Politecnico di Torino

Politecnico di Torino

Master's Degree in Mechatronic Engineering
Control Technologies for Industry 4.0

A.Y. 2022/2023
Degree Session April 2023

Self-Driving Cars: Nonlinear MPC for Steering, Throttle and Brake Control

Perpendicular Parking Manoeuvres

Supervisor

Prof. Stefano Alberto MALAN

Candidate

Gianmarco PICARIELLO

Summary

Vehicles, since their invention, have undergone countless changes, ranging from advancements in aerodynamics, construction and internal components, and more recently in their degree of autonomy, even if the development of the first Automated Driving System (ADS) can be traced back to the 1920s. However, it is crucial to differentiate between *automated* and *fully automated* vehicles, with the former being designed to operate autonomously for a specific time period, while still requiring driver intervention, and the latter being designed to operate without any driver supervision. The Society of Automotive Engineers (SAE) in their "6 Levels of Automation" offers a comprehensive overview and classification of autonomous vehicles, ranging from level 0, no driving automation, to level 5, fully driving automation, which achievement is made possible by also a correct use and fusion of data of the on-board sensors. The successful implementation of Advanced Driver Assistance System (ADAS), thus of the autonomous driving in general terms, will not only offer benefits in terms of road safety and time-saving, but also improvement of infrastructures.

With the always more and more increasing demand for secure and efficient mobility, many organizations and scholars are dedicated to advancing the technology of autonomous driving. A notable competition in this field is the Bosch Future Mobility Challenge (BFMC), held by the Bosch Engineering Centre Cluj, that aims at fostering creativity and innovation, besides raising awareness on the potential of these emerging technologies. The BFMC invites student teams from across the world to develop autonomous driving algorithms for a 1:10 scale vehicle provided by the organizers. Among the algorithms required to be developed, the autonomous parking algorithm serves as the main motivation behind this thesis work, which delves into exploring and improving the knowledge and awareness of autonomous driving and parking. The topic of autonomous parking can be categorized into *semi-autonomous* and *fully autonomous* parking, with the former referring to automated vehicles, where the driver can stop the manoeuvre and regain control of the vehicle. In contrast, with fully autonomous Parking Assist (PA), the driver choice is limited to whether they prefer to assist the manoeuvre inside or outside the car.

Several control strategies are available in the literature for developing an autonomous parking algorithm, one of which involves the Model Predictive Control (MPC), along with its variants that are well-suited for nonlinear systems, namely the Nonlinear Model Predictive Control (NMPC) and its derivative, the Multistage NMPC (msNMPC). The vehicle model derivation forms the backbone of these algorithms and the vehicle kinematic single-track model is particularly suited for this task due to low speeds at which the manoeuvre is performed.

Therefore, this Master's Thesis examines the design and simulation of three control strategies for autonomous parking manoeuvres in perpendicular parking scenarios, using the Nonlinear Model Predictive Control algorithm. Inspired by the examples provided by The MathWorks, Inc. for parallel parking applications, the strategies are:

- Perpendicular Parking using Nonlinear Model Predictive Control.
- Perpendicular Parking using Multistage Nonlinear Model Predictive Control.
- Perpendicular Parking using RRT* Planner and NMPC Tracking Controller.

These manoeuvres address a likely real-case scenario: the backward motion, as illustrated by the blue line in Figure 1.

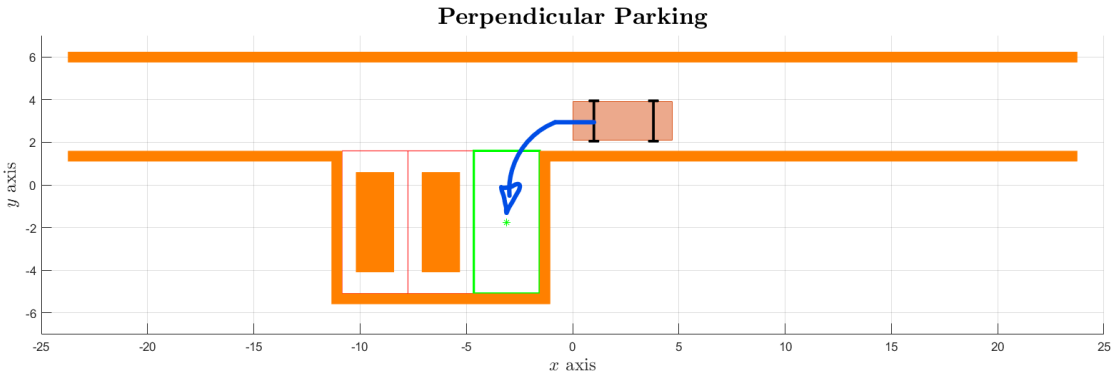


Figure 1: Perpendicular Parking Environment in MATLAB®.

Each case study is comprehensively covered from the MATLAB® functions used, to the simulation results. The first case study uses a Nonlinear Model Predictive Control to optimize the vehicle trajectory based on a predictive model of its dynamics. The second case study uses the extension of NMPC, the Multistage NMPC, which allows for the use of more accurate nonlinear models for prediction over longer horizons. The third case study combines optimized Rapidly-exploring

Random Tree (RRT*) path planning combined with NMPC tracking controller, using a probabilistic algorithm to generate a feasible trajectory of the vehicle. A comparison between the different approaches and their evaluation against a real-life scenario is also carried out.

The results showed that all the three approaches could achieve successful perpendicular parking manoeuvres, with some differences in terms of execution time and computational complexity. The msNMPC approach showed the best overall performance, with the shortest execution time and the most stable control output, besides of being more close to the reality. However, the choice of the best method will depend on the specific requirements and constraints of the application since each approach has its advantages and limitations like in the first and third case-study where the steering angle δ saturates at ± 45 deg which is relatively large and unsuitable for most normal driving situations, but this is necessary to achieve a sharp turn to navigate the 90-degree curve (Fig. 1) and reach the parking spot.

Acknowledgements

"If you do not believe in yourself,
no one will do it for you."
Kobe Bryant

Table of Contents

List of Tables x

List of Figures xi

Acronyms xiv

1 Introduction 1
- 1.1 The 6 Levels of Vehicle Autonomy . . . 3
- 1.2 Required Sensors in Autonomous Vehicles . . . 5
- 1.3 Benefits of Autonomous Vehicles . . . 7
- 1.4 Thesis Outline . . . 10

2 The Bosch Future Mobility Challenge 2022 12
- 2.1 The Competition . . . 12
- 2.2 The Car-Kit . . . 16
- 2.3 The Project . . . 18
  - 2.3.1 Competition Documentation and First Steps . . . 18
  - 2.3.2 The Brain Project . . . 19
  - 2.3.3 The Embedded Project . . . 19
  - 2.3.4 The GitHub Repository . . . 19
- 2.4 The Structure behind the Algorithms . . . 20

3 Autonomous Parking 22
- 3.1 State of the Art . . . 23
- 3.2 Autonomous Parking @ BFMC22 . . . 25
  - 3.2.1 The TF-Luna LiDAR Sensor . . . 28
  - 3.2.2 The HC-SR04 Ultrasonic Sensor Module . . . 29

4 Vehicle Model Derivation 32
- 4.1 Vehicle Longitudinal Model . . . 32
  - 4.1.1 Longitudinal Vehicle Dynamics . . . 33

4.1.2 Driveline Dynamics . . . 37
4.2 Vehicle Lateral Model . . . 42
4.2.1 Kinematic Model of Vehicle Lateral Motion . . . 42
4.2.2 Single-Track Model of Lateral Vehicle Dynamics . . . 46
4.3 Ego Vehicle Model for NMPC . . . 50

5 Model Predictive Control 52
5.1 Introduction . . . 52
5.1.1 MPC Pipeline . . . 54
5.1.2 Design Parameters . . . 54
5.2 Applications in the Autonomous Driving . . . 57
5.3 Types of MPC . . . 59
5.3.1 Linear MPC . . . 59
5.3.2 Nonlinear MPC . . . 62
5.3.3 Multistage Nonlinear MPC . . . 66

6 Perpendicular Parking: Software Implementation 69
6.1 Perpendicular Parking using Nonlinear MPC . . . 69
6.1.1 Parking Environment . . . 70
6.1.2 Ego Vehicle Definition . . . 72
6.1.3 Design the Nonlinear Model Predictive Controller . . . 74
6.1.4 Controller Simulation in MATLAB® . . . 78
6.2 Perpendicular Parking using Multistage Nonlinear MPC . . . 83
6.2.1 Vehicle Path Planner System Block Configuration . . . 84
6.2.2 Controller Simulation in MATLAB® and Simulink® . . . 86
6.3 Perpendicular Parking using RRT* Planner and NMPC Tracking Controller . . . 91
6.3.1 Trajectory Planning and the RRT, RRT* Algorithms . . . 91
6.3.2 Path Planning from RRT* in MATLAB® . . . 93
6.3.3 Design of the NMPC Tracking Controller . . . 94
6.3.4 Controller Simulation in MATLAB® . . . 95

7 Conclusion 99

A Autonomous Parking Algorithm @ BFMC22 101

B helperSLVisualizeParking.m 103

C parkingVehicleStateFcn.m 106

D parkingCostFcn.m 107

E parkingIneqConFcn.m 108

F Nonlinear Model Predictive Controller 110

G analyseParkingResults.m 112

H runParkingAndPlot.m 113

I plotAndAnimateParking.m 114

J Perpendicular Parking with Multistage NMPC 116

K Perpendicular Parking using RRT* and NMPC 119

L parkingStateValidator.m 121

Bibliography 124

List of Tables

3.1 Parameters Specification of the TF-Luna LiDAR Sensor. 29
3.2 Parameters Specification of the HC-SR04 Ultrasonic Sensor Module. 31

6.1 Relevant Dimensions of the Parking Scenario. 71
6.2 Default Dimensions of the Ego Vehicle. 74
6.3 CS 1: NMPC Design Parameters Values. 76
6.4 CS 1 (Trials 1 to 4): Values for the Weight Matrices R_p, Q_t and R_t. 78
6.5 CS 1 (Trials 1 to 4): Tuning of the Process Weight Matrices. 78
6.6 CS 1 (Trials 1 to 4): Comparison of Parking Results Analysis. . . . 79
6.7 CS 1 (Trials 5 to 9): Tuning of the Terminal Weight Matrices. . . . 80
6.8 CS 1 (Trials 10 to 13): Tuning of the Terminal Weight Matrices. . . 81
6.9 CS 1 (Trial 12): Parking Results Analysis. 81
6.10 CS 2 (Trials 1 to 4): Loosening of the Speed Bounds. 87
6.11 CS 2 (Trials 1 to 4): Comparison of Parking Results Analysis. . . . 88
6.12 CS 2 (Trial 5): Parking Results Analysis. 88
6.13 CS 2 (Trial 6): Parking Results Analysis. 89
6.14 CS 3: NMPC Tracking Controller Design Parameters Values. 94
6.15 CS 3: Parking Results Analysis. 95

List of Figures

1 Perpendicular Parking Environment in MATLAB®. iii

1.1 SAE J3016 Levels of Driving Automation. 4
1.2 Graphical Representation of the Six Levels of Driving Automation. 5
1.3 ADAS General Block Diagram. 5
1.4 ADAS and Related Sensors. 7
1.5 Benefits of Self-Driving in the EU. 10

2.1 Test Track. 14
2.2 Timeline. 15
2.3 Best New Participating Team Award. 16
2.4 The Car-Kit. 17
2.5 Website Layout of the Shared Documentation. 18
2.6 Team PoliTron's Complete Project Architecture. 21

3.1 Audi Parking System Plus. 23
3.2 PA General Block Diagram. 24
3.3 Parallel Parking Scenario @ BFMC22. 26
3.4 BFMC22 Competition Track with Nodes. 27
3.5 Schematics of the ToF Principle. 28
3.6 Pin-out of the HC-SR04 Ultrasonic Sensor Module. 30

4.1 Longitudinal Forces acting on a Vehicle driving on an Inclined Road. 33
4.2 Calculation of the Normal Tyre Loads. 36
4.3 Transmission System Layout for the 3 Different Types of Drive. . . 37
4.4 Power Flow and Loads in the Driveline. 38
4.5 I/O Schematic of a Torque Converter Model. 39
4.6 I/O Schematic of a Transmission Model. 39
4.7 I/O Schematic of an Engine Inertia Model. 41
4.8 I/O Schematic of a Wheel Inertia Dynamics Model. 41
4.9 Kinematics of Lateral Vehicle Motion. 42
4.10 Ackermann Turning Geometry. 45

4.11 Differential Steer from a Trapezoidal Tie-Rod Arrangement. 46
4.12 Lateral Vehicle Dynamics. 47
4.13 Tyre Slip Angle. 48
4.14 Sign Convention for the Banking Angle ϕ. 50

5.1 Concepts and Parameters constituting the MPC Problem. 56

6.1 Perpendicular Parking Environment in MATLAB®. 70
6.2 Parking Environment Dimensions. 72
6.3 Vehicle Dimensions stored in the `vehicleDimensions` Object. . . . 73
6.4 CS 1: Ego Vehicle Behaviour at the End of Trial 4. 79
6.5 CS 1: Ego Vehicle Behaviour at the End of Trial 9. 80
6.6 CS 1: Valid Result at Trial 12. 81
6.7 CS 1: Behaviour of the Ego Vehicle States at Trial 12. 82
6.8 CS 1: Behaviour of the Ego Vehicle Control Inputs at Trial 12. . . . 83
6.9 VPP System Block as per The MathWorks, Inc. Documentation. . . 83
6.10 VPP Simulink® Model. 84
6.11 VPP System Block Parameters. 85
6.12 msNMPC Controller Model. 85
6.13 CS 2: Ego Vehicle Behaviour at the End of Trial 3. 88
6.14 CS 2: Valid Result at Trial 6. 89
6.15 CS 2: Behaviour of the Ego Vehicle States at Trial 6. 90
6.16 CS 2: Behaviour of the Ego Vehicle Control Inputs at Trial 6. . . . 90
6.17 CS 3: Tree Expansion and Trajectory Generation in the Parking Environment. 94
6.18 CS 3: Ego Vehicle Behaviour at the End of the Simulation. 96
6.19 CS 3: Goodness of the NMPC Tracking Controller. 96
6.20 CS 3: Behaviour of the Ego Vehicle States. 97
6.21 CS 3: Behaviour of the Ego Vehicle Control Inputs. 97

Acronyms

ABS
Anti-Lock Brake System

ACC
Adaptive Cruise Control

AD
Autonomous Driving

ADAS
Advanced Driver Assistance System

ADS
Automated Driving System

AI
Artificial Intelligence

ALV
Autonomous Land Vehicle

API
Application Programming Interface

AR
Augmented Reality

AWD
All-Wheel Drive

BFMC
Bosch Future Mobility Challenge

CAS

Collision Avoidance System

CC

Cruise Control

CCW

counterclockwise

CoG

Center of Gravity

CPU

Central Processing Unit

CS

Case-Study

CSI

Camera Serial Interface

CTE

Cross-Tracking Error

CV

Control Variable

CW

clockwise

DARPA

Defense Advanced Research Projects Agency

DDT

Dynamic Driving Task

DOF

Degree of Freedom

EBS

Emergency Braking System

ECU

Electronic Control Unit

ERSO

European Road Safety Observatory

EU

European Union

EV

Electric Vehicle

FoV

Field of View

FWD

Front-Wheel Drive

GPIO

General Purpose Input/Output

GPS

Global Positioning System

HEV

Hybrid Electric Vehicle

ICE

Internal Combustion Engine

ICR

Instantaneous Center of Rotation

IMU

Inertial Measurement Unit

I/O

Input-Output

ISO

International Organization for Standardization

LED

Light-Emitting Diode

LFC

Lane-Following Control

LiDAR

Light imaging, Detection, and Ranging

LiPo

Lithium-ion polymer

LKA

Lane-Keeping Assist

LTI

Linear Time Invariant

MIMO

Multiple-Input Multiple-Output

MPC

Model Predictive Control

msNMPC

Multistage NMPC

MV

Manipulated Variable

NIR

Near-Infrared

NMPC

Nonlinear Model Predictive Control

OA

Obstacle Avoidance

PA

Parking Assist

PID

Proportional-Integral-Derivative

RADAR

Radio Detection and Ranging

ROS

Robot Operating System

RPi

Raspberry Pi

RRT

Rapidly-exploring Random Tree

RRT*

optimized Rapidly-exploring Random Tree

RWD

Rear-Wheel Drive

SAE

Society of Automotive Engineers

SiP

System in Package

SISO

Single-Input Single-Output

SoC

System on Chip

TCS

Traction Control System

TF

Transfer Function

ToF

Time of Flight

TTL

Transistor-Transistor Logic

V2X

Vehicle-to-Everything

VPP

Vehicle Path Planner

WHO

World Health Organization

Chapter 1

Introduction

Vehicles are the most important mean of transportation for people and/or cargoes. Land vehicles are classified by what is used to apply steering and drive force against ground: wheeled, tracked, railed or skied. ISO 3833-1977 is the international standard used in legislation, for road vehicle types, terms and definitions.

Since their invention back in 1886 thanks to Karl Benz who patented the three-wheeled motor car (Motorwagen) – even if Nicolas-Joseph Cugnot is often credited as the father of the first self-propelled mechanical vehicle or automobile (1769) – automobiles (and all their components) have withstood daily improvements bringing to what we see today.

Experiments on Automated Driving System (ADS) have been conducted since at least the 1920s, with the first trials in 1950s. The first semi-automated car was developed in 1977, by Japan's Tsukuba Mechanical Engineering Laboratory, which required specially marked streets that were interpreted by two cameras on the vehicle and an analogue computer.

A landmark autonomous car appeared in the 1980s with Carnegie Mellon University's Navlab and ALV (Autonomous Land Vehicle) projects funded by the United States' DARPA (Defense Advanced Research Projects Agency) starting in 1984 and Mercedes-Benz and Bundeswehr University Munich's EUREKA Prometheus Project in 1987 [1].

In these last years the concept of autonomous vehicles has become more and more predominant in our lives, especially in discussions in both the academic and industrial environments. This is because the development and mass production of self-driving cars has the potential to revolutionize the transportation mobility and safety. Currently, motor vehicle operating laws, impaired driving laws, insurance

laws and most other laws addressing the operation of vehicles in every country are based on a significant assumption: the human driver is behind the wheel and operates the vehicle. However, lawmakers around the world are considering the development and a future diffusion of driverless cars, including how existing laws and systems may need to be modified to facilitate the implementation of this new technology since autonomous vehicles will transform the way we live, work and play creating safer and more efficient roads. Yet, public opinion could see the results and problems about this technology and seems to be divided by this upcoming revolution in the mobility field.

Basically, an autonomous car is a vehicle able of sensing its environment and operating without human intervention. A human passenger is not required to take control of the vehicle at any time. He/She is neither required to be present in the vehicle at all. Indeed, a self-driving car can go anywhere a traditional car goes and do everything an experienced driver does.

The SAE (Society of Automotive Engineers) International currently defines 6 levels of driving automation spanning from Level 0 (fully manual) to Level 5 (fully autonomous). The SAE uses the term *automated* instead of *autonomous* and one reason is because the word *autonomy* has implications beyond the electromechanical. A *fully autonomous* car would be self-aware and capable of making its own choices, in fact the word *autonomous* means *self-governing*, while a *fully automated* car would follow orders and then drive itself. Many historical projects related to vehicle automation have been *automated* (i.e., made automatic) subject to a heavy reliance on artificial aids in their environment, such as magnetic strips. Autonomous control requires satisfactory performance under significant uncertainties in the environment and the ability to compensate for system failures without external intervention [2].

In Europe, the words *autonomous* and *automated* might be used together: Regulation (EU) 2019/2144 of the European Parliament and of the Council of 27 November 2019 [3] on type-approval requirements for motor vehicles defines *automated vehicle* and *fully automated* vehicle based on their autonomous capacity:

- *Automated vehicle*: motor vehicle designed and constructed to move autonomously for certain periods of time without continuous driver supervision but in respect of which driver intervention is still expected or required.

- *Fully automated vehicle*: motor vehicle designed and constructed to move autonomously without driver supervision.

However, the term *self-driving* is often used interchangeably with autonomous even if they are slightly different because a *self-driving* car can drive itself in some

or even all situations, but it requires the presence of a human passenger who is ready to take control. For this reason, as it will be shown later, *self-driving* cars fall under Level 3 (conditional driving automation) or Level 4 (high driving automation). They are subject to geofencing, unlike a fully autonomous Level 5 car that could go anywhere.

Self-driving cars are ushering in a new era on the roads. For automated driving to become reality, three key requirements must be met: The car must capture all its surroundings ("sense"), process this information and plan a driving strategy ("think") and then implement it reliably and safely ("act").

1.1 The 6 Levels of Vehicle Autonomy

The SAE International standard is the SAE J3016™ Recommended Practice: Taxonomy and Definitions for Terms Related to Driving Autonomous Systems for On-Road Motor Vehicle, commonly known as the **SAE Levels of Driving Automation™** (Figure 1.1 [4]). In the context of motor vehicles and their operation on roadways, it defines the SAE Levels from Level 0 (No Driving Automation) to Level 5 (Full Driving Automation), thus making it the industry's most-cited source for driving automation.

- Level 0 – No Driving Automation
  Most today's road vehicles belong to this level, i.e. manually controlled. The driver provides the *Dynamic Driving Task* (DDT) although there may be systems in place to help the driver like the EBS (Emergency Braking System), since it technically does not drive the vehicle, it does not qualify as automation.

- Level 1 – Driver Assistance
  Lowest level of automation – the vehicle features a single automated system for driver assistance, such as steering or accelerating (Cruise Control, CC). The Adaptive Cruise Control (ACC), where the vehicle can be kept at a safe distance from the ahead car, qualifies as Level 1 because the driver monitors other aspects of driving like steering and braking.

- Level 2 – Partial Driving Automation
  ADAS – the vehicle can control both steering and accelerating/decelerating. However, the driver can still take control of the car at any time. Tesla *Autopilot* and Cadillac *Super Cruise* systems both qualify as Level 2.

- Level 3 – Conditional Driving Automation
  The gap with the previous is substantial from a technological perspective but negligible from a human perspective because what is added are *just* the

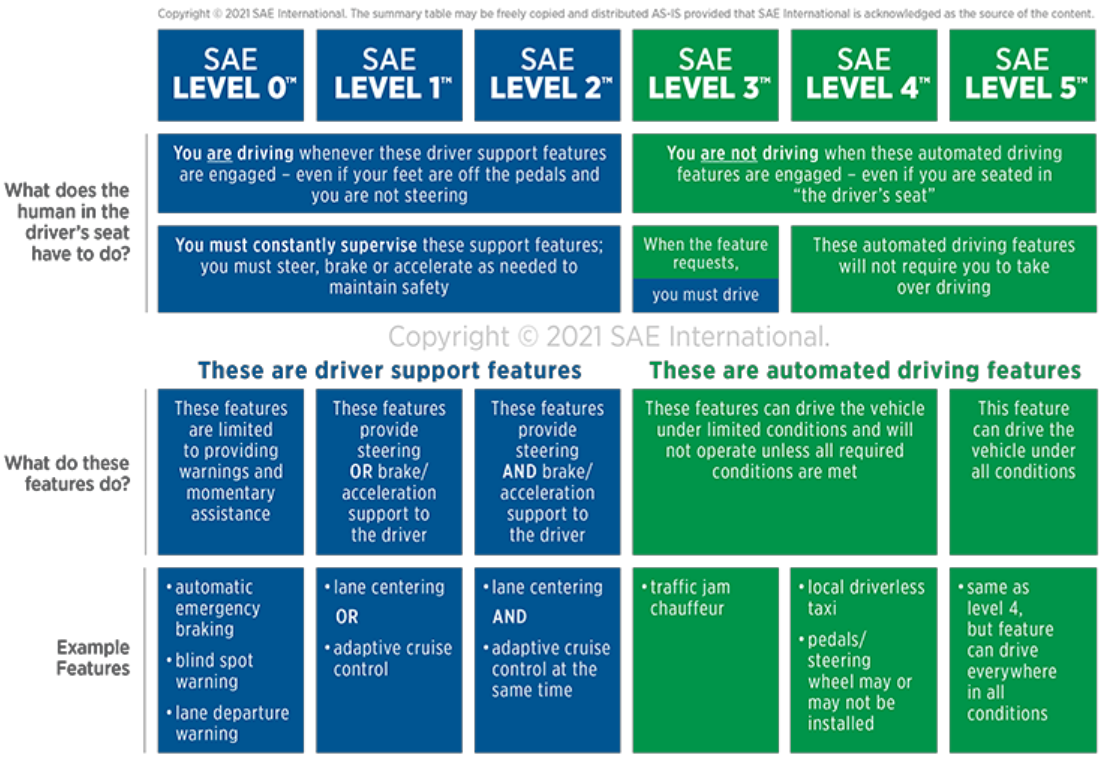


Figure 1.1: SAE J3016 Levels of Driving Automation.

environmental detection capabilities allowing the vehicle to make informed decisions for itself, such as accelerating past a slow-moving car. The driver must yet remain alert and ready to take control if the system is unable to execute the task.

- Level 4 – High Driving Automation
  Vehicles belonging to this class can intervene if things go wrong or in case of system malfunction. Thus, in most circumstances, these cars do not require human interaction. However, the driver still can manually override. Level 4 vehicles can operate in self-driving mode only within limited areas (e.g.: urban areas where top speeds reach an average of 50 km/h). This is known as geofencing: most Level 4 vehicles in existence are geared toward ride-sharing.

- Level 5 – Full Driving Automation
  Vehicles belonging to this class do not require human interaction, this implies that the DDT is eliminated. These cars will not even have steering wheels and pedals (throttle/braking) and are free from geofencing: they can go anywhere and perform as experienced driver does. Fully autonomous cars are undergoing testing in several locations of the world, but none is yet available to the market.

Figure 1.2 [5] provides a graphical representation of the six levels of driving automation where it is possible to see what exactly the driver can do at each level.

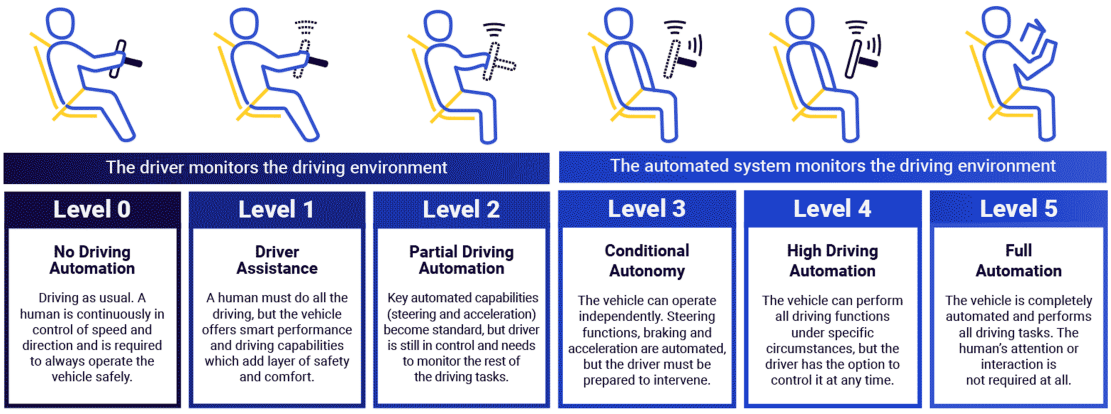


Figure 1.2: Graphical Representation of the Six Levels of Driving Automation.

1.2 Required Sensors in Autonomous Vehicles

Autonomous cars rely on sensors, actuators, machine learning algorithms and much more. As mentioned previously, the three primary functions on which an autonomous system is based are:

- **Perception:** sense the surrounding environment and define its representation.
- **Decision:** decide actions according to the analysis of the environment.
- **Actuation:** compute the action(s).

From these, the general block diagram of an ADAS (Figure 1.3) is derived:

Figure 1.3: ADAS General Block Diagram.

and where:

- **Environment:** it considers either the outer or the inner environment of the vehicle since you cannot perceive the complete environment.

- **Sensor(s):** the environment is perceived by sensors like LiDAR, RADAR, camera.
- **Perception:** it performs the information extraction from data, a sensor data fusion procedure is employed - take the data from multiple sensors and fuse them together.
- **Processing and Decision-making:** it represents the brain of the ADAS function, it takes as input the information coming from the perception block.
- **Actuation:** compute the action(s) like LED blinking, audio tones, braking, accelerating.

All the sensors with which a vehicle is equipped, are helpful in accomplishing the first task, i.e. perception. Indeed, an autonomous car creates and maintain a map of the environment based on the sensors mounted in different parts of its body. Sensors are mainly categorised in:

- Proprioceptive: sensors measuring the state of the car itself (Inertial Measurement Unit (IMU); wheel encoders; etc.).
- Exteroceptive: sensors measuring the state of the environment (LiDARs; RADARs; cameras; ultrasonic; etc.).

As the reader can imagine, the last category is the most important in Autonomous Driving (AD) because these sensors provide a detailed description of the surrounding environment. LiDARs bounce pulses of light off the car surroundings to measure distances, detect road edges and identify lane markings. RADARs monitor the position of nearby vehicles. Cameras detect traffic lights, read road signs, track other vehicles and look for pedestrians. Ultrasonic sensors detect curbs and others vehicle while parking.

Sophisticated software then processes all the sensor inputs, plots a path and sends instructions to the actuators which control acceleration, braking and steering. Hard-coded rules, obstacle avoidance algorithms, predictive modelling and object recognition help the software follow traffic rules and navigate obstacles. Figure 1.4[1] gives a graphical overview on how the different sensors in a vehicle are used for the different types of Advanced Driver Assistance System.

[1]**Figure credits:** https://www.insightsonindia.com/2016/11/30/insights-issues-autonomous-vehicle-technology/.

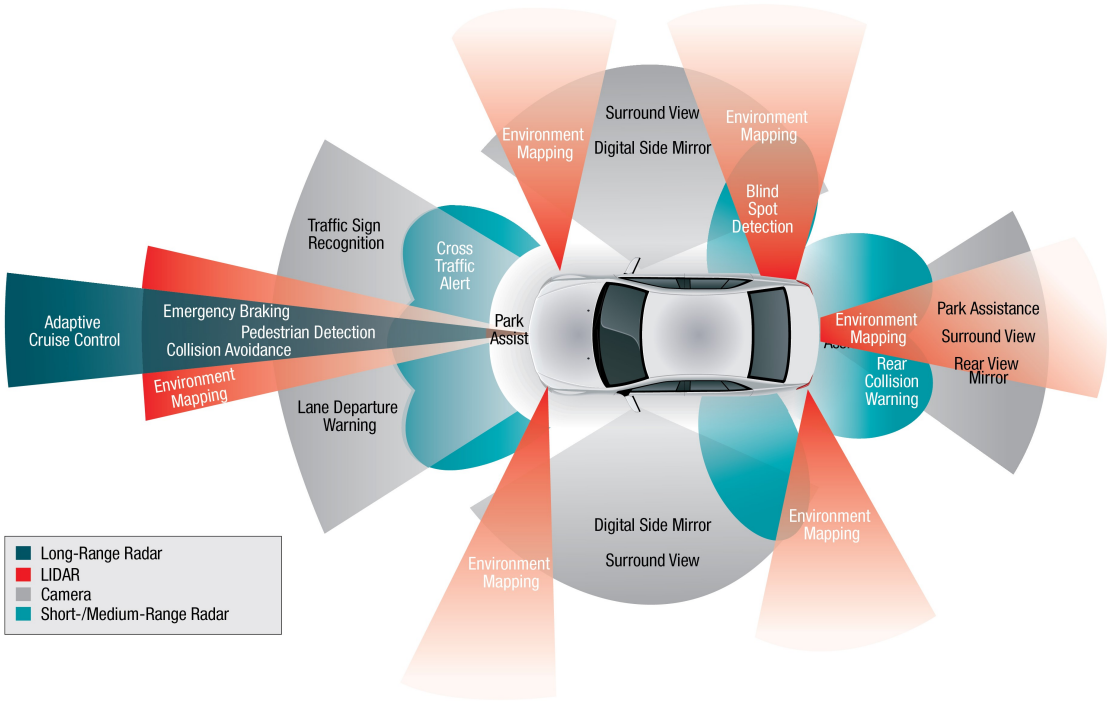


Figure 1.4: ADAS and Related Sensors.

1.3 Benefits of Autonomous Vehicles

All over the world, there are many people who spend a great amount of their time driving and despite they want to do so quick and safe, in most countries the one of the highest cause of death is due to car accidents. According to the WHO (World Health Organization) [6], every year the lives of approximately 1.3 million people are cut short as a result of a road traffic crash (in EU, in 2017 alone, 25300 people died on the Union's road [7]) and between 20 and 50 million more people suffer non-fatal injuries, with many incurring a disability.

Road traffic injuries cause considerable economic losses to individuals, their families and to nations as a whole. Road traffic crashes cost most countries 3% of their gross domestic product. Plus, in recent years it came out the environmental problem. For this reason, scenarios for convenience and quality-of-life improvements are limitless e.g.: elderly and physically disabled would have independence and that is why in last years a lot of car companies around the world put (almost) all their efforts in developing the ADAS and self-driving vehicles.

However, the real promise of autonomous cars is the potential for dramatically

lowering CO_2 emissions. Three trends that, if adopted concurrently, would unleash the full potential of autonomous cars: vehicle automation, vehicle electrification and ride-sharing. By 2050, these three could:

- Reduce traffic congestion (30% fewer vehicles on the road).
- Cut transportation costs by 40% (in terms of vehicles, fuel and infrastructure).
- Improve walkability and liveability.
- Free up parking lots for other uses (schools, parks, community centres).
- Reduce urban CO_2 emissions by 80% worldwide [2].

Instead, the role of ADAS is to minimize and even cancel human errors (involved in about 95% of all road traffic accidents in EU [8]) during driving which are the main cause of vehicle accidents. Essential safety critical ADAS applications include:

- Pedestrian detection/avoidance.
- Lane departure warning/correction.
- Traffic sign recognition.
- Automatic emergency braking.
- Blind spot detection.

These lifesaving systems are key to the success of ADAS applications. They incorporate the latest interface standards and run multiple vision-based algorithms to support real-time multimedia, vision co-processing and sensor fusion subsystems [9].

The modernization of ADAS applications is the first step toward realizing autonomous vehicles. According to the ERSO (European Road Safety Observatory) [10], ADAS can be defined as: "*vehicle-based intelligent safety systems which could improve road safety in terms of crash avoidance, crash severity mitigation and protection and post-crash phases. ADAS can, indeed, be defined as integrated in-vehicle or infrastructure based on systems which contribute to more than one of these crash-phases*".

Automobiles are the foundation of the next generation of mobile-connected devices, with rapid advances being made in autonomous vehicles. Autonomous application solutions are partitioned into various chips: SoCs (System on Chips), which connect sensors to actuators through interfaces and high-performance ECUs (Electronic Control Units). Self-driving cars use a variety of these applications

and technologies to gain 360-degree vision meaning that hardware designs are using more advanced process nodes to meet ever-higher performance targets while simultaneously reducing demands on power and footprint.

ADAS systems actively improve safety with the help of embedded vision by reducing the occurrence of accidents and injury to occupants. The implementation of cameras in the vehicle involves a new AI function that uses sensor fusion to identify and process objects.

However, vehicle control is one of the most critical parts of the autonomous driving vehicle architecture since it is responsible for both safety and comfort of the vehicle.

The challenges of AD in the EU concern the followings [7] (Figure 1.5):

- **Road Safety**: self-driving vehicles will share the road with pedestrians, bicycles and non-automated vehicles, thus adequate safety requirements and traffic rules are essential.
- **Liability Issues**: EU liability laws need to evolve and clarify who is responsible (i.e., the driver or the car company) in case of road traffic accidents since the driving task is shifted from humans to AI.
- **Data Processing**: cybersecurity must be guaranteed and cyberattacks must be prevented.
- **Ethical Question**: human dignity and freedom of choice must be respected by self-driving vehicles. Thus, specific standards for AI are necessary (yet in drafting).
- **Infrastructure**: investments in research and innovation are of vital importance to develop technologies and deploy necessary infrastructure.

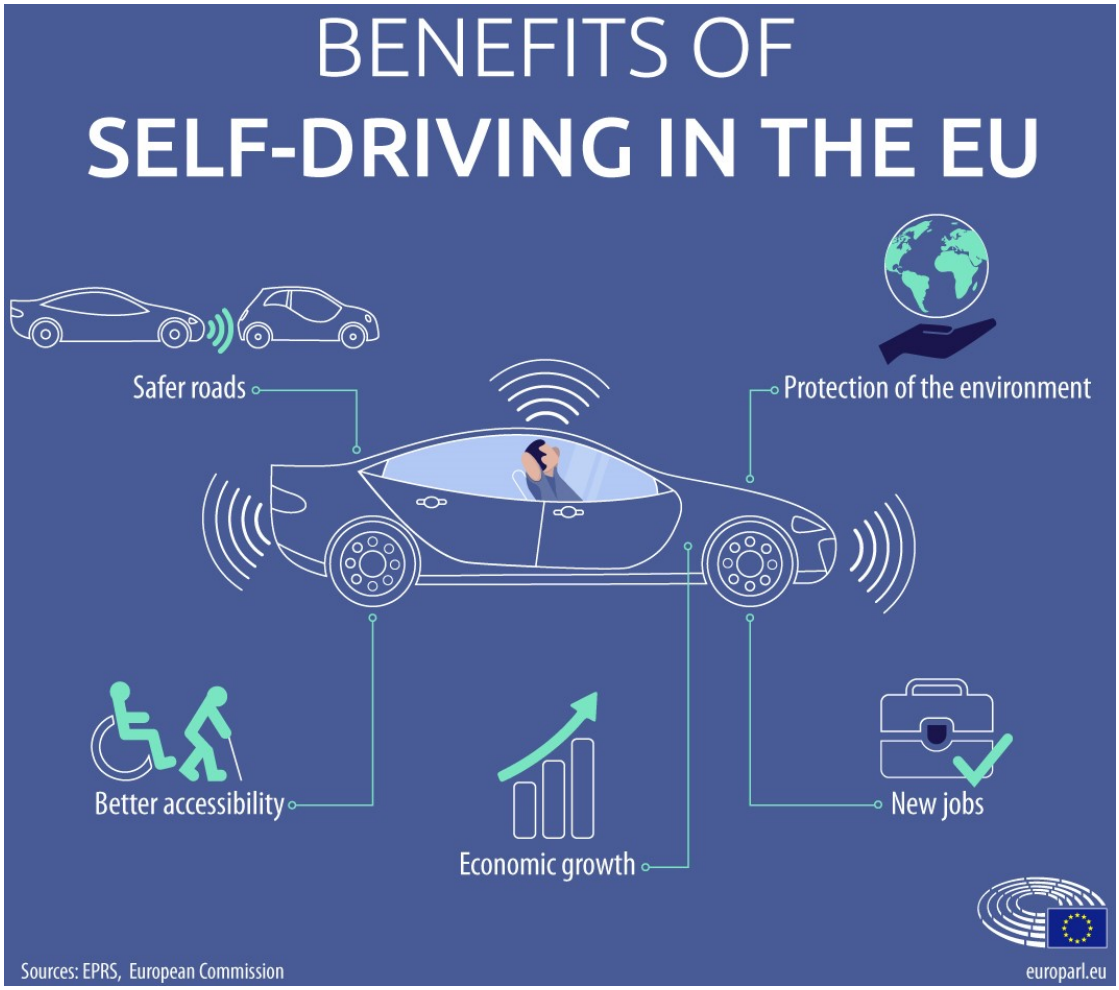


Figure 1.5: Benefits of Self-Driving in the EU.

1.4 Thesis Outline

As expounded previously, the autonomous car has three primary functions to achieve. The perception and planning functions will be analysed in this final project work. In particular, the lane detection and the RRT* algorithm for trajectory computation will be considered. To this purpose, the object of this work is to design a model predictive controller for autonomous driving vehicles in the scenario of a parking manoeuvre.

What will be found later is hereby briefly reported:

- Chapter 2: the BOSCH Future Mobility Challenge is presented starting from

the team selection process to the technical part in which algorithms will be considered.

- Chapter 3: the autonomous parking case study is presented, explaining the functions necessary to manage it.
- Chapter 4: the vehicle model is provided with its mathematical derivation.
- Chapter 5: the model predictive control method is introduced in its general terms explaining its working principle with a brief review of its application in the automotive industry followed by a focus on the computational problem of the Model Predictive Control (MPC) and of the variants it has.
- Chapter 6: the software and hardware implementation are discussed with a brief presentation of the engine used to realize the scenario.
- Chapter 7: the results and conclusions are given, retracting the aforementioned steps of the implementation.

Chapter 2

The Bosch Future Mobility Challenge 2022

This work has been realized based on the effort and the assignments performed during the participation to the Bosch Future Mobility Challenge (BFMC), an international autonomous driving and connectivity competition for Bachelor and Master students [11] organized by the *Bosch Engineering Centre Cluj* since 2017. Students team from all over the world are invited every year to develop autonomous driving and connectivity algorithms on 1 : 10 scaled vehicles, provided by the company, to navigate in a designated environment simulating a miniature smart city. The students work on their projects in collaboration with Bosch experts and academic professors for several months to develop the best-performing algorithms.

The author of this work has joined the challenge under the team *PoliTron* composed by 4 other colleagues from the master's degree program in Mechatronic Engineering of the Polytechnic of Turin, with the guidance of the supervisor himself, Professor Stefano Malan.

The job to carry out during the challenge, which lasts from November to May, consists in developing the algorithms involved in the realization of the autonomous car guide and implementing them into the received car, therefore it commits both the software and the hardware parts. All in all, it is a real and complete accomplishment of self-driving car.

2.1 The Competition

The competition requires that, in addition to the activities carried out by the teams to achieve the final objective, participating teams send a monthly periodic

status via the competition website containing the followings to show their progress to the Bosch representatives:

- A **technical report** describing the development in the last sprint.
- A **project plan** alongside with the **project architecture**.
- A **video file** emphasizing with visual aid the contributions from the past month activity (already present in the report and project plan).

During the competition, specifically in the middle of March, there is a preliminary elimination round, the *Mid-Season Quality Gate*. Each participating team is required to submit a video lasting no longer than 3 minutes in which the vehicle completes the following series of tasks in a single autonomous run:

1. Lane keeping.
2. Intersection crossing.
3. Complete manoeuvre after the following signs:
   - 3.1. *Stop* sign – the car must stop for at least 3 s.
   - 3.2. *Crosswalk* sign - the car must visibly reduce the speed and if a pedestrian is crossing, the car must stop.
   - 3.3. *Priority Road* sign - act normal, you are on a priority road and any vehicle coming from a non-priority road should stop.
   - 3.4. *Parking* sign - it is found before and after a parking lot and, if empty, can be used to perform the parking manoeuvre.

These tasks can be demonstrated by means of one of three possible alternatives:

- A video of the car performing the actions on a real-life like map.
- A video of the car in front of a Desktop, taking a video as a simulated input and acting accordingly.
- A video of the car in front of a Desktop where the simulator is running, taking as visual input the one from the camera inside the simulator.

The author's team has chosen the first option, realizing physically the track shown in Figure 2.1.

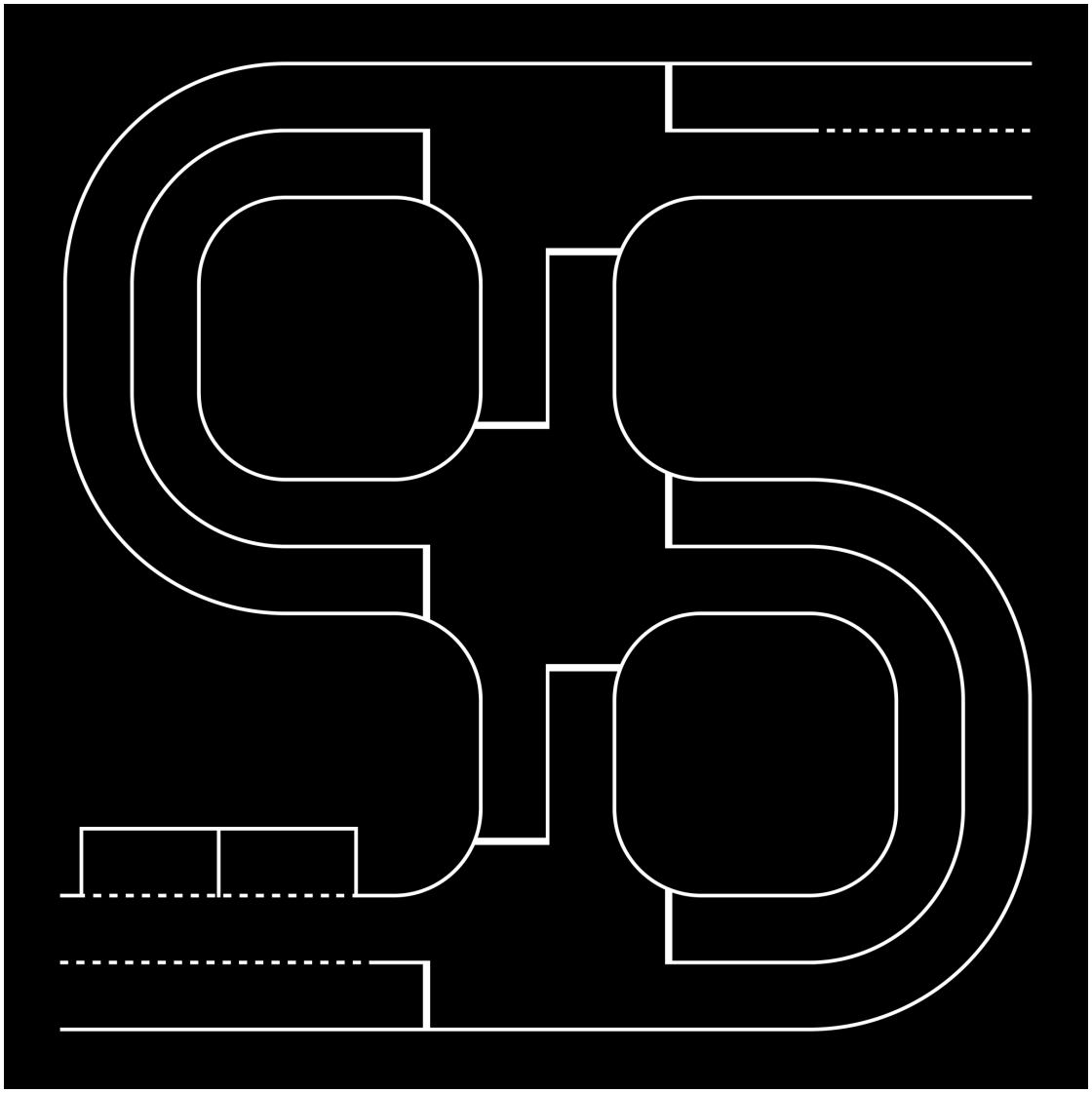


Figure 2.1: Test Track.

Based on the videos, the jury will decide which teams possess the right skills to continue the competition and to go to the Bosch Engineering Centre site in Cluj-Napoca (Romania) for the final round: qualifications, semifinals and finals.

During the race period in Romania, the teams will have to face two challenges: the technical and the speed one. The former requests that the car can correctly respect most of the road signs, such as traffic signs, traffic lights, lanes, intersections, ramps and roundabouts. Moreover, it must detect pedestrians and overtake other cars present in the same lane. The latter asks the car to complete a determined

path in the shortest time possible, this time respecting only the lanes and the road markings. In addition to this, the teams will make a presentation in front of the jury.

Only a maximum of 8 teams will be selected to participate to the final race, in which the first 3 qualified teams will win both a money prize and the car kit and another team, not included in the top 3, will be rewarded as the "best newcomer", meaning a team which did not take part to the competition in the previous year. All the phases of the challenge are reported in Figure 2.2.

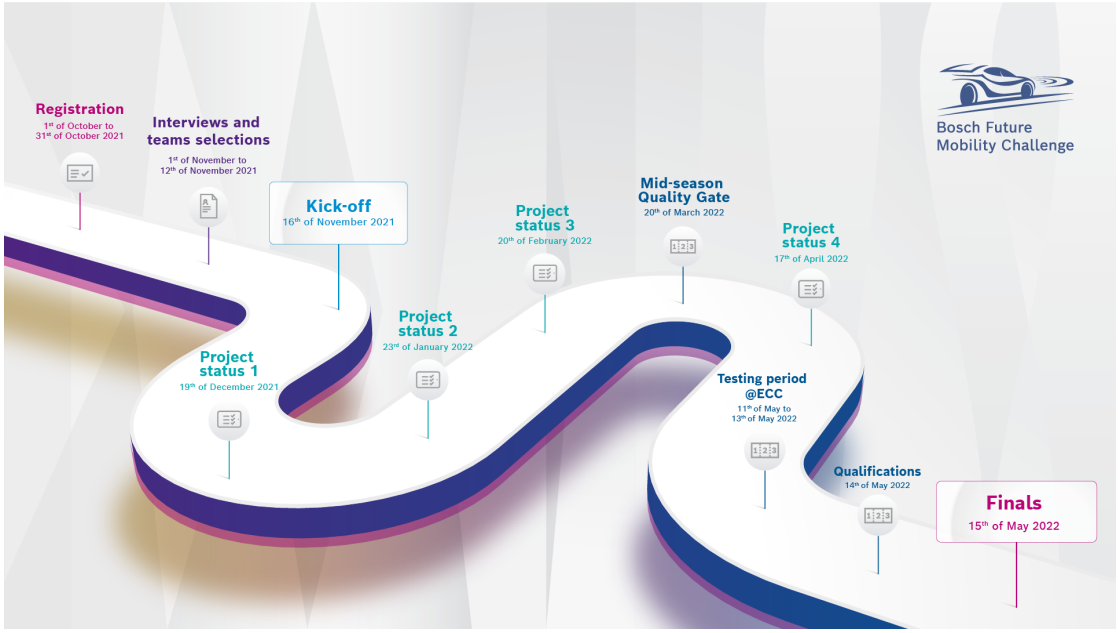


Figure 2.2: Timeline.

The author's team managed to reach the finals and competed with other 7 talented teams from Greece, Romania, Portugal and Italy and won the Best new participating team award (Figure 2.3).

Figure 2.3: Best New Participating Team Award.

2.2 The Car-Kit

Going into the details of the car kit provided by Bosch, the following components are found and highlighted in Figure 2.4 :

- Nucleo F401RE.
- Raspberry Pi 4 Model b.
- VNH5012 H-bridge Motor Driver.
- ATM103 Encoder.
- DC/DC converters.
- Servomotor.
- LiPo Battery.

- Chassis.
- Camera.
- IMU Sensor.

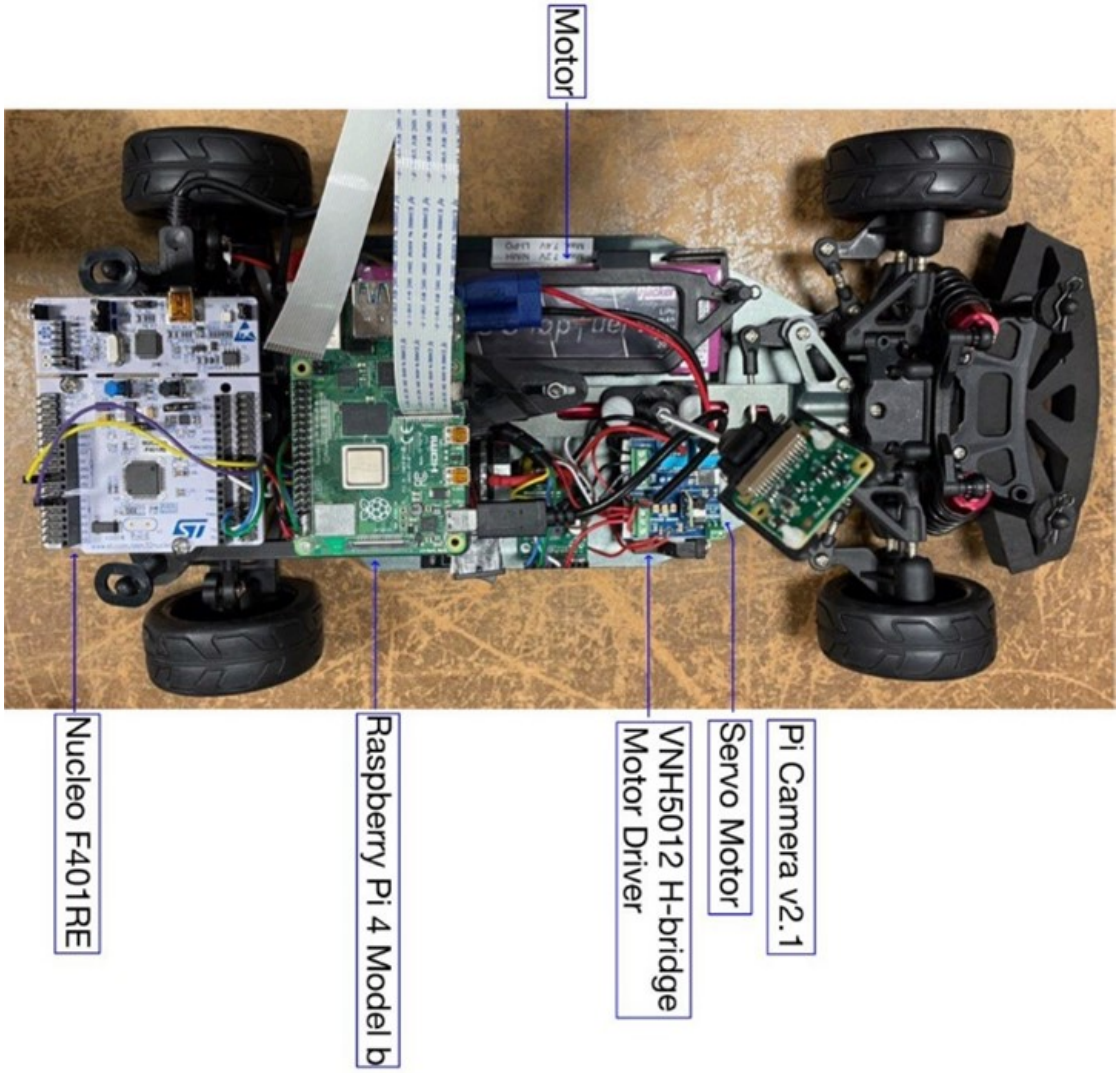


Figure 2.4: The Car-Kit.

In addition to these basic elements, the team decided to furnish the car with a LiDAR sensor and an ultrasonic sensor, placed respectively in the front and the right-hand side of the car.

2.3 The Project

To start working on the project, the teams are provided with a complete documentation necessary to better understand the project structure, especially the hardware side and the Python/C++ codes for the correct communication of all the car components. The documentation is subdivided as in Figure 2.5 and a brief explanation of the content of each section is provided in the following subsections.

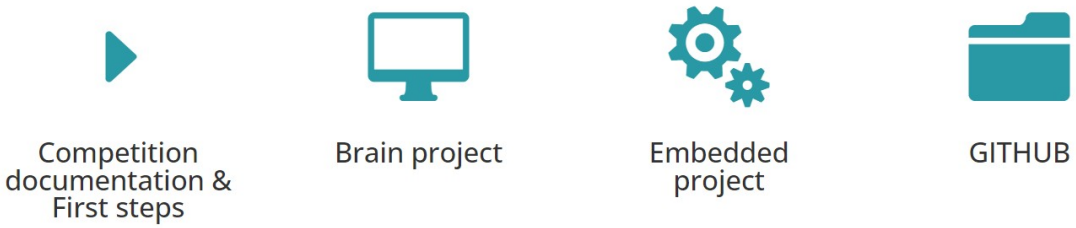


Figure 2.5: Website Layout of the Shared Documentation.

2.3.1 Competition Documentation and First Steps

It includes:

- Connection diagram and description with links to the car components.
- Racetrack: the description of the provided racetrack and its elements, the given components and the diagrams, as well as a starting point and directions of the knowledge required.
- V2X (Vehicle-to-Everything): it includes localization, semaphore, environmental server and vehicle-to-vehicle communication.
- Printed components and circuit boards.
- Hardware improvements: it includes settings for the hardware components.
- Useful links for Raspberry Pi, ROS and Python.
- Periodic status: project plan and architecture, reports and media.

2.3.2 The Brain Project

The Brain Project describes the given code for the RPi platform. It includes the start-up code and the documentation for the provided API (Application Programming Interface), which will help the V2X communication. The project uses concepts of multi-processing and distributed system and it implements a basic flexible structure, which can be extended with new features. This folder contains:

- Introduction: concept and architectures, in particular remote car control and camera streaming, installation and configuration, IMU displayer.
- Utils layer: camera streamer, remote control.
- Hardware layer: camera, serial handler process and camera spoofer process.
- Data acquisition layer: traffic lights, localization system, environmental server.

The computer project is already implemented on the provided RPi, while the embedded project is already implemented on the Nucleo board. Together, they give a good starting point for the project, providing a remote keyboard control, remote camera stream, constant speed control of the given kit and others.

2.3.3 The Embedded Project

This documentation outlines the low-level application that operates on the Nucleo-F401RE microcontroller. Its purpose is to manage the car movement and act as a link between the higher-level controllers and lower-level actuators and sensors. This section is divided into four distinct parts:

- Tools for development containing the instructions to upload the codes related to the correct functioning of the Nucleo.
- Brain layer contains the state machine of the Nucleo (speed and steering).
- Hardware package includes the drivers for actuator and sensors.
- Signal, utils and periodics namespace: 'signal' includes libraries for processing signals, 'utils' package incorporates some util functionalities and 'periodics' layer includes some periodic tasks.

2.3.4 The GitHub Repository

Bosch provided their own link of GitHub in which all the Python/C++ codes related to the topics described above are held. Specifically:

- `Brain` and `Brain_ROS`: the project includes the software already present on the development board (Raspberry Pi) for controlling the car remotely, use the API and test the simulated servers, respectively for Raspbian and ROS.
- `Startup_C`: the project includes some of the scripts transcribed in C++ language from the startup project.
- `Embedded_Platform`: the project includes the software already present on the Embedded platform (Nucleo board). It describes all the low-level software for controlling the speed and steering of the car.
- `Simulator`: the project includes the software for the Gazebo simulator, which is the official on-line environment of the competition.
- `Documentation`: the project includes all the general documentation of the competition environment, guides, diagrams, car components, etc.

2.4 The Structure behind the Algorithms

The tasks to perform by the end of the competition are the following:

- Lane Keeping and Following.
- Intersection Detection and crossing.
- Correct maneuvers under the detection of the following traffic signs: stop, priority, crosswalk, parking, roundabout, highway entrance and highway exit, one-way, no entry.
- Parallel and perpendicular parking.
- Object Detection: pedestrian and overtake over a static and/or moving vehicle.
- Navigation by means of nodes and localization system (GPS).

The brain of the car must be inserted in the Raspberry Pi which, basing on the tasks to perform, sends the commands to the Nucleo which, in turn, acts on the motor and on the servo motor to regulate both the speed and steer. More in details, in order to process the image, the Raspberry takes as input the camera frame and the IMU data for the position of the vehicle, runs the specific control algorithms and sends the corresponding output commands to the Nucleo; for example, an increased speed in presence of a ramp which signs the entrance to the highway, a decreased speed and a specific steer when traveling along a tight curve and a zero

speed when the traffic light turns red.

The correlation between all project components, sensors, algorithms and vehicle actuation is represented in the project architecture (Figure 2.6).

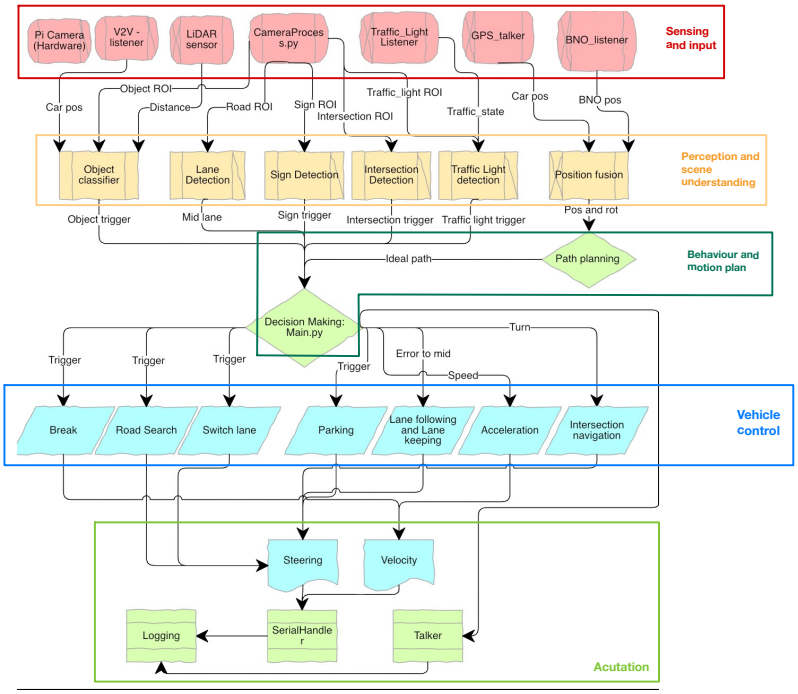


Figure 2.6: Team PoliTron's Complete Project Architecture.

The project presented in this chapter sinks the roots for the work developed by three members of team PoliTron: specifically, Cristina Chicca deals with the image processing part, Gianmarco Picariello's work consists in the development of an MPC controller for the autonomous parking and Claudia Viglietti's thesis concerns optimization algorithms for path planning.

Chapter 3

Autonomous Parking

The history of autonomous parking can be traced back to the early 2000s, when the first prototypes and concepts were developed. However, it was not until the 2010s that autonomous parking technology became more advanced and started to be integrated into commercial vehicles. Companies such as Tesla and BMW were among the first to offer semi-autonomous parking features, such as automated parallel and perpendicular parking, in their vehicles. In recent years, fully autonomous parking systems have become increasingly common in luxury (like Audi, Figure 3.1[1]) and high-end vehicles and the technology continues to improve and evolve.

[1]Parking system plus informs the driver, visually and audibly, about obstacles in front of and behind the vehicle (`https://www.audi-mediacenter.com/en/photos/detail/parking-system-plus-42333`).

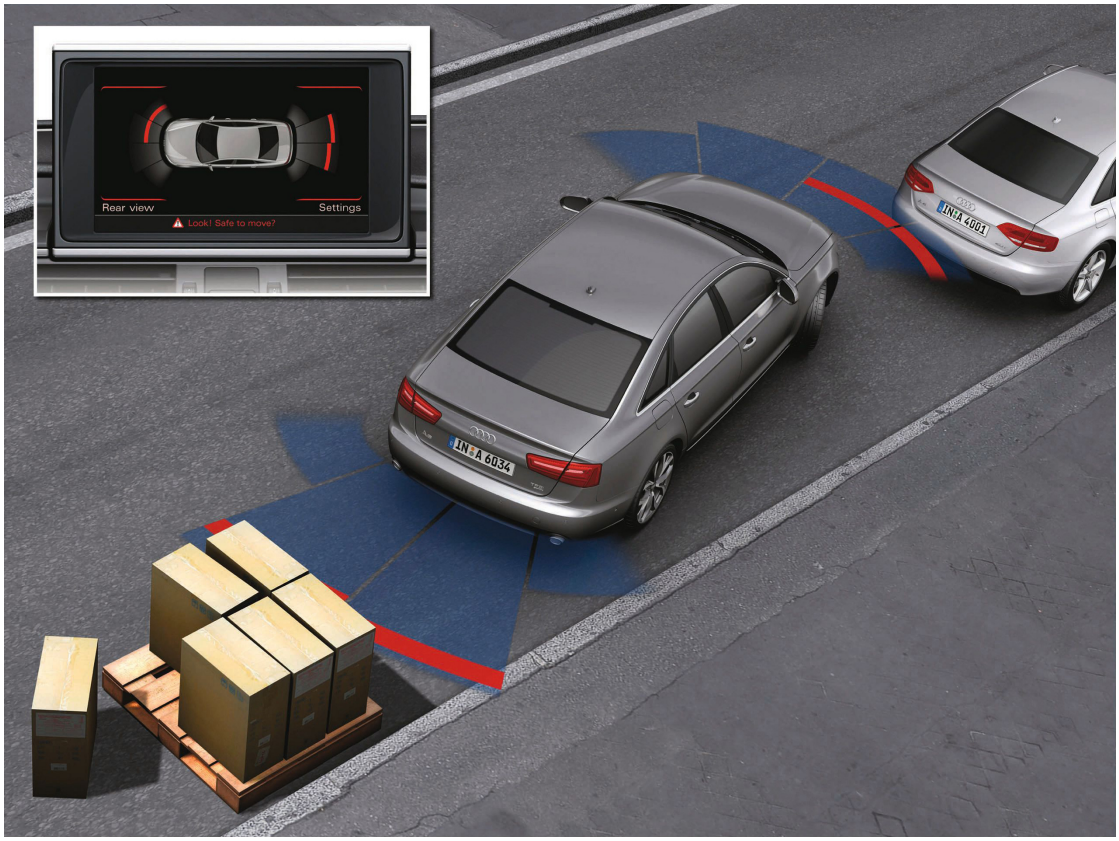


Figure 3.1: Audi Parking System Plus.

3.1 State of the Art

The PA (Parking Assist) is an ADAS that helps the driver parking his/her own car with great precision using guidance systems technology, thus reducing the stress associated with complex parking manoeuvres and minimizing the chance of potential scratches and/or dings that could happen while manually steering in tight spaces. Mostly ultrasonic sensors are used for this system, but nowadays many companies are equipping vehicles with cameras as well that mostly help in the searching of an adequate parking spot. However, it is common that the aforementioned are accompanied by a combination of the following sensors:

- LiDAR (Light imaging, Detection, and Ranging) sensors.
- RADAR (Radio Detection and Ranging) sensors.
- GPS (Global Positioning System) sensors.
- IMU (Inertial Measurement Unit) sensors.

When the PA system is active, it informs the driver as soon as a suitable parking space is found and then, it calculates the best way of approaching the gap, included the number of steering manoeuvres.

By recalling the general block diagram for ADAS (Fig. 1.3), it is possible to derive the block diagram for Parking Assist (Figure 3.2):

Figure 3.2: PA General Block Diagram.

Two types of autonomous parking system are available:

- *Semi-autonomous* PA: the driver controls the parking process by carefully accelerating and braking. The system performs all the necessary steering actions with the help of the power steering. The driver can stop the manoeuvre at any time.
- *Fully autonomous* PA: besides the steering action, now the system takes over also the speed control, i.e. acceleration and braking, driving the vehicle in and out of the chosen spot. Particularly advanced systems (SAE Level 5, p. 4) let the driver deciding whether he/she wishes to remain in the car whilst parking or get out beforehand.

This thesis work, keeping faith to the aim of the Bosch Future Mobility Challenge, will treat the second type of autonomous parking that will consider the *perpendicular parking* case and it will be developed in MATLAB®/Simulink® using a NMPC (Nonlinear Model Predictive Control) controller in three different scenarios:

1. Perpendicular Parking using Nonlinear Model Predictive Control (Sec. 6.1).
2. Perpendicular Parking using Multistage NMPC (Sec. 6.2).
3. Perpendicular Parking using RRT* Planner and NMPC Tracking Controller (Sec. 6.3).

The choice of the *perpendicular* parking scenario is dictated by two factors:

- For the BFMC a basic autonomous *parallel* parking system has been developed.
- In The MathWorks, Inc. documentation those three cases have been already developed for the parallel case:

- Parallel Parking using Nonlinear Model Predictive Control [12].
- Plan Parallel Parking Path using Multistage NMPC [13].
- Parallel Parking using RRT Planner and MPC Tracking Controller [14].

3.2 Autonomous Parking @ BFMC22

In the Challenge context, the PA system algorithm is written using the Python programming language and developed using state machines.

Since the car, after team's modifications, is mounting a LiDAR sensor on the front (TF-Luna, Subsection 3.2.1), an ultrasonic sensor (HC-SR04, Subsection 3.2.2) on the right-hand side, a camera (RPi Camera V2[2]) and a IMU sensor (Smart IMU Sensor - BNO055[3]), the primordial idea was to take advantage of those sensors to develop the algorithm so that the car could have detected the parking symbols with the camera, trigger the ultrasonic sensor to detect an empty spot and use the IMU to handle the care during the manoeuvre. The LiDAR is used to help in parking the car in case of front obstacles.

With reference to the code provided in Appendix A, the autonomous parking algorithm is realized making use of the process `SignDetectionProcess`, part of `MovCarProcess`, which exploits the camera to detect the parking sign and consequently it sets the flag value to 1, `flag = 1`. Thus, at this time the ultrasonic sensor is triggered and starts searching for an empty spot. If this is found, the parking manoeuvre is initiated.

Counters are used to time the different phases of the manoeuvre and taking as example the case of a parallel parking with the first slot empty (Figure 3.3), the

[2]The Raspberry Pi Camera V2 is a high quality 8 megapixel Sony IMX219 image sensor custom designed add-on board for the RPi, featuring a fixed focus lens. It is capable of 1080p video and still images. It is ideal for various projects like home security systems, wildlife cameras, time-lapse photography, etc. It includes automatic control of image quality and level of detail, automatic white balance, automatic gain control and back-light compensation. The camera is compatible with all RPi models that have the 40-pin GPIO header and it connects to the RPi via a ribbon cable and can be controlled using the Raspberry Pi's CSI (Camera Serial Interface) bus.

[3]The smart sensor BNO055 is a SiP (System in Package) solution that integrates a tri-axial 14-bit accelerometer, an accurate closed-loop tri-axial 16-bit gyroscope, a tri-axial geomagnetic sensor and a 32-bit micro-controller running the BSX3.0 FusionLib software. This smart sensor is significantly smaller than comparable solutions. By integrating sensors and sensor fusion in a single device, the BNO055 makes integration easy, avoids complex multi-vendor solutions and thus simplifies innovations [15].

stages are the following:

1. The ego vehicle places its middle lateral axis perpendicular to the last line delimiting the parking lot.
2. The ego vehicle steers the wheels and initiates the backward motion.
3. The ego vehicle steers again the wheels to have them straight and terminate the manoeuvre.
4. If the ego vehicle is not close to a front car, it proceeds straight till reaching the parking spot center. If/When it is too close, the LiDAR detects the obstacle and it immediately stops the ego vehicle motion.

then, the car has to exit from the parking spot and thus reverse the steps from 3 back to 1.

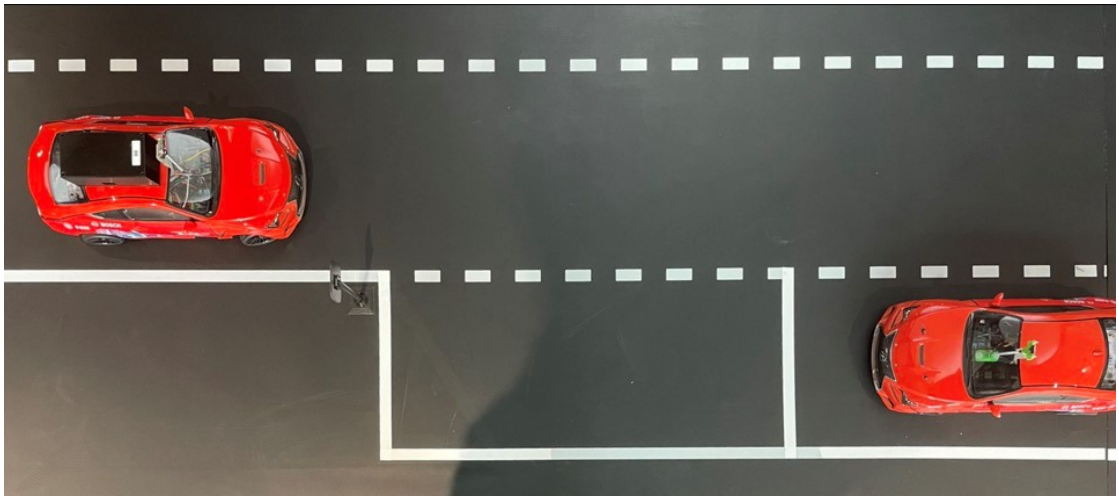


Figure 3.3: Parallel Parking Scenario @ BFMC22.

When the parking spot is occupied, the car proceeds its straight motion until the ultrasonic detects an empty space. However, the reader has to bear in mind that the algorithm is not for general use because manoeuvres are defined for the parking lot of the Competition Track (Figure 3.4[4]) and only the autonomous parallel parking manoeuvre has been developed because according to competition's rules, the drive path (yellow path in Fig. 3.4) could have been arbitrarily chosen and thus, following the planned trajectory, the parallel parking was the best fit.

[4]Nodes are used alongside the GPS system provided by the Bosch infrastructure and are given in a .graphml file format where each node has 3 parameters: ID, an x- and a y-coordinate.

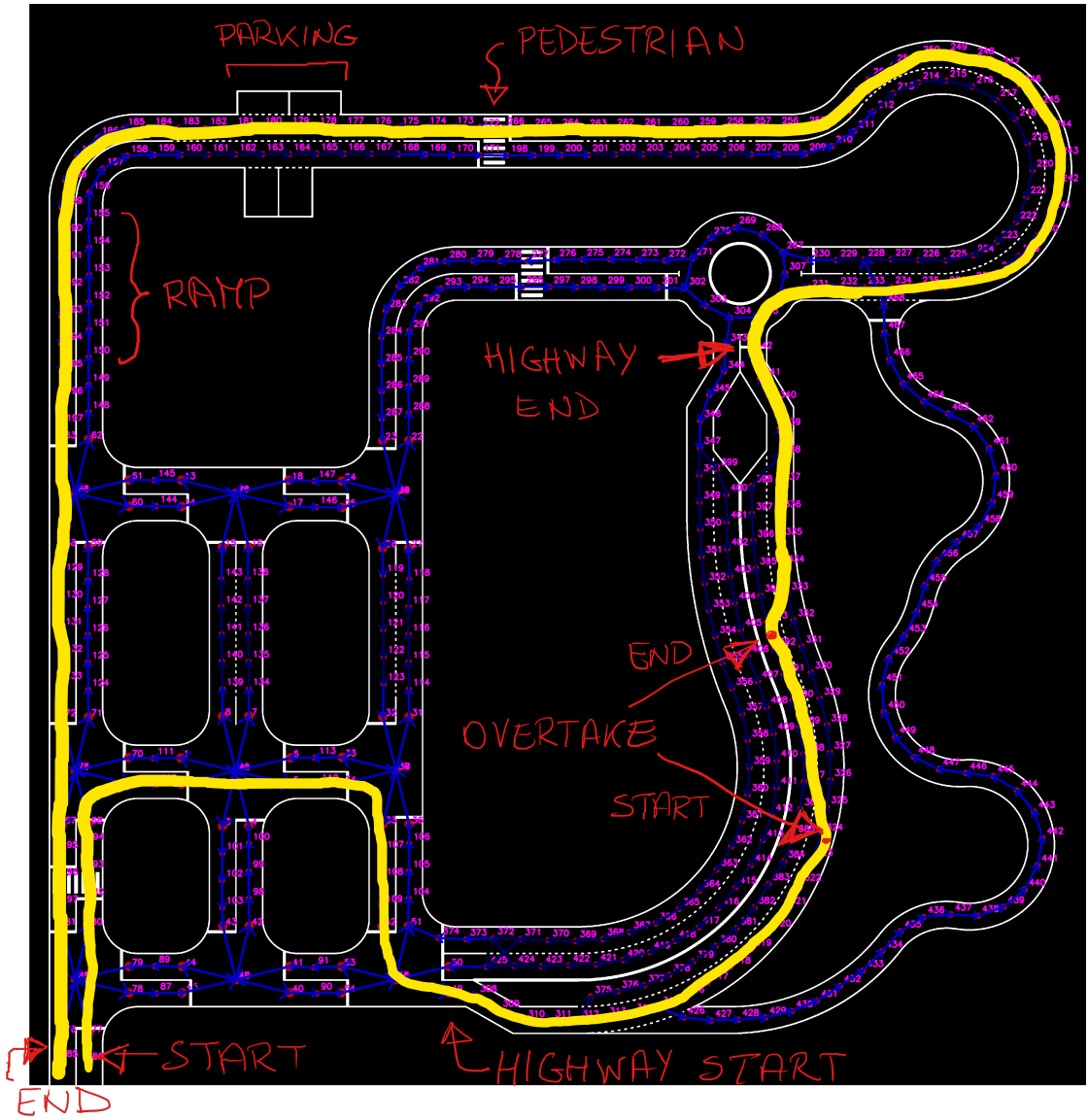


Figure 3.4: BFMC22 Competition Track with Nodes.

3.2.1 The TF-Luna LiDAR Sensor

The TF-Luna LiDAR sensor uses the ToF (Time of Flight) principle[5] to measure the distance by periodic emission of NIR (Near-Infrared) modulated waves. The relative distance D is given by the time obtained from measuring the phase difference $\Delta\varphi$ between original and reflected wave (Equation 3.1, Figure 3.5[6]).

$$D = \frac{c}{2} \cdot \frac{1}{2\pi f} \cdot \Delta\varphi \tag{3.1}$$

where c is the speed of light ($c \approx 3 \times 10^8\ m/s$) and f represents the waves frequency.

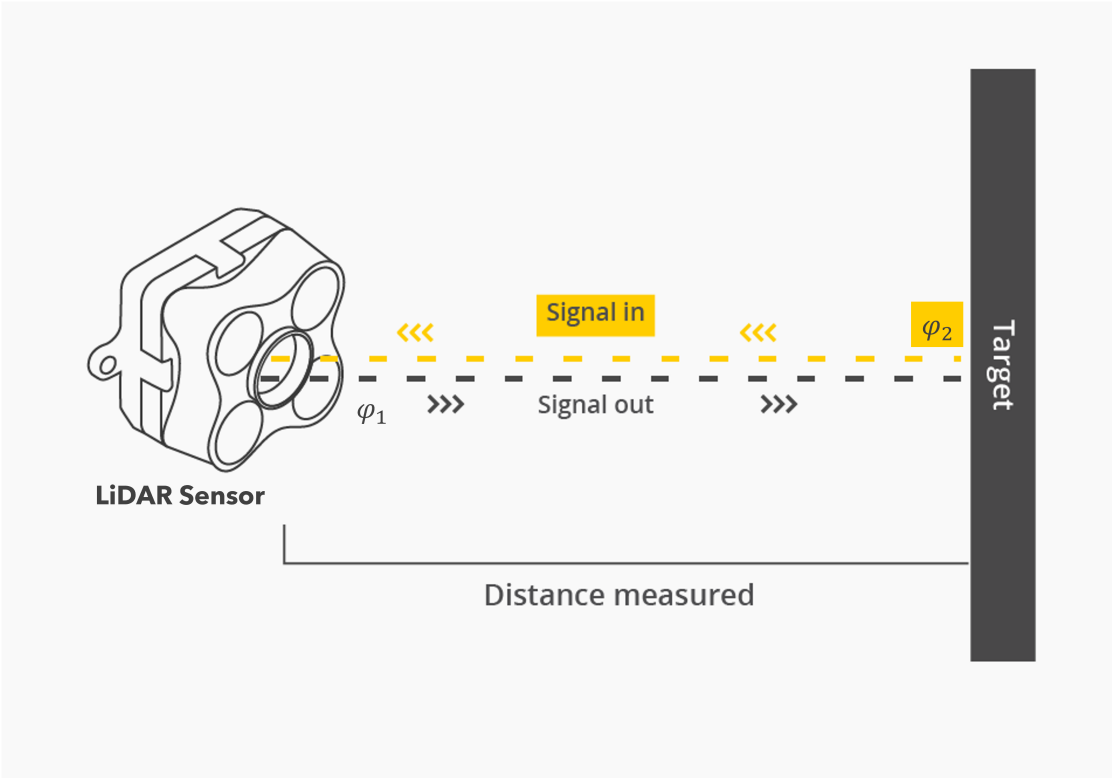


Figure 3.5: Schematics of the ToF Principle.

[5]The ToF principle is used to determine the distance to an object by measuring the time it takes for a light signal or pulse to travel from a device to an object and back. It is based on the speed of light, which is constant and the time it takes for a light signal to travel a certain distance can be calculated using simple math. This information can then be used to compute the distance to an object, allowing for precise depth mapping, 3D imaging and other applications in fields such as robotics, manufacturing and entertainment.

[6]**Figure credits:** `https://www.terabee.com/time-of-flight-principle/`.

Parameters specifications of the LiDAR sensor are tabulated in Table 3.1.

Description	**Parameter Value**
Operating Range	$0.2 \div 0.8\ m$
Accuracy	$\pm 6\ cm$ @ $0.2 \div 3\ m$
	$\pm 2\%$ @ $(3 \div 8)\ m$
Measurement Unit	cm (default)
Range Resolution	$1\ cm$
FoV	$2°$
Frame Rate	$1 \div 250\ Hz$ (adjustable)

Table 3.1: Parameters Specification of the TF-Luna LiDAR Sensor.

3.2.2 The HC-SR04 Ultrasonic Sensor Module

The HC-SR04 is an ultrasonic sensor module able to measure the distance between the module and other objects in proximity using ultrasound waves and with high accuracy. For this reason, it is employed in obstacle avoidance devices; distance measuring devices; automation projects; vehicles parking systems; etc.

The module is made by an ultrasonic transmitter, an ultrasonic receiver, on-board electronics like operational amplifiers and passive components like capacitors and resistors.

Its operating principles is the following: ultrasound waves are transmitted by the transmitter (signal is coming through the TRIG pin, Figure 3.6) and when an object is in front of the sensor, the waves get reflected by the object itself back to the HC-SR04. Thus, these reflected waves are received by the receiver and then output on the ECHO pin as a digital signal with frequency directly proportional to the time required by an ultrasound wave to travel from the sensor to the obstacle and back to the module.

The emitted waves are on a frequency higher that the human hearing range ($f_{emit} > 20\ kHz$) and the sound wave speed, for a wave traveling through air at room temperature ($T_{room} = 20\ °C$), is $c_{sound} \approx 343\ m/s$. Thus, the distance, D between the module and obstacle is evaluated as the product between the sound wave speed and the sound wave travel time, t_{travel}, from the module to the obstacle and back (Equation 3.2):

$$D = c_{sound} \cdot t_{travel} \tag{3.2}$$

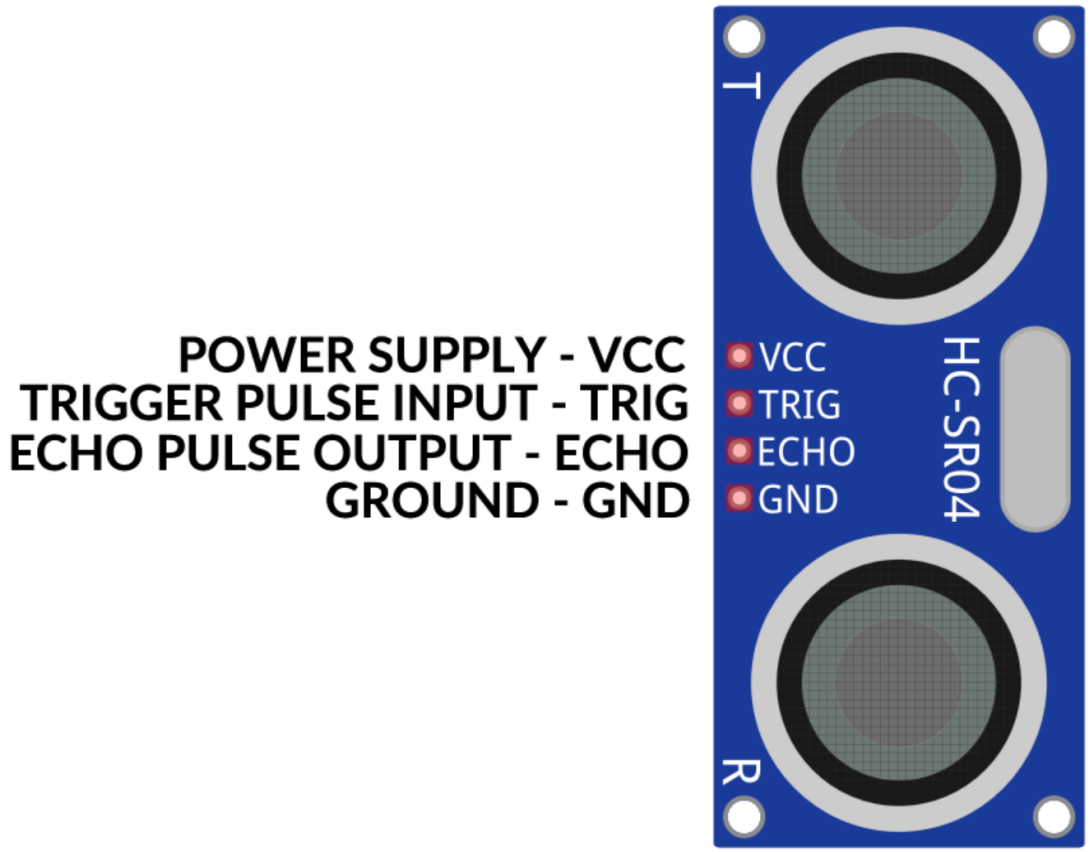


Figure 3.6: Pin-out of the HC-SR04 Ultrasonic Sensor Module.

Moreover, the reader has to bear in mind that since a sound wave travels 343 meters per second, to measure distances from 20 mm (minimum distance for valid measurements) to 4000 mm (sensor range), the time is measured in *microseconds* $[\mu s]$ and consequently, the sound speed should be converted from m/s to $cm/\mu s$ (Equation 3.3):

$$343\, m/s = 0.0343\, cm/\mu s \tag{3.3}$$

Then, to get the actual distance, D_{actual}, between the module and the obstacle, the distance computed in Eqn. (3.2) should be halved because the ultrasound wave travels from the module to the obstacle and back (Equation 3.4):

$$D_{actual}\,[cm] = \frac{c_{sound}\,[cm/\mu s] \cdot t_{travel}\,[\mu s]}{2} \tag{3.4}$$

Parameters specifications of the ultrasonic sensor are reported in Table 3.2.

Description	**Parameter Value**
Power Supply Voltage	up to 5 V
Operating Voltage	$3 \div 5$ V
Output Voltage	5 V
Current Consumption	15 mA
Quiescent Current	< 2 mA
Ultrasonic Frequency	40 kHz
Trigger Input Signal	10 μs TTL pulse
Measuring Angle	30 $^\circ$
Effectual Angle	< 15 $^\circ$
Operating Distance Range	$20 \div 4000$ mm
Claimed Precision	3 mm
Dimensions	$45 \times 20 \times 15$ mm

Table 3.2: Parameters Specification of the HC-SR04 Ultrasonic Sensor Module.

Chapter 4

Vehicle Model Derivation

The topics discussed in this chapter are referred to Chapter 2, *Lateral Vehicle Dynamics* and Chapter 4, *Longitudinal Vehicle Dynamics*, from the book *Vehicle Dynamics and Control* [16].

4.1 Vehicle Longitudinal Model

The control of longitudinal vehicle motion has been pursued at many different levels by researchers and automotive manufacturers. Today's available systems that involve longitudinal vehicle control are: ABS (Anti-Lock Brake System), CC (Cruise Control), TCS (Traction Control System), etc.

In this section the dynamic models for the vehicle longitudinal motion will be presented and the two major elements of the longitudinal vehicle model are: *vehicle dynamics* (Subsection 4.1.1) and *powertrain*[1] *dynamics* (Subsection 4.1.2), where the former is influenced by the following forces:

- Longitudinal tyre forces at the front and rear tyres, F_{xf} and F_{xr}.

[1]The difference between driveline, powertrain and drivetrain is as follows:

- **Driveline** transmits power from the engine to the wheels: drive shaft, differential, axles and half-shafts.
- **Powertrain** generates power and transmits it to the wheels: engine, transmission, driveline and any other components that delivers power to the wheels.
- **Drivetrain** often used interchangeably with powertrain. Drivetrain transfers power from the engine to the wheels: engine, transmission, driveline, axles and any other components that transfers of power.

- Equivalent longitudinal aerodynamic drag force, F_{aero}.
- Forces due to rolling resistance at the front and rear tyres, R_{xf} and R_{xr}.
- Gravitational force due to road inclination, $mg \cdot \sin\theta$ (m, vehicle mass; g, gravity acceleration; θ road inclination[2]).

while the vehicle longitudinal powertrain system is made up by:

- ICE (Internal Combustion Engine).
- Torque Converter.
- Transmission.
- Wheels.

4.1.1 Longitudinal Vehicle Dynamics

The longitudinal vehicle dynamic model is simply based on the dynamics of the vehicle that generates forward motion. By considering a classical case of a vehicle longitudinal motion (i.e., along the x axis) on an inclined road (Figure 4.1[3]) and

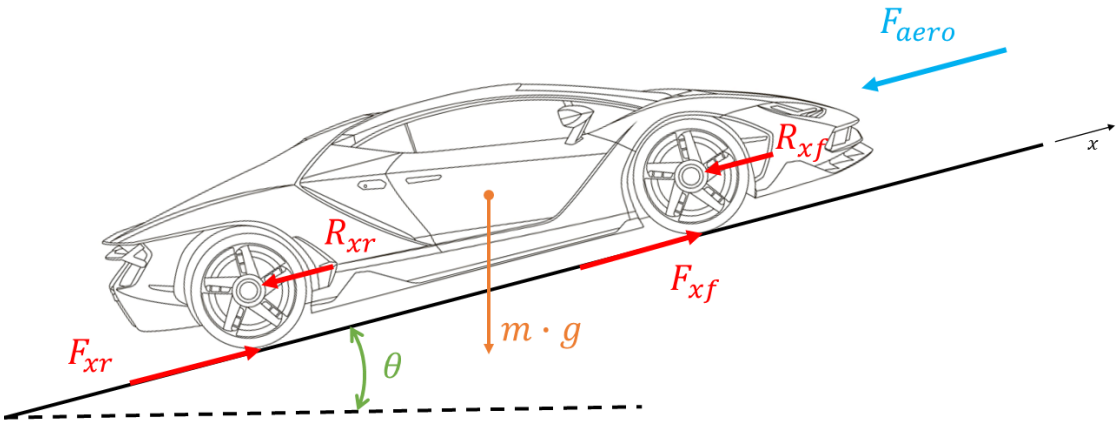


Figure 4.1: Longitudinal Forces acting on a Vehicle driving on an Inclined Road.

based on Newton's Second Law of Motion[4]: the longitudinal tyre forces F_{xf} and

[2]When the longitudinal driving direction x is **leftward**, θ is positive **clockwise**. When the longitudinal driving direction x is **rightward**, θ is positive **counterclockwise**.

[3]**Car image credits:** ID 217689120 © Andrey Vyrypaev | `https://thumbs.dreamstime.com/z/outline-drawing-super-car-view-three-sides-217689120.jpg`.

[4]The change of motion of an object is proportional to the force impressed and is made in the direction of the straight line in which the force is impressed: $F = m \cdot a$.

F_{xr}, which come from the vehicle powertrain, must overcome the resistance forces, i.e. F_{aero}, $mg \cdot \sin\alpha$, R_{xf} and R_{xr}. The imbalance among these forces defines the acceleration of the vehicle in the longitudinal direction denoted by $\ddot{x}$. Therefore, by performing a force balance along the vehicle longitudinal axis, it is possible to obtain the *Full Longitudinal Dynamics* equation (Equation 4.1).

$$m\ddot{x} = F_{xf} + F_{xr} - F_{aero} - R_{xf} - R_{xr} - mg \cdot \sin\theta \tag{4.1}$$

Eqn. (4.1) can be further simplified by grouping the front and rear tyre forces as F_x ($F_x = F_{xf} + F_{xr}$) and the front and rear rolling resistance forces as R_x ($R_x = R_{xf} + R_{xr}$). Moreover, we can assume moderate road inclinations, which means that the small angle approximation can be applied. So, $\sin\theta \approx \theta$. Thus, the *Simplified Longitudinal Dynamics* (Equation 4.2) is:

$$m\ddot{x} = F_x - F_{aero} - R_x - mg \cdot \theta \tag{4.2}$$

where:

- $m\ddot{x}$ represents the inertial term defining the vehicle longitudinal acceleration.
- F_x represents the traction force generated by the powertrain.
- F_{aero}, R_x and $mg \cdot \theta$ make up the total resistance forces acting on the vehicle, F_{load}.

However, the models for each of the forces in Eqn. (4.2) have still to be developed together with the definition of how they will connect to the throttle and break inputs that the autonomous system will apply. Through the rest of this section, these model will be developed.

The longitudinal tyre forces F_{xf} and F_{xr} instead, are friction forces acting on the tyres from the ground and the longitudinal tyre force generated by each tyre depends on:

- Slip ratio, σ_x.
- Normal load on the tyre, N.
- Friction coefficient at the tyre-road interface, C_{σ_f} and C_{σ_r}.

Then, it is possible to consider that a vehicle longitudinal motion is resisted by aerodynamic drag, rolling resistance and the force due to gravity. Thus, the *Total Resistance Load*, F_{load} on the vehicle (Equation 4.3) is:

$$F_{load} = F_{aero} + R_x + mg \cdot \theta \tag{4.3}$$

The equivalent aerodynamic drag force acting on a vehicle (Equation 4.4) can typically be modeled as dependent on air-mass density ρ, vehicle frontal area A_F (projected vehicle area in the driving direction), the aerodynamic drag coefficient (C_d) and the squared sum of the longitudinal vehicle velocity $V_x = \dot{x}$ plus the wind velocity V_{wind} (positive for headwind, negative for tailwind).

$$F_{aero} = \frac{1}{2} \cdot \rho \cdot C_d \cdot A_F \cdot (V_x + V_{wind})^2 \tag{4.4}$$

As tyres rotate, both tyres and road are subjected to deformation at the contact patch. Due to the normal load N, the tyre material is deflected vertically as it goes through the contact patch and it springs back to its original shape as soon as it leaves the contact area. However, due to the internal damping of the tyre material, the energy spent in deforming the tyre is not completely recovered once the tyre recovers its original shape and this loss of energy is represented by the rolling resistance force that opposes the vehicle motion. Rolling resistance is commonly modeled as being roughly proportional to N on each tyre-set (Equation 4.5):

$$R_{xf} + R_{xr} = f(N_f + N_r) \tag{4.5}$$

where:

- f, rolling resistance coefficient.
- N_f and N_r, normal forces at the front and rear tyres.

In addition to the total weight of the vehicle, the computation of the normal forces acting on the tyres (Figure 4.2[5]) involves:

- Fore-aft location of the CoG.
- Longitudinal acceleration of the vehicle, $\ddot{x}$.
- Equivalent longitudinal aerodynamic drag force, F_{aero}.
- Road inclination, θ.

[5]**Car image credits:** ID 217689120 © Andrey Vyrypaev | `https://thumbs.dreamstime.com/z/outline-drawing-super-car-view-three-sides-217689120.jpg`.

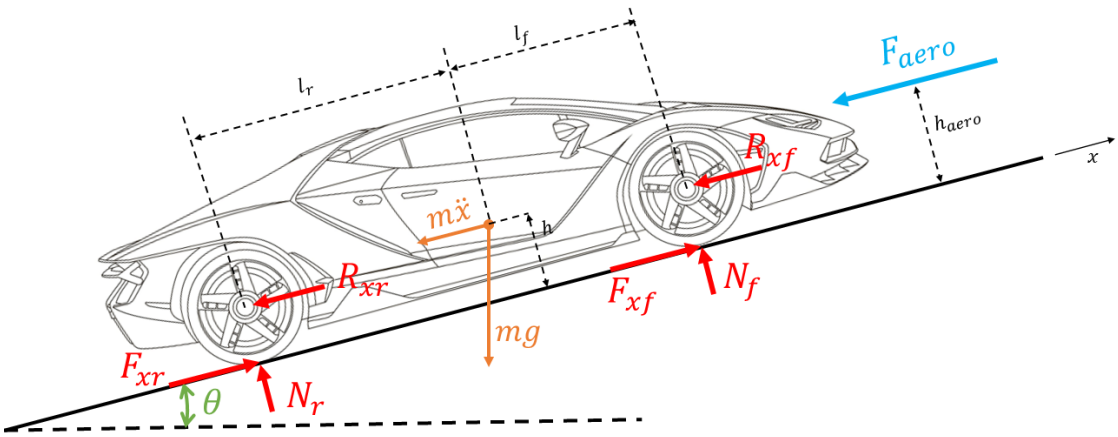


Figure 4.2: Calculation of the Normal Tyre Loads.

The normal load distribution on the tyres can be computed assuming null net pitch torque on the vehicle (i.e., the vehicle pitch angle is assumed to have reached the steady-state value). From Fig. 4.2 the following variables are defined:

- h, height of the vehicle CoG.
- h_{aero}, height at which F_{aero} acts.
- l_f, longitudinal distance of the front axle from the vehicle CoG.
- l_r, longitudinal distance of the rear axle from the vehicle CoG.
- r_{eff}, effective tyre radius.

Now, considering the moments at the contact point of the front and rear tyre in Fig. 4.2, one has (Equations 4.6a and 4.6b):

$$N_r \cdot (l_f + l_r) - F_{aero} \cdot h_{aero} - m\ddot{x} \cdot h - mgh \cdot \sin\theta - mgl_f \cdot \cos\theta = 0 \tag{4.6a}$$

$$N_f \cdot (l_f + l_r) + F_{aero} \cdot h_{aero} + m\ddot{x} \cdot h + mgh \cdot \sin\theta - mgl_r \cdot \cos\theta = 0 \tag{4.6b}$$

and solving equations 4.6a, 4.6b for N_r, N_f, one obtains (Equations 4.7a and 4.7b):

$$N_r = \frac{F_{aero} \cdot h_{aero} + m\ddot{x} \cdot h + mgh \cdot \sin\theta + mgl_f \cdot \cos\theta}{l_f + l_r} \tag{4.7a}$$

$$N_f = \frac{-F_{aero} \cdot h_{aero} - m\ddot{x} \cdot h - mgh \cdot \sin\theta + mgl_r \cdot \cos\theta}{l_f + l_r} \tag{4.7b}$$

from which it is clear that as the vehicle accelerates, the normal load on front tyres (N_f) decreases while the normal load on rear tyres (N_r) increases. The opposite occurs while braking thus helping in distributing the braking forces and maintain stability and traction while braking.

4.1.2 Driveline Dynamics

In Subsec. 4.1.1, Eqn. (4.1) the vehicle longitudinal motion has been derived and where the longitudinal forces on the driving wheels (F_{xf}, F_{xr}) are the primary forces helping the vehicle to move forward and they depend on the difference between vehicle longitudinal velocity, $\dot{x}$, and rotational wheel velocity, $r_{eff} \cdot \omega_w$, with ω_w highly influenced by the driveline dynamics. Figure 4.3[6] shows the major driveline components for the three different types of drive: FWD, RWD, AWD.

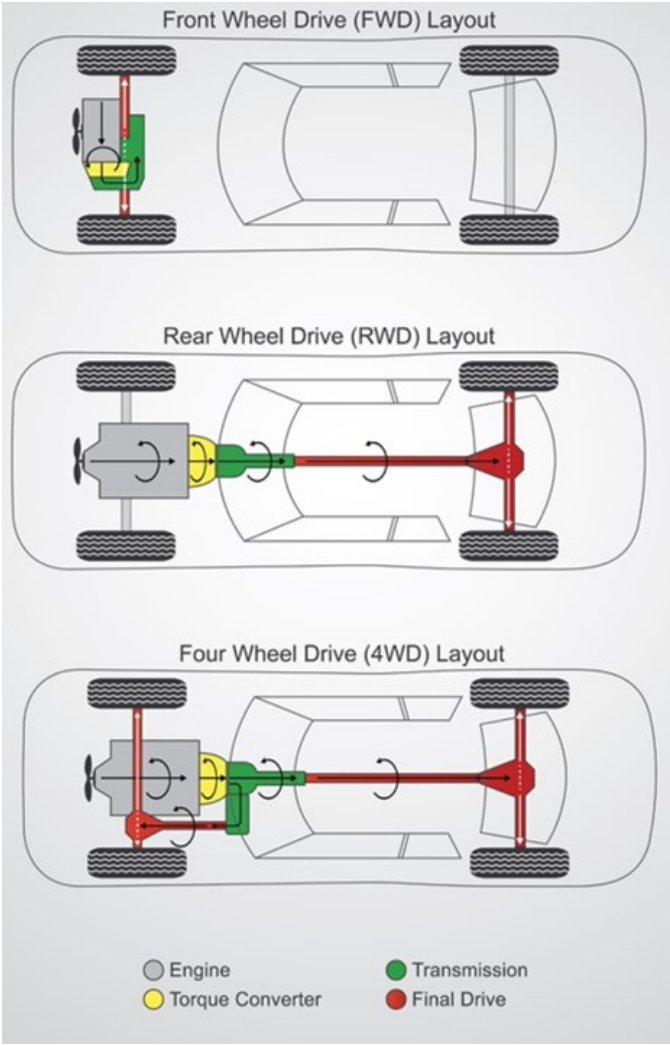


Figure 4.3: Transmission System Layout for the 3 Different Types of Drive.

[6]**Figure credits:** https://www.scienceabc.com/innovation/what-is-a-four-wheel-drive-system-and-how-does-it-work.html.

Figure 4.4 displays the power-flow and the loads-direction on the components.

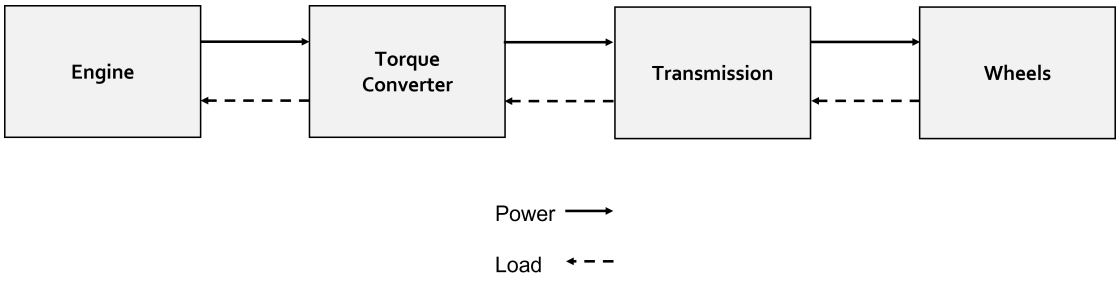


Figure 4.4: Power Flow and Loads in the Driveline.

Torque Converter

The torque converter, which major components are:

- **Pump**, which fins are attached to the flywheel of the engine, thus the pump turns at the same speed as the engine.
- **Turbine**, linked to the transmission causes the transmission to spin at the same speed as the turbine allowing then the car to move.
- **Transmission Fluid**, this makes the coupling between pump and turbine.

is a fluid coupling linking engine and transmission. If the engine is turning slowly, like when the car is idling at a stoplight, the torque amount passed through the converter is very small, so only a small pressure on the brake pedal is required to keep the car still. Besides allowing the car to completely stop without stalling the engine, it gives also the car more torque in case of accelerations out of a stop.

To model now the torque converter, the *Kotwicki's Static Model*(1982) can be used since this is desirable for control due to its simplicity. This model is a quadratic regression fit of the data from a simple experiment which involves measurements of only the input and output speed and torques of the torque converter.

By assuming T_p, T_t to be respectively the pump and turbine torques and ω_p, ω_t the pump and turbine speeds, the pump and turbine torques, for *converter mode* (i.e., $\omega_t/\omega_p < 0.9$) are (Equations 4.8a and 4.8b):

$$T_p = 3.4325 \times 10^{-3}\omega_p^2 + 2.2210 \times 10^{-3}\omega_p\omega_t - 4.6041 \times 10^{-3}\omega_t^2 \tag{4.8a}$$

$$T_t = 5.7656 \times 10^{-3}\omega_p^2 + 0.3107 \times 10^{-3}\omega_p\omega_t - 5.4323 \times 10^{-3}\omega_t^2 \tag{4.8b}$$

instead for *fluid coupling mode* (i.e., $\omega_t/\omega_p \geq 0.9$), pump and turbine torques are (Equation 4.9):

$$\begin{aligned} T_p &= T_t \\ &= -6.7644 \times 10^{-3}\omega_p^2 + 32.0024 \times 10^{-3}\omega_p\omega_t - 25.2441 \times 10^{-3}\omega_t^2 \end{aligned} \tag{4.9}$$

The I/O schematic of the torque converter model is displayed in Figure 4.5.

Figure 4.5: I/O Schematic of a Torque Converter Model.

Transmission Dynamics

The *transmission steady-state gear ratio* (R) value depends on: operating gear and final gear reduction in the differential. The operating gear is determined by a gear shift schedule that depends on both transmission shaft speed and throttle valve opening. In general, $R < 1$ and increases as the gear shifts upwards.

From the I/O schematic of the transmission model (Figure 4.6), T_t is the turbine torque and input to the transmission block, instead T_{wheel} is the torque transmitted to the wheels (thus, output from the transmission block).

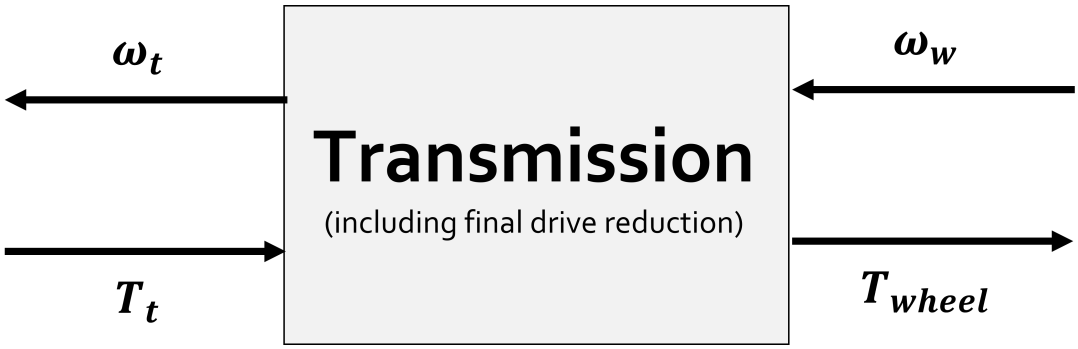


Figure 4.6: I/O Schematic of a Transmission Model.

At steady-state operation under the first, second or higher gears of the transmission, T_{wheel} is obtained as in Equation 4.10:

$$T_{wheel} = \frac{1}{R} \cdot T_t \tag{4.10}$$

and the relationship between transmission and wheels speeds is (Equation 4.11):

$$\omega_t = \frac{1}{R} \cdot \omega_w \tag{4.11}$$

Equations describing the dynamics during a gear change are obtained by replacing Eqns. (4.10) and (4.11) with first order equations (Equations 4.12a and 4.12b):

$$\tau \dot{T}_{wheel} + T_{wheel} = \frac{1}{R} \cdot T_t \tag{4.12a}$$

$$\tau \dot{\omega}_t + \omega_t = \frac{1}{R} \cdot \omega_w \tag{4.12b}$$

Eqn. (4.12a) is initialized with $T_{wheel} = 0$ at the instant in which the gear shift is started, R is the gear ratio at the new gear in which the transmission shifts. Eqn. (4.12b) is initialized with $\omega_t = (1/R_{old}) \cdot \omega_w$, where R_{old} is the old gear ratio. The gear change is assumed to be complete when T_{wheel} and ω_t converge to $(1/R) \cdot T_t$ and $(1/R) \cdot \omega_w$.

Engine Dynamics

The engine rotational speed dynamics is described by Equation (4.13):

$$I_e \dot{\omega}_e = T_i - T_f - T_a - T_p \tag{4.13}$$

where:

- T_i, ICE torque.
- T_f, torque due to frictional losses.
- T_a, accessory torque.
- T_p, pump torque representing the engine load from the torque converter.

By using the notation (Equation 4.14):

$$T_e = T_i - T_f - T_a \tag{4.14}$$

representing the *net engine torque* after losses, which depends on: dynamics in the intake (T_i) and exhaust manifold (T_f) of the engine and on driver accelerator input (T_a), Eqn. (4.13) can be rewritten as (Equation 4.15):

$$I_e \dot{\omega}_e = T_e - T_p \tag{4.15}$$

The engine I/O schematic is displayed in Figure 4.7.

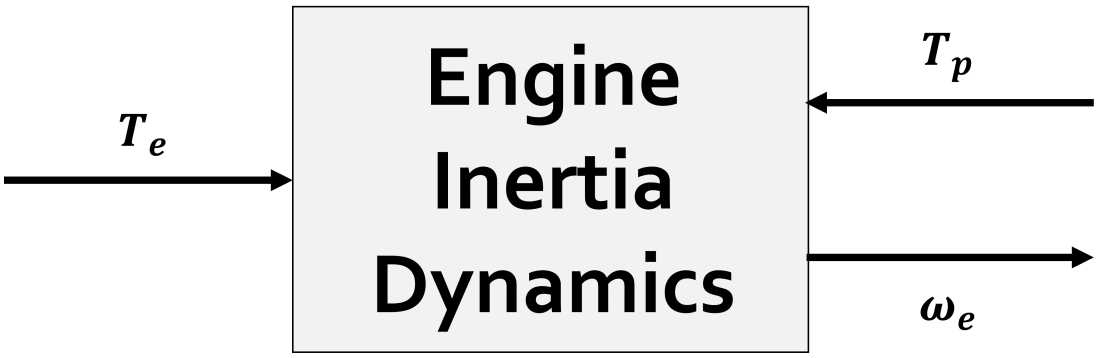


Figure 4.7: I/O Schematic of an Engine Inertia Model.

Wheel Dynamics

For a FWD car, the wheel rotational dynamics equations for the *driving* and *non-driven* wheels are (Equations 4.16a and 4.16b):

$$I_w \dot{\omega}_{wf} = T_{wheel} - r_{eff} \cdot F_{xf} \tag{4.16a}$$

$$I_w \dot{\omega}_{wr} = -r_{eff} \cdot F_{xr} \tag{4.16b}$$

The total longitudinal tyre force is given by the sum of the longitudinal tyre forces at the front and rear tyres (Equation 4.17):

$$F_x = F_{xf} + F_{xr} \tag{4.17}$$

The I/O schematic for the wheel dynamics is provided in Figure 4.8.

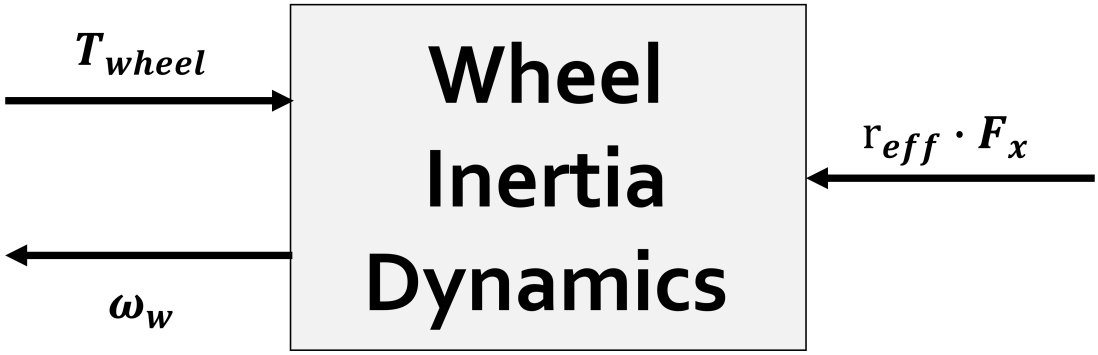


Figure 4.8: I/O Schematic of a Wheel Inertia Dynamics Model.

4.2 Vehicle Lateral Model

4.2.1 Kinematic Model of Vehicle Lateral Motion

To build a kinematic model for the vehicle lateral motion, some assumptions must be made based on the *Kinematics of Lateral Vehicle Motion* (Figure 4.9) and by considering the following aspects:

- Planar vehicle motion.
- Three coordinates describe the vehicle motion: x, y are the inertial coordinates of the location of the vehicle CoG; ψ is the vehicle yaw/heading angle, it describes the vehicle orientation.
- Vehicle velocity at the CoG is represented by v and makes an angle β (vehicle slip angle) with the vehicle longitudinal axis.

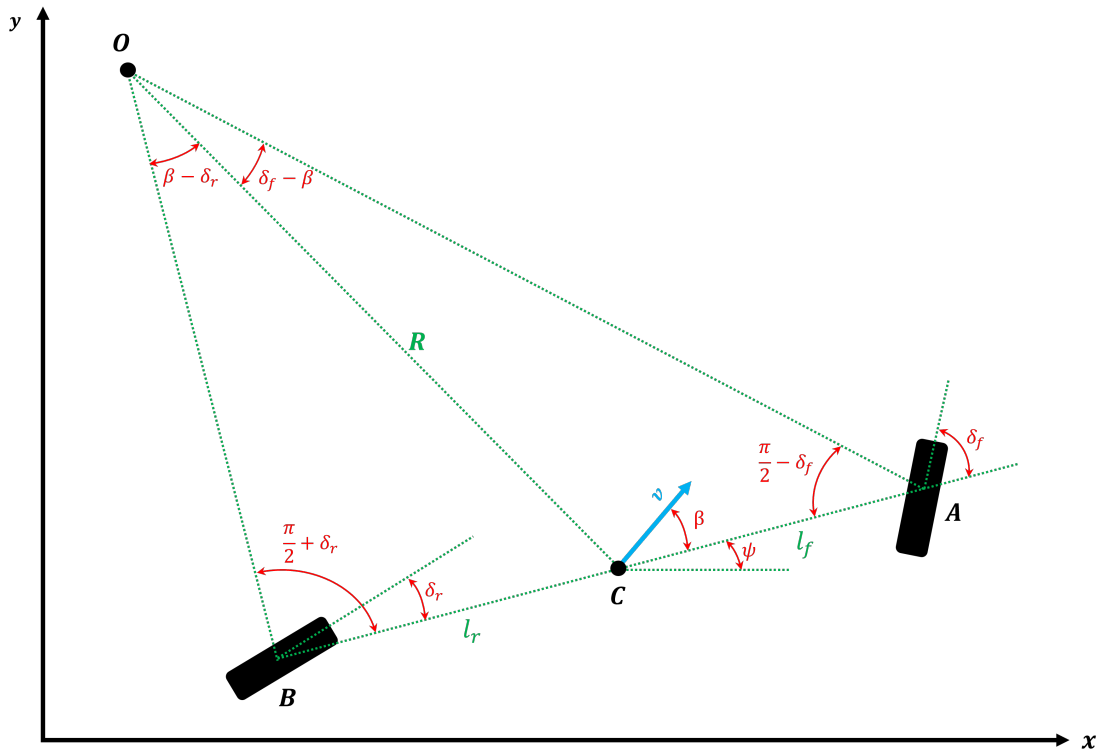


Figure 4.9: Kinematics of Lateral Vehicle Motion.

One of the primary assumptions made during the creation of the kinematic model is that the velocity vectors at points A and B are aligned with the front

and rear wheel orientations, respectively. Thus, at the front, the velocity vector makes an angle δ_f with the longitudinal axis of the vehicle and similarly at the rear with an angle δ_r with the longitudinal axis of the vehicle. This leads to the assumption that the slip angles at both wheels are zero, which is reasonable for low-speed motions (i.e., speeds smaller than 5 m/s, such as the one considered for the parking case). Therefore, since the total lateral force from both tyres to drive on any circular road of radius R varies quadratically with the speed (Equation 4.18), at low speeds, this force is small, thus leading to the assumption that the velocity vector at each wheel is in the direction of the wheel.

$$F_{tot,lat} = \frac{m \cdot v^2}{R} \tag{4.18}$$

With reference to Fig. 4.9:

- O, vehicle ICR (Instantaneous Center of Rotation) defined by the intersection of lines $\overline{AO}$, $\overline{BO}$ drawn perpendicular to the orientation of the two wheels.
- R, radius of the vehicle trajectory defined by the length of the line $\overline{CO}$ connecting the vehicle CoG, C, with the ICR, O.

By applying the *sine rule*[7] to triangles OCA and OCB, one obtains (Equations 4.19a and 4.19b):

$$\frac{\sin(\delta_f - \beta)}{l_f} = \frac{\sin(\frac{\pi}{2} - \delta_f)}{R} \tag{4.19a}$$

$$\frac{\sin(\beta - \delta_r)}{l_r} = \frac{\sin(\frac{\pi}{2} + \delta_r)}{R} \tag{4.19b}$$

and by applying trigonometric identities, like the *Angle Sum and Difference* and the *Reflection/Shift* to the left- and right-hand side of Eqns. (4.19a) and (4.19b), respectively, one obtains (Equations 4.20a and 4.20b):

$$\frac{\sin\delta_f \cdot \cos\beta - \sin\beta \cdot \cos\delta_f}{l_f} = \frac{\cos\delta_f}{R} \tag{4.20a}$$

$$\frac{\cos\delta_r \cdot \sin\beta - \cos\beta \cdot \sin\delta_r}{l_r} = \frac{\cos\delta_r}{R} \tag{4.20b}$$

[7]The sine rule relates the length of the sides of a triangle to the sine of its angles and states that the ratio of the length of a side of a triangle to the sine of the angle opposite that side is the same for all three sides of the triangle. Thus, given a triangle OCA with sides o, c, a and angles O, C, A opposite to the sides o, c, a respectively, then:

$$\frac{o}{\sin O} = \frac{c}{\sin C} = \frac{a}{\sin A}$$

Then, by multiply both sides of Eqn. (4.20a) by $l_f/\cos\delta_f$ and both sides of Eqn. (4.20b) by $l_r/\cos\delta_r$, Equations 4.21a and 4.21b are obtained:

$$\tan\delta_f \cdot \cos\beta - \sin\beta = \frac{l_f}{R} \tag{4.21a}$$

$$\sin\beta - \tan\delta_r \cdot \cos\beta = \frac{l_r}{R} \tag{4.21b}$$

and by adding Eqns. (4.21a) and (4.21b), Equation 4.22 is obtained:

$$(\tan\delta_f - \tan\delta_r) \cdot \cos\beta = \frac{l_f + l_r}{R} \tag{4.22}$$

Assuming now that the radius of the vehicle trajectory changes slowly (low speed assumption), the vehicle orientation rate of change, $\dot{\psi}$ (*yaw rate*) must be equal to the vehicle angular velocity ($\omega = v/R$). It follows (Equation 4.23):

$$\dot{\psi} = \frac{v}{R} \tag{4.23}$$

Thus, by merging Eqn. (4.23) into Eqn. (4.22), Equation 4.24 is obtained:

$$\dot{\psi} = \frac{v \cdot \cos\beta}{l_f + l_r} \cdot (\tan\delta_f - \tan\delta_r) \tag{4.24}$$

Finally, the *Overall Equations of Motion* are given by (Equation 4.25):

$$\begin{cases} \dot{x} = v \cdot \cos(\psi + \beta) \\ \dot{y} = v \cdot \sin(\psi + \beta) \\ \dot{\psi} = \frac{v \cdot \cos\beta}{l_f + l_r} \cdot (\tan\delta_f - \tan\delta_r) \end{cases} \tag{4.25}$$

In this model, three inputs are present: δ_f, δ_r and v. The latter is an external variable and can considered to be a time-varying function or it can be obtained from the vehicle longitudinal model.

The slip angle β can be obtained by subtracting from the product between Eqn. (4.21b) and l_f the product between Eqn. (4.21a) and l_r (Equation 4.26):

$$\beta = \arctan\left(\frac{l_f \cdot \tan\delta_r + l_r \cdot \tan\delta_f}{l_f + l_r}\right) \tag{4.26}$$

An important remark on the assumption of *single-track* model is that both left and right front wheels were collapsed into a single front wheel therefore leading the conclusion that the left and right steering angles are equal. This can be true in a first approximation but in reality, these steering angles are slightly different because the trajectory and thus the radius these wheels cover while turning is different. To have a clear view of this, consider the *Ackerman Turning Geometry* of Figure 4.10 in which a front wheels steered vehicle is considered and where:

- l_w, vehicle track width.
- δ_o and δ_i, outer and inner steering angles, respectively.
- L, vehicle wheelbase given by $L = l_f + l_r$.
- R, turning radius which is bigger than L.

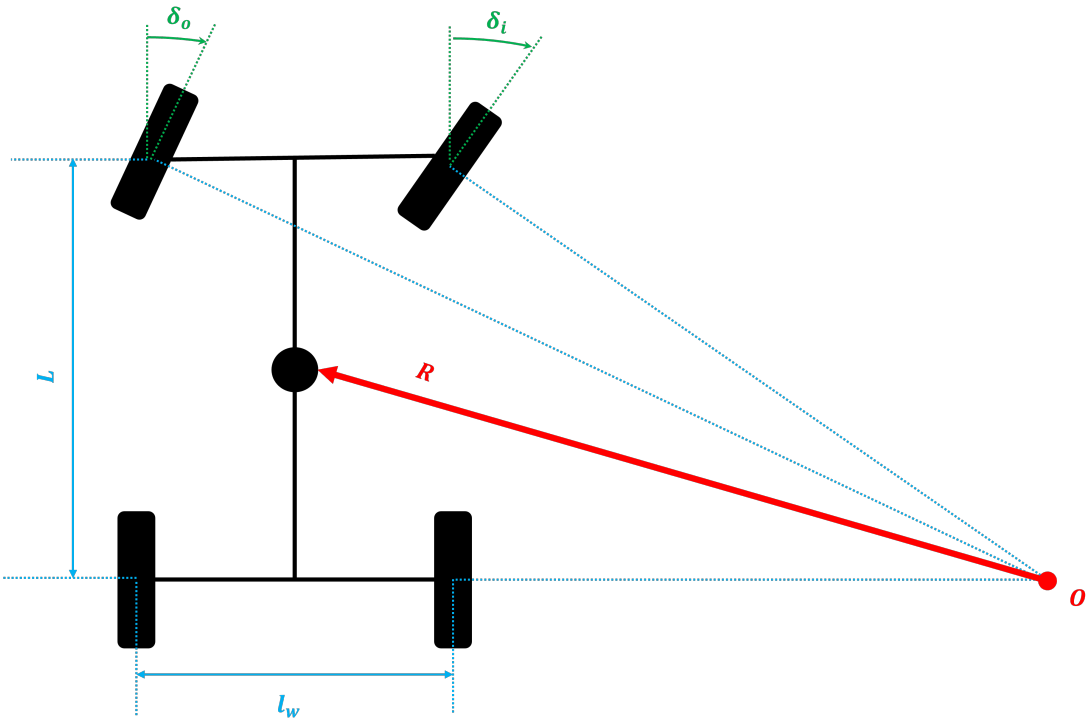


Figure 4.10: Ackermann Turning Geometry.

Then, if the slip angle β is small, the equation for the yaw rate, $\dot{\psi}$ in Eqn. (4.25) can be rewritten as (Equation 4.27):

$$\frac{\dot{\psi}}{V} \approx \frac{1}{R} = \frac{\delta}{L} \Longrightarrow \delta = \frac{L}{R} \tag{4.27}$$

Being the radius at the inner and outer wheels different, one has (Equations 4.28a and 4.28b):

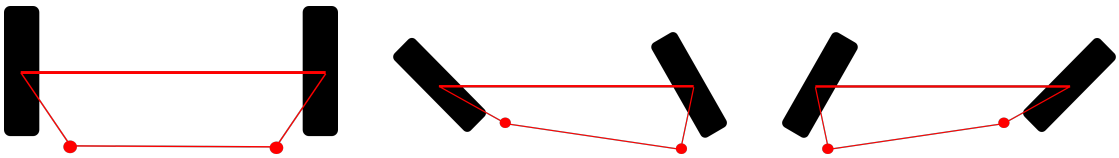
$$\delta_o = \frac{L}{R + \frac{l_w}{2}} \tag{4.28a}$$

$$\delta_i = \frac{L}{R - \frac{l_w}{2}} \tag{4.28b}$$

and the difference between Eqns. (4.28a), (4.28b) is (Equation 4.29):

$$\delta_i - \delta_o = \frac{L}{R^2} \cdot l_w = \delta^2 \cdot \frac{l_w}{L} \tag{4.29}$$

It follows that the difference in the steering angles at the front wheels is proportional to the square of the average steering angle. Such a differential steer can be obtained from a trapezoidal tie rod arrangement (Figure 4.11) where the inner wheel always turns a larger steering angle.



(a) Trapezoidal Geometry. **(b)** Left Turn. **(c)** Right Turn.

Figure 4.11: Differential Steer from a Trapezoidal Tie-Rod Arrangement.

4.2.2 Single-Track Model of Lateral Vehicle Dynamics

For the sake of completeness, the case of higher speed will be briefly discussed. In this scenario, the assumption that the velocity at each wheel is in the direction of the wheel no longer holds. Therefore, a dynamic model for lateral vehicle motion must be employed.

For this reason, start considering a *double track* model of the vehicle (Figure 4.12) where 2 DOFs are considered and which are:

- y, vehicle lateral position measured along the vehicle lateral axis to the point O, vehicle center of rotation.

- ψ, vehicle yaw angle measured with respect to the global X axis (V_x, longitudinal velocity at the vehicle CoG).

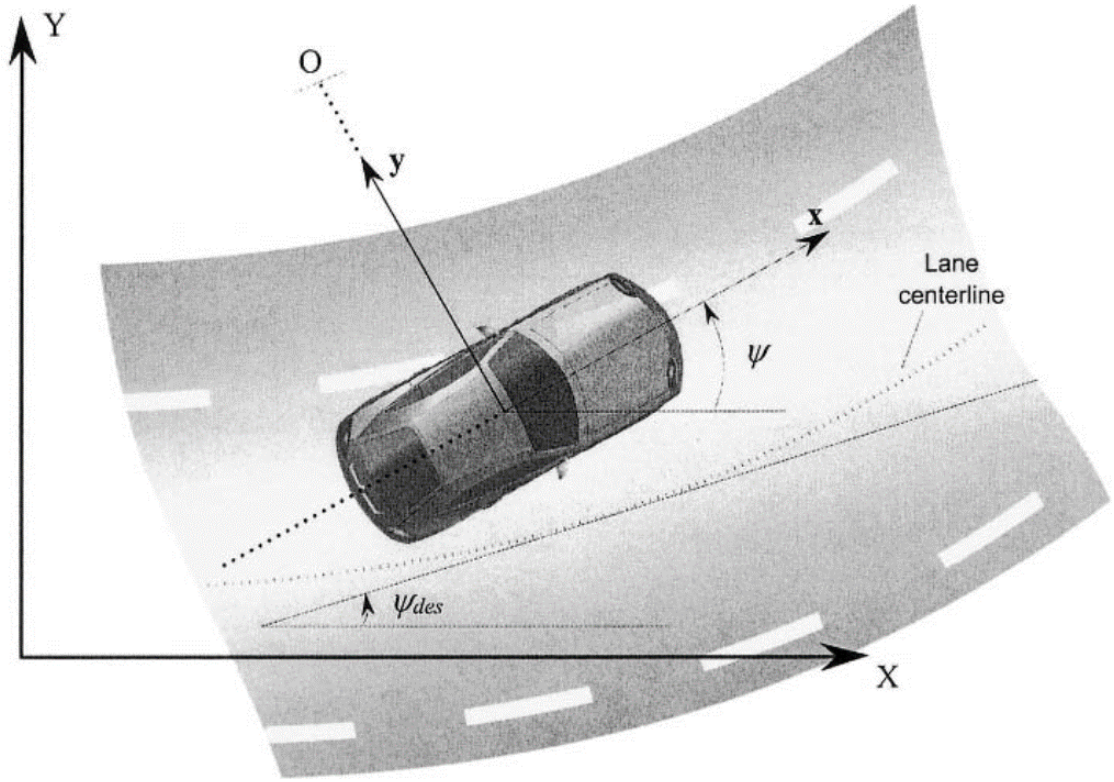


Figure 4.12: Lateral Vehicle Dynamics.

Ignoring for the moment the road banking and by applying Newton's Second Law of Motion along the y axis (Guldner, et. al., 1996 - Equation 4.30):

$$m \cdot a_y = F_{yf} + F_{yr} \tag{4.30}$$

where:

- $a_y = (d^2y/dt^2)_{inertial}$, vehicle inertial acceleration at the CoG in the y-axis direction to which the followings contribute (Equation 4.31):

$$a_y = \ddot{y} + V_x \cdot \dot{\psi} \tag{4.31}$$

  - $\ddot{y}$, acceleration due to the motion along the y axis.
  - $V_x \cdot \dot{\psi}$, centripetal acceleration

- F_{yf} and F_{yr}, lateral tyre forces at the front and rear tyres, respectively.

By substituting Eqn. (4.31) into Eqn. (4.30), the *Equation for the Vehicle Lateral Translational Motion* is (Equation 4.32):

$$m \cdot (\ddot{y} + V_x \cdot \dot{\psi}) = F_{yf} + F_{yr} \tag{4.32}$$

By performing a moment balance at the z axis, the *Yaw Dynamics Equation* (Equation 4.33) is obtained:

$$I_z \cdot \ddot{\psi} = l_f \cdot F_{yf} - l_r \cdot F_{yr} \tag{4.33}$$

The modeling of lateral tyre forces comes from experimental results showing that a tyre lateral tyre force is proportional to the slip angle α for small angles. The definition of slip angles is: *the slip angle is the angle between the orientation of the tyre and the orientation of the velocity vector of the wheel* (Figure 4.13). Thus,

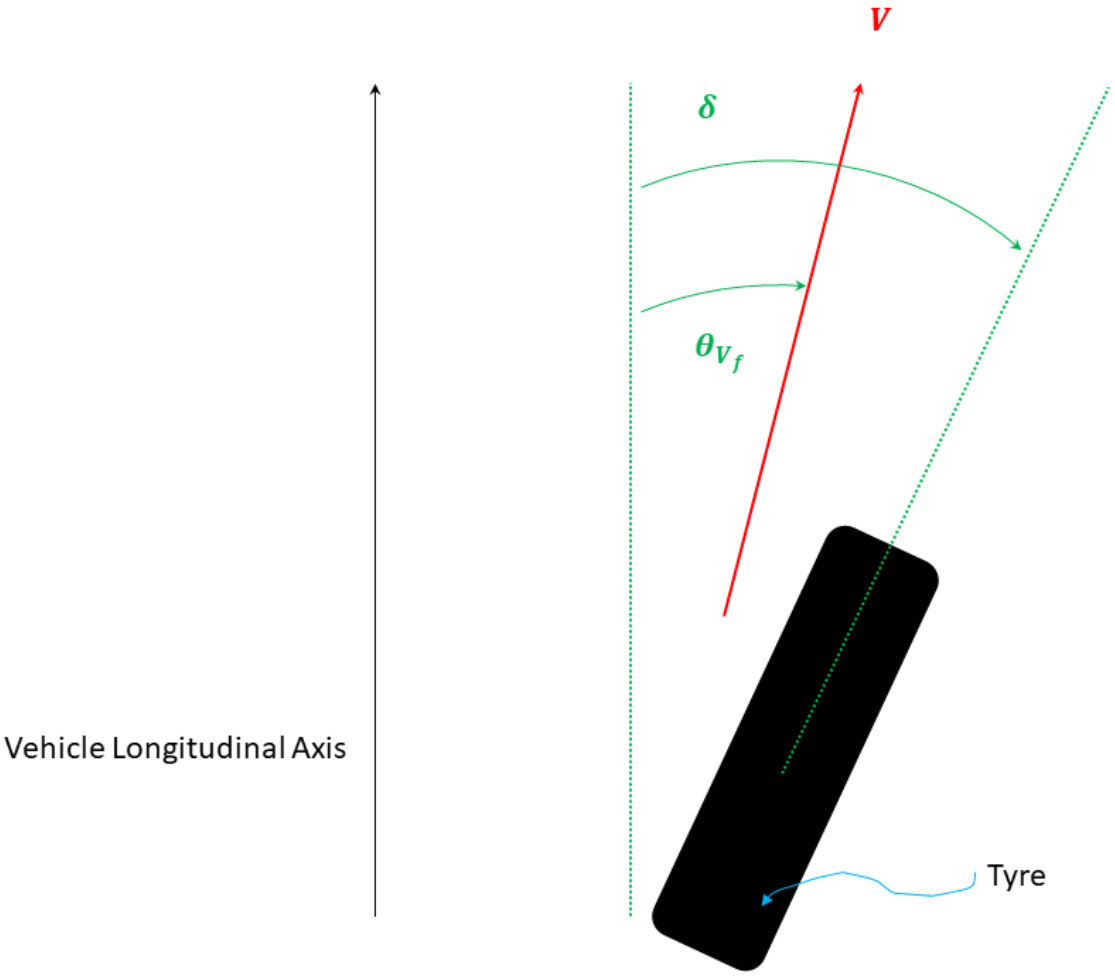


Figure 4.13: Tyre Slip Angle.

the slip angle at the front and rear wheel is (Equation 4.34a and 4.34b):

$$\alpha_f = \delta - \theta_{V_f} \tag{4.34a}$$

$$\alpha_r = -\theta_{V_r} \tag{4.34b}$$

where:

- δ, front wheel steering angle.
- θ_{V_f} and θ_{V_r}, front and rear tyre velocity angle, angle that the velocity vector makes with the vehicle longitudinal axis which can be calculated according to Equations 4.35a and 4.35b.

$$\tan\left(\theta_{V_f}\right) = \frac{V_y + l_f \cdot \dot{\psi}}{V_x} \tag{4.35a}$$

$$\tan\left(\theta_{V_r}\right) = \frac{V_y - l_r \cdot \dot{\psi}}{V_x} \tag{4.35b}$$

that can be rewritten as in Equations 4.36a and 4.36b by using $V_y = \dot{y}$ and the *small-angle approximation*:

$$\theta_{V_f} = \frac{\dot{y} + l_f \cdot \dot{\psi}}{V_x} \tag{4.36a}$$

$$\theta_{V_r} = \frac{\dot{y} - l_r \cdot \dot{\psi}}{V_x} \tag{4.36b}$$

The lateral tyre forces at the front and rear tyres are (Equation 4.37a and 4.37b):

$$F_{yf} = 2 \cdot C_{\alpha_f} \cdot \left(\delta - \theta_{V_f}\right) \tag{4.37a}$$

$$F_{yr} = -2 \cdot C_{\alpha_r} \cdot \theta_{V_r} \tag{4.37b}$$

with:

- "2", wheels number (i.e., two wheels) at the front and at the rear.
- C_{α_f} and C_{α_r}, cornering stiffness of each front and rear tyre, respectively.

Consequently, by merging Eqns. (4.34a), (4.34b), (4.36a) and (4.36b) into Eqns. (4.32) and (4.33), the *State Space Model* (Equation 4.38) can be written as:

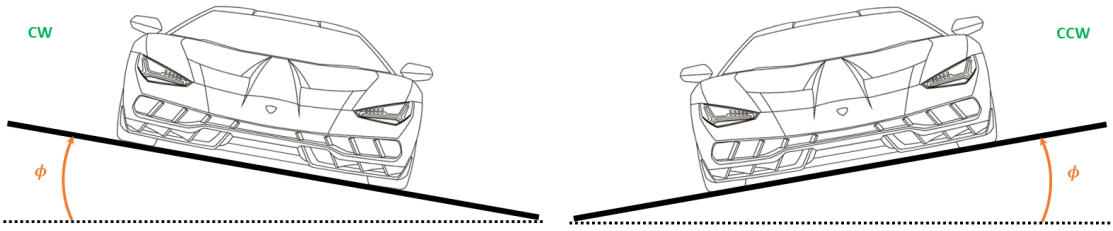
$$\frac{d}{dt}\begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{2\cdot C_{\alpha_f} + 2\cdot C_{\alpha_r}}{m \cdot V_x} & 0 & -V_x - \frac{2\cdot C_{\alpha_f}\cdot l_f - 2\cdot C_{\alpha_r}\cdot l_r}{m\cdot V_x} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{2\cdot C_{\alpha_f}\cdot l_f - 2\cdot C_{\alpha_r}\cdot l_r}{I_z \cdot V_x} & 0 & -\frac{2\cdot C_{\alpha_f}\cdot l_f^2 + 2\cdot C_{\alpha_r}\cdot l_r^2}{I_z \cdot V_x} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{2\cdot C_{\alpha_f}}{m} \\ 0 \\ \frac{2\cdot C_{\alpha_f}\cdot l_f}{Iz} \end{bmatrix} \cdot \delta \tag{4.38}$$

Finally, if the road banking is included, Eqn. (4.32) becomes (Equation 4.39):

$$m \cdot (\ddot{y} + V_x \cdot \dot{\psi}) = F_{yf} + F_{yr} + F_{bank} \tag{4.39}$$

where $F_{bank} = m \cdot g \cdot \sin\phi$ and ϕ is the banking angle which sign convention is the following (Figure 4.14[8]):

- Positive **clockwise** if the inclination increases leftward from the road center (Figure 4.14a).
- Positive **counterclockwise** if the inclination increases rightward from the road center (Figure 4.14b).



(a) Banking Angle ϕ positive CW. **(b)** Banking Angle ϕ positive CCW.

Figure 4.14: Sign Convention for the Banking Angle ϕ.

4.3 Ego Vehicle Model for NMPC

Since in this thesis work the vehicle steering, driving and braking have to be controlled, the vehicle dynamics (Section 4.2) is considered for modelling the vehicle for the simulation purposes. Then, by considering that in parking problems the vehicle travels at low speeds, the kinematic single-track model of the vehicle with front steering wheel is taken into account.

Therefore, the longitudinal, lateral and yaw motion of the vehicle, Eqn. (4.25), are taken into account. Then, under the assumption of single-track model with front steering wheel, the steering angle at the rear wheel, δ_r, is zero and thus, for simplicity assume $\delta_f = \delta$. It follows that Eqn. (4.26) simplifies into Equation 4.40:

$$\beta = \arctan\left(\frac{l_r \cdot \tan\delta}{wb}\right) \tag{4.40}$$

where $wb = l_f + l_r$ is the vehicle wheelbase that is equal to 2.8 m (default value

[8]**Car image credits to:** ID 217689120 © Andrey Vyrypaev | `https://thumbs.dreamstime.com/z/outline-drawing-super-car-view-three-sides-217689120.jpg`.

of the `vehicleDimensions`[9] object in the MATLAB® environment). However, the slip angles at both wheels are null ($\beta = 0$) for the parking problem due to low speeds (smaller than 5 m/s, Section 4.2).

Simplifying the *Overall Equations of Motion*, Eqn. (4.25), Equation 4.41 is obtained:

$$\begin{cases} \dot{x} = v \cdot \cos(\psi) \\ \dot{y} = v \cdot \sin(\psi) \\ \dot{\psi} = \frac{v}{wb} \cdot (\tan \delta) \end{cases} \tag{4.41}$$

where:

- (x, y), vehicle position.
- ψ, vehicle yaw angle.
- (x, y, ψ), *state variables* for the vehicle state functions.
- Speed v and steering angle δ, *control variables* for the vehicle state functions.

and the Jacobian matrix of the system of state equations is (Equation 4.42):

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \dot{x}}{\partial x} & \frac{\partial \dot{x}}{\partial y} & \frac{\partial \dot{x}}{\partial \psi} \\ \frac{\partial \dot{y}}{\partial x} & \frac{\partial \dot{y}}{\partial y} & \frac{\partial \dot{y}}{\partial \psi} \\ \frac{\partial \dot{\psi}}{\partial x} & \frac{\partial \dot{\psi}}{\partial y} & \frac{\partial \dot{\psi}}{\partial \psi} \end{bmatrix} = \begin{bmatrix} 0 & 0 & -v \cdot \sin(\psi) \\ 0 & 0 & v \cdot \cos(\psi) \\ 0 & 0 & 0 \end{bmatrix} \tag{4.42}$$

[9]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/driving/ref/vehicledimensions.html`

Chapter 5

Model Predictive Control

This chapter delves into the fundamental concepts of MPC (Model Predictive Control) and NMPC (Nonlinear Model Predictive Control), including their theoretical basis, algorithms and applications. It examines the principles, constraints and limitations of these control strategies and suggests methods to tackle some of these obstacles. The chapter objective is moreover to showcase the usefulness of NMPC in autonomous driving applications and provide ideas for further research in this domain.

5.1 Introduction

A controller, in general, is an algorithm that converts an error signal into an actuation signal to achieve a desired set-point for a given process [17]. Key terms are:

- **Control Variable (CV)**, variable being regulated.
- **Set-point**, CV desired value.
- **Error**, difference between the CV current state and the set-point.
- **Actuation**, signal sent to the process to influence the error reduction.
- **Process**, system being controlled. Sometimes referred to as *plant* or *transfer function*.

The Model Predictive Control (MPC) is a model-based control strategy that has evolved significantly since its inception in the late seventies. Nowadays, it is widely employed in several industries including automotive, aerospace and robotics. However, it is worth considering that the term MPC, when speaking in general

terms, refers to a broad range of control methods which basic working principle involves [18]:

- Explicit use of a model to predict future system behaviour (*horizon*).
- Calculation of a control sequence by minimizing an objective function.
- Receding strategy where the horizon is displaced towards the future at each step, with the first control signal of the sequence applied.

This results in linear controllers with similar structures and adequate DOFs, making it particularly useful for complex systems with constraints and nonlinear dynamics. However, the high computational requirements of MPC make it less feasible for real-time systems.

Nonlinear Model Predictive Control (NMPC) is an extension of MPC that can effectively control nonlinear systems by using a nonlinear dynamic model to predict future system behaviour and determine optimal control inputs, making it an effective tool for controlling nonlinear systems with constraints. Despite its potential to solve complex control problems, NMPC presents challenges, including increased computational complexity and the requirement for accurate system models. These challenges are due to the need to solve an optimization problem at each sampling time and the non-convexity of the cost function and constraints.

Notwithstanding these obstacles, NMPC has become increasingly popular due to its ability in handling nonlinear dynamics and constraints more accurately and efficiently. Although it is challenging to implement NMPC in real-time systems, especially for complex systems, researchers have proposed several approaches to overcome computational complexity, such as using reduced-order models or parallel computing techniques. Additionally, researchers are exploring new optimization methods and algorithms that can improve the efficiency of NMPC while maintaining its accuracy.

In summary, MPC and NMPC are effective control strategies for complex systems with nonlinear dynamics and constraints, making them well-suited for autonomous driving applications. However, the high computational requirements of these algorithms remain a challenge and further research is needed to improve their efficiency and scalability. Nevertheless, with recent advancements in computing power and algorithms, these techniques have become increasingly popular in various fields, including automotive, aerospace and chemical engineering. NMPC has been successfully applied in various fields, including process control, robotics and autonomous systems.

5.1.1 MPC Pipeline

According to [17], Model Predictive Controllers take a similar approach to dynamic driving tasks as humans do. Thus, an MPC pipeline can be defined as follows:

1. Establish constraints such as the vehicle dynamic model for estimating its state in the next time step: maximum steering angle, maximum throttle (or brake), etc.

2. Establish a cost function that includes the cost of not being at the desired state, the cost of using actuators and the cost of collisions.

3. Simulate possible trajectories and the associated control inputs that obey the mathematical cost and constraints for the next N time steps.

4. Use optimization algorithms to select the simulated trajectory with the lowest cost.

5. Execute the control input for one time step.

6. Measure the system state at the new time step.

7. Repeat steps from 3 to 6.

5.1.2 Design Parameters

The design parameters that are essential to the design process of various types of predictive controllers, including MPC and NMPC, include:

- Sample Time, T_s.
- Prediction Horizon h_p.
- Control Horizon h_c.

These parameters have a significant impact on both control performance and the computational burden of the algorithm. The optimal values of these parameters depend on the process characteristics, the actuation system and the control objectives. Therefore, selecting appropriate values for these parameters is crucial to ensure optimal controller performance.

Sample Time, T_s

The *sample time* T_s is the time between consecutive control calculations in the MPC algorithm. It determines how frequently the controller updates the control signal applied to the process.

This parameter represents the time interval between successive measurements of the process output and application of the control signal. It determines the rate at which the control algorithm is executed and, therefore, how quickly it can respond to changes in the process. A shorter sample time can lead to better control performance, but it can also increase the computational burden of the control algorithm.

In general, it is a given parameter and cannot be chosen. If there is the chance to choose it, the followings have to be considered [19]:

- It should be sufficiently small to deal with the plant dynamics.
- It should be not too small to avoid numerical problems and slow computations.

Prediction Horizon, h_p

The *prediction horizon* h_p is a crucial design parameter in MPC that determines the length of time for which the controller predicts the future behaviour of the process. It plays a crucial role in striking a balance between the accuracy of predictions and the computational burden of the algorithm. Specifically, h_p determines the number of future time steps for which the controller will optimize the control signal, making it an essential factor in deciding the overall control performance.

The choice of h_p is critical since it has both advantages and disadvantages and usually depends on the process characteristics, the actuation system and the control objectives. A *trial and error* procedure can be adopted to select the optimal value of h_p, keeping in mind that a long prediction horizon can improve the control performance allowing the controller to anticipate future changes in the process, thereby increasing closed-loop stability and robustness. However, it can also increase the computational burden of the algorithm and reduce the short-term tracking accuracy (performance degradation) [19].

Control Horizon, h_c

The *control horizon* h_c is the length of time over which the controller will apply the first part of the control sequence computed for the entire prediction horizon. It

determines how many control actions the MPC controller will take before recomputing the control sequence.

This parameter represents the length of time over which the control signal is applied to the process. The control horizon is typically shorter than the prediction horizon and is chosen to balance the control performance with the limitations of the actuation system. In general, a short control horizon reduces the computational time without affecting performance [19], while a longer control horizon can lead to more aggressive control actions but can also result in instability if the process is highly nonlinear or subject to significant disturbances.

Moving on to the overall concepts and parameters of the MPC problem, these are depicted in Figure 5.1[1].

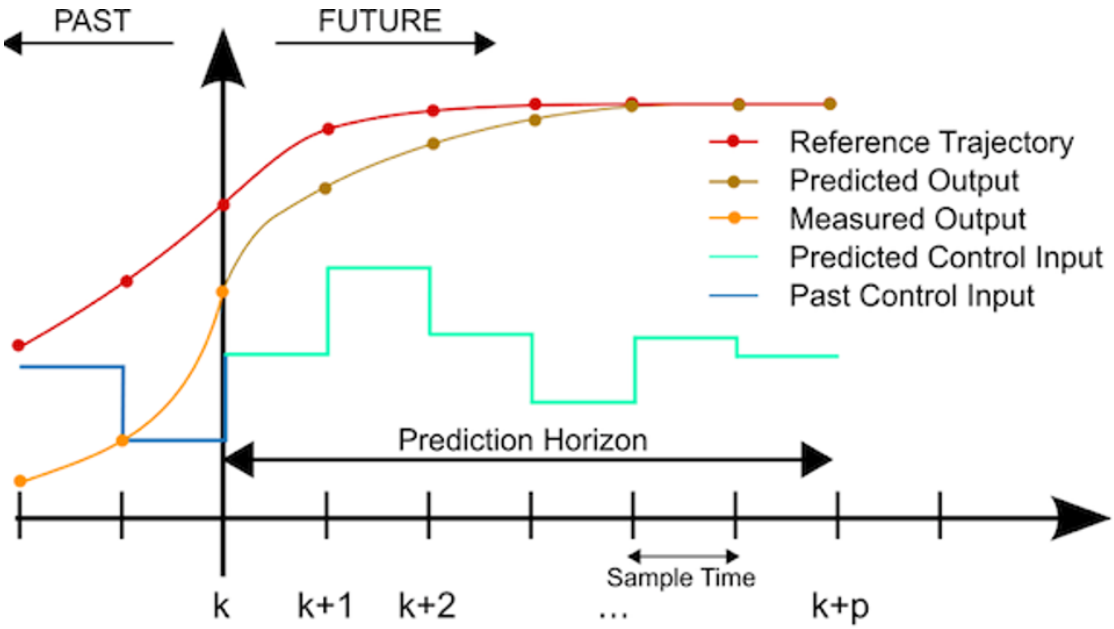


Figure 5.1: Concepts and Parameters constituting the MPC Problem.

The description of each parameter shown in Fig. 5.1 is:

[1]**Figure credits:** Wikipedia, *Model predictive control*, subchapter *Theory behind MPC*: `https://en.wikipedia.org/wiki/Model_predictive_control#/media/File:MPC_scheme_basic.svg`

- **Reference Trajectory**, desired trajectory of the controlled variable (e.g.: vehicle lateral position in the lane).
- **Measured Output**, measured past state of the controlled variable.
- **Predicted Output**, prediction of the controlled variable state after the predicted control input has been applied. It is informed by the dynamic model of the system, the constraints and the previously measured output.
- **Predicted Control Input**, system prediction of the control actuations that must be performed to achieve the predicted output.
- **Past Control Input**, actual control actuations performed in the past, leading up to the current state.

5.2 Applications in the Autonomous Driving

Self-driving cars require a controller to replace the human driver and control the steering, throttle and brake actuators to move the vehicle from its initial to target pose. Two common controllers used in self-driving cars are the PID and MPC, with the latter being the best option for mimicking human driving. MPCs are used in MIMO systems like a vehicle, for which the inputs are the throttle, brake and steering and the outputs can be the car lateral position relative to the lane center and car speed.

During human driving, the driver selects the destination, plans the route (waypoints) and executes it while continuously assessing the state of other cars, traffic signs, speed, throttle, brake and more. The driver also constantly simulates various manoeuvres based on the current state of traffic, weighing the cost of each manoeuvre before executing it. Ultimately, the driver chooses the manoeuvre with the lowest cost based on the cost attributed to each consideration.

Recent advances in computing speed have made the Model Predictive Control a popular and powerful tool for safe and efficient autonomous driving tasks. It has the potential to revolutionize the future of autonomous vehicles by improving their safety, efficiency and comfort [20]. By using an internal model to predict the future behaviour of the vehicle at each control interval, the MPC allows the controller to compute optimal control actions in real-time decisions, such as changing road conditions or unpredictable traffic patterns.

Common applications of MPC in autonomous driving include the development of the following ADAS:

- **Adaptive Cruise Control** (ACC) for maintaining a safe distance from the vehicle ahead.
- **Collision Avoidance System** (CAS) for predicting the behaviour of other vehicles on the road and helping an autonomous vehicle avoid collisions in real-time.
- **Lane-Keeping Assist** (LKA) and **Lane-Following Control** (LFC) for maintaining and following the vehicle trajectory within a lane and/or following a curved path.
- **Path Planning Algorithms and Trajectory Optimization** for optimizing an autonomous vehicle trajectory and speed based on a predictive model of the environment, enabling safer and more efficient decision-making.
- **Parking Assist** (PA) for helping the vehicle park itself in a safe and efficient manner.
- **Obstacle Avoidance** (OA) for detecting and avoiding obstacles on the road and predicting the behaviour of pedestrians and other obstacles to help an autonomous vehicle avoid collisions in real-time.

Among the MPC features, those valuable in automated driving are:

- Handling of I/O constraints: I/O constraints on the system are accounted for while computing optimal control moves.
- Ego vehicle behaviour prediction across receding horizon: the vehicle dynamics internal model is used by the MPC controller to predict how the vehicle will behave to a given control action across a prediction horizon.
- Reference trajectory and disturbances prediction across prediction horizon: by predicting reference trajectories or disturbances across the prediction horizon, the MPC controller can incorporate these information when computing optimal control actions.
- Internal vehicle model update at run time: adaptive MPCs are used to update the controller internal model if the dynamics of the ego vehicle vary over time, such as for velocity-dependent steering dynamics.
- Automatic code generation for deployment of model predictive controllers.

When discussing the applications of MPC in autonomous driving, it is important to provide examples of how MPC has been used in real-world applications and to discuss the benefits and limitations of the technology. Taking the lane keeping of a

self-driving car as an example, the set-point is defined by the lane center, which is where the driver wants to keep your car. However, to begin this process, it is first needed to calculate the error, which is the difference between the current position and the lane center, also known as the *Cross-Tracking Error* (CTE). Next, it is necessary to determine the appropriate actuator(s) to involve for bringing the car back to the lane center and minimize the CTE. In addition, the controller has to be thought as a function that continually attempts to reduce the car error relative to the set-point of that variable, given a CV.

5.3 Types of MPC

In this section the different types of Model Predictive Control used in the development of the autonomous parking algorithms, that will be presented in Chapter 6, are presented starting with the presentation of the classical MPC which constitutes the baseline for all MPCs.

5.3.1 Linear MPC

The classical MPC refers to a class of control problems involving Linear Time Invariant (LTI) systems whose dynamics are described by a discrete time model that is not subject to any uncertainty [21].

System Dynamics

In a first assumption, the system dynamics can be described by the LTI state-space model (Equation 5.1):

$$x_{k+1} = A \cdot x_k + B \cdot u_k \tag{5.1a}$$

$$y_k = C \cdot x_k \tag{5.1b}$$

Here, $x_k \in \mathbb{R}^{n_x}$, $u_k \in \mathbb{R}^{n_u}$ and $y_k \in \mathbb{R}^{n_y}$ denote the system state, control input and system output at time step $k = 0,\ 1,\ \ldots$ (discrete time index), respectively. The dimensions of x_k, u_k and y_k are:

- n_x, number of state equations: three in the autonomous parking case, Eqn. (4.41).
- n_u, number of inputs: two in the autonomous parking case, i.e. vehicle speed v and steering angle δ.
- n_y, number of outputs: three in the autonomous parking case, i.e. x, y positions and the yaw angle ψ

The state vector represents the current state of the system at a given time. It contains all the relevant physical variables that describe the state of the system. In the MPC problem, the state vector is typically used to represent the current values of the system internal variables, such as its temperature, pressure, or position.

The output vector, on the other hand, represents the measurable outputs of the system. These are typically the variables that are of interest to the controller and can be directly measured or inferred from sensors. For example, in a chemical reactor, the output vector might include the concentration of a certain chemical in the reactor, the temperature of the reactor or the flow rate of a certain gas.

The state vector and output vector are related through the system dynamics, which describe how the state of the system changes over time as a result of its inputs and internal dynamics. By using system dynamics models to describe the behaviour of the system, we can use MPC to optimize the system inputs over a future time horizon to achieve a desired set of outputs.

Linear Constraints

Additionally, the controlled system is assumed to be subject to linear constraints, which generally involve both states and inputs and are expressed as a set of linear inequalities (Equation 5.2):

$$F \cdot x + G \cdot u \leq \mathbf{1} \tag{5.2}$$

Here, $F \in \mathbb{R}^{n_C \times n_x}$ and $G \in \mathbb{R}^{n_C \times n_u}$ are matrices that define the constraints on the system state and control input, respectively. n_C denotes the number of constraints and $\mathbf{1}$ is a vector with elements equal to unit and which dimension is given by: $\mathbf{1} = [1 \cdots 1]^T \in \mathbb{R}^{n_C}$. By setting either F or G to zero, constraints only on inputs or states will result.

Control Objective

The goal of the controller is to find the sequence of control inputs that minimizes the cost function, while satisfying the system dynamics and constraints. This is typically achieved by solving an optimization problem over a finite horizon, which involves minimizing the cost function subject to the dynamics and constraints of the system. The resulting optimal control inputs are then applied to the system over the finite horizon and the process is repeated at the next time step. This is the basic idea behind Model Predictive Control.

One common approach to solve the classical MPC problem is to solve a finite horizon optimal control problem at each time step k. Specifically, the goal is to find

the optimal control input sequence $u_k^\star, u_{k+1}^\star, \cdots, u_{k+N-1}^\star$ that minimizes the cost function (Equation 5.3) subject to the state dynamics and the input constraints over a finite time horizon of N steps:

$$J_N(x_k, u_k, \ldots, u_{k+N-1}) = \sum_{i=k}^{k+N-1} \left(\|x_i\|_Q^2 + \|u_i\|_R^2 \right) + \|x_{k+N}\|_{Q_F}^2 \tag{5.3}$$

Here, $J_N(x_k, u_k, \ldots, u_{k+N-1})$ is the cost function that the controller aims to minimize, starting from the initial state x_k and the sequence of control inputs $u_k, \ldots, u_{k+N-1}$. The cost function is defined as the sum of the quadratic forms[2] of the state and input vectors, respectively weighted by the matrices Q and R which determine the relative importance of the state variables and control inputs in the cost function, plus a final weight matrix Q_F which specifies the emphasis on the final state.

- $Q \in \mathbb{R}^{n_x \times n_x}$, *symmetric* and *positive semidefinite*: all eigenvalues of Q are real and non-negative, $Q \succeq 0$.
- $R \in \mathbb{R}^{n_u \times n_u}$, *symmetric* and *positive definite*: the eigenvalues of R are real and strictly positive, $R \succ 0$.
- Q_F, *symmetric* and *positive semidefinite*.

The matrix Q penalizes deviations of the state variables from their desired values. Its diagonal elements represent the relative importance of the corresponding state variables in the cost function, while the off-diagonal elements represent the correlations between different state variables. For example, if the position of the vehicle is more important than its orientation, then the diagonal elements of Q corresponding to the position have to be larger than the diagonal element corresponding to the orientation.

The matrix R balances the trade-off between control effort and tracking performance. Its diagonal elements represent the relative importance of the corresponding control inputs in the cost function. If the control inputs are constrained, then R can be used to penalize violations of these constraints.

By adjusting the values of Q and R, the controller can be tuned to achieve different performance objectives. For example, increasing the weight on certain state variables in Q can improve the tracking performance of these variables at the cost of increased control effort. Similarly, increasing the weight on certain control

[2]General quadratic form: $\|w\|_A^2 = w^T A w$, $\forall w \in \mathbb{R}^{n_w} \wedge A = A^T \in \mathbb{R}^{n_w \times n_w}$.

inputs in R can reduce the magnitude of these inputs at the cost of poorer tracking performance.

The optimal control problem can be solved using numerical optimization techniques, such as quadratic programming or nonlinear programming. Once the optimal control input sequence is obtained, the first control input $u_k^\star$ is applied to the system and the process is repeated at the next time step $k+1$. This procedure is known as receding horizon control or model predictive control because the control sequence is recomputed at each time step based on the current state and future predictions. However, solving the optimization problem in real-time can be computationally demanding, especially for systems with high-dimensional state and input spaces or for problems with tight constraints. Additionally, accurate modeling of the system dynamics and constraints can be challenging and modeling errors can lead to suboptimal or even unstable control behaviour. Thus, a trade-off must be made between the complexity of the optimization problem and the accuracy of the model.

In summary, the classical MPC is a control approach for Linear Time Invariant systems with linear constraints. The goal is to find the optimal control input sequence over a finite time horizon that minimizes a quadratic cost function subject to state dynamics and input constraints. The control sequence is recomputed at each time step based on the current state and future predictions. The weighting matrices Q and R play a crucial role in determining the behaviour of the MPC controller. By adjusting these matrices, the controller can be tuned to achieve different trade-offs between tracking performance and control effort.

5.3.2 Nonlinear MPC

Nonlinear systems are systems where the relationship between the input and output is nonlinear, which means that the system behaviour is highly nonlinear and can be difficult to predict. In contrast, linear systems have a linear relationship between the input and output, which makes them easier to model and control. However, many real-world systems are nonlinear and it can be challenging to find a suitable control strategy that can handle their complex behaviour.

This is where Nonlinear Model Predictive Control comes in. One of its benefits is that it allows dealing with constraints on the system state(s), input(s) and output(s) and to manage the trade-off between performance and command effort. In addition, the input variable(s) are always constrained because typically, it is not possible to provide large inputs, for example, due to actuators limitations.

The command input is chosen as the one yielding the best prediction (the closest one to the desired behaviour) by online optimization algorithms. This process repeats at each time step, allowing the controller to adapt to changes in the system behaviour and achieve the desired performance.

By considering the MIMO nonlinear system in Equation 5.4:

$$\dot{x} = f(x, u) \tag{5.4a}$$
$$y = h(x, u) \tag{5.4b}$$

where:

- $x \in \mathbb{R}^{n_x}$, state.
- $u \in \mathbb{R}^{n_u}$, command input.
- $y \in \mathbb{R}^{n_y}$, output.

The state x is measured in real-time, with a sampling time T_s and the measurements are (Equation 5.5):

$$x(t_k), \quad t_k = T_s \cdot k \qquad \text{with} \qquad k = 0,1,\dots \tag{5.5}$$

where t_k is a time multiple of T_s and at each time t_k the system state and output are predicted over the time interval $[t,\ t + h_p]$[3]. The prediction is obtained by integration of the MIMO system of Eqn. (5.4).

At any time $\tau \in [t,\ t + h_p]$, the predicted output $\hat{y}(\tau)$ is a function of the initial state $x(t)$ and the input signal (Equation 5.6):

$$\hat{y}(\tau) \equiv \hat{y}(\tau,\ x(t),\ \hat{u}(t : \tau)) \tag{5.6}$$

where the initial condition $x(t)$ is fixed (state measured at a given time) and the future behaviour in this time interval depends on the input, which