cit. on p. 39).

[63] Alessandro Vigliani. *Lateral Dynamics - Single Track Model.* Vehicle Dynamics Simulation course. 2023-2024 (cit. on p. 40).

[64] Nicola Amati. *Lateral Dynamics - part 1.* Driver Assistance System Design B course. 2023-2024 (cit. on p. 42).

[65] Alessandro Vigliani. *Es02 Tire models.* Vehicle Dynamics Simulation course. 2023-2024 (cit. on p. 43).

[66] Hans B. Pacejka. *Tire and Vehicle Dynamics.* Delft University of Technology, The Netherlands, 2012 (cit. on p. 44).

[67] *Hans B. Pacejka.* Accessed: 2025-07-07. n.d. URL: `https://it.wikipedia.org/wiki/Hans_B._Pacejka#:~:text=Pacejka%20ha%20sviluppato%20una%20serie,pneumatici%20e%20di%20condizioni%20operative.` (cit. on p. 44).

[68] Douglas L. Milliken William F. Milliken. *Race Car Vehicle Dynamics.* SAE International, 1994 (cit. on p. 45).

[69] Sara Abdallaoui, Halima Ikaouassen, Ali Kribèche, Ahmed Chaibet, El-Hassane Aglzim. «Advancing autonomous vehicle control systems: An in-depth overview of decision-making and manoeuvre execution state of the art». In: *n.d.* (2023) (cit. on pp. 46, 47).

[70] Brad Templeton. «Waymo's 6th-Generation Robotaxi Is Cheaper. How Cheap Can They Go?» In: *Forbes* (2024) (cit. on p. 47).

[71] Carlo Novara. *Control of dynamic systems.* Driver Assistance System Design A course. 2023-2024 (cit. on pp. 47, 48).

[72] *Guida al PID Tuning di un Drone.* Accessed: 2025-07-07. 2021. URL: `https://northfpv.com/come-tunare-i-pid/` (cit. on p. 48).

[73] Carlo Novara. *Proportional Integrative Derivative control.* Driver Assistance System Design A course. 2023-2024 (cit. on p. 49).

[74] *Root Locus: Example 1.* Accessed: 2025-07-07. 2021. URL: `https://lpsa.swarthmore.edu/Root_Locus/Example1/Example1.html` (cit. on p. 50).

[75] Carlo Novara. *State Feedback Control.* Driver Assistance System Design A course. 2023-2024 (cit. on p. 50).

[76] Giuseppe Franze, Massimiliano Mattei, Luciano Ollio, and Valerio Scor damaglia. «A robust constrained model predictive control scheme for norm bounded uncertain systems with partial state measurements». In: *Interna tional Journal of Robust and Nonlinear Control* (2019) (cit. on p. 51).

[77] 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) (cit. on p. 51).

[78] S Joe Qin and Thomas A Badgwell. «An overview of nonlinear model predictive control applications». In: *Nonlinear model predictive control* (2000) (cit. on p. 51).

[79] Jacques Richalet, André Rault, JL Testud, and J Papon. «Model predictive heuristic control». In: *Automatica (journal of IFAC) 14.5* (1978) (cit. on p. 51).

[80] 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) (cit. on p. 51).

[81] David M Prett Carlos E Garcia and Manfred Morari. «Model predictive control: Theory and practice — A survey». In: *Automatica 25.3* (1989) (cit. on p. 51).

[82] Frank Allgöwer and Alex Zheng. «Nonlinear model predictive control. Vol. 26». In: *Birkhäuser* (2012) (cit. on p. 51).

[83] 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) (cit. on p. 51).

[84] Moritz Diehl, Hans Georg Bock, Holger Diedam, and P-B Wieber. «Fast direct multi- ple shooting algorithms for optimal robot control». In: *Fast motions in biomechanics and robotics: optimization and feedback control* (2006) (cit. on p. 51).

[85] Moritz Diehl, H.Georg Bock, Johannes P. Schlöder, Rolf Findeisen, Zoltan Nagy, and Frank Allgöwer. «Real-time optimization and nonlinear model predictive control of processes governed by differential-algebraic equations». In: *Journal of Process Control* 12.4 (2002), pp. 577–585 (cit. on p. 51).

[86] Toshiyuki Ohtsuka. «A continuation/GMRES method for fast computation of nonlinear receding horizon control». In: *Automatica* 40.4 (2004), pp. 563–574 (cit. on p. 51).

[87] Victor M. Zavala and Lorenz T. Biegler. «The advanced-step NMPC controller: Optimality, stability and robustness». In: *Automatica* 45.1 (2009), pp. 86–93 (cit. on p. 51).

[88] Mattia Boggio, Carlo Novara, and Michele Taragna. «Nonlinear Model Predictive Control: an Optimal Search Domain Reduction». In: *IFAC-PapersOnLine* 56.2 (2023). 22nd IFAC World Congress, pp. 6253–6258 (cit. on p. 51).

[89] Mattia Boggio, Carlo Novara, and Michele Taragna. «Set Membership identification for NMPC complexity reduction». In: *IFAC-PapersOnLine* 58.15 (2024). 20th IFAC Symposium on System Identification SYSID 2024, pp. 217–222 (cit. on p. 51).

[90] Janick V. Frasch, Andrew Gray, Mario Zanon, Hans Joachim Ferreau, Sebastian Sager, Francesco Borrelli, and Moritz Diehl. «An auto-generated nonlinear MPC algorithm for real-time obstacle avoidance of ground vehicles». In: *2013 European Control Conference (ECC)*. 2013, pp. 4136–4141 (cit. on p. 51).

[91] Ningyuan Guo, Basilio Lenzo, Xudong Zhang, Yuan Zou, Ruiqing Zhai, and Tao Zhang. «A Real-Time Nonlinear Model Predictive Controller for Yaw Motion Optimization of Distributed Drive Electric Vehicles». In: *IEEE Transactions on Vehicular Technology* 69.5 (2020), pp. 4935–4946 (cit. on p. 51).

[92] Mattia Boggio, Carlo Novara, and Michele Taragna. «Trajectory planning and control for autonomous vehicles: a "fast" data-aided NMPC approach». In: *European Journal of Control* (2023), p. 100857 (cit. on p. 51).

[93] KS Holkar and LM Waghmare. «Discrete model predictive control for dc drive using orthonormal basis function». In: (2010) (cit. on pp. 52, 53).

[94] Carlo Novara. *Non-Linear Model Predictive Control.* Driver Assistance System Design A course. 2023-2024 (cit. on pp. 52, 53).

[95] KS Holkar and Laxman M Waghmare. «An overview of model predictive control. «An overview of model predictive control». In: *International Journal of control and automation 3.4* (2010) (cit. on p. 53).

[96] *Introduction - CARLA Simulator.* Accessed: 2025-07-07. n.d. URL: `https://carla.readthedocs.io/en/latest/start_introduction/` (cit. on pp. 57, 58).

[97] *Traffic Manager - CARLA Simulator.* Accessed: 2025-07-07. n.d. URL: `https://carla.readthedocs.io/en/latest/adv_traffic_manager/` (cit. on pp. 58–60).

[98] *CARLA-PID-Controller.* Accessed: 2025-07-07. n.d. URL: `https://github.com/MoemenGaafar/CARLA-PID-Controller/blob/main/SpecsCalculations.m` (cit. on pp. 60, 61).

[99] *Measurements - CARLA Simulator.* Accessed: 2025-07-07. n.d. URL: `https://carla.readthedocs.io/en/0.9.5/measurements/` (cit. on p. 66).

[100] *Calcolatore area frontale.* Accessed: 2025-07-07. n.d. URL: `https://jumpjack.altervista.org/frontal-area/` (cit. on p. 83).

[101] Robert Bosch GmbH. *Bosch Automotive Handbook - 9th Edition.* n.d., 2014 (cit. on p. 83).

[102] Luco, Zhu. «Energy efficient cornering: Simulation and verification». In: *KTH Royal Institute of Technology* (2018) (cit. on p. 84).

[103] *ros-bridge/carla*$_a$*ckermann*$_c$*ontrol/src/carla*$_a$*ckermann*$_c$*ontrol/carla*$_c$*ontrol*$_p$*hysics.py.* Accessed: 2025-07-07. n.d. URL: `https://github.com/carla-simulator/ros-bridge/blob/master/carla_ackermann_control/src/carla_ackermann_control/carla_control_physics.py` (cit. on p. 84).

[104] *Seat Leon 1.9 TDi 100 Scheda Tecnica.* Accessed: 2025-07-07. n.d. URL: `https://www.ultimatespecs.com/it/auto-caratteristiche-tecniche/Seat/1182/Seat-Leon-19-TDi-100.html` (cit. on p. 95).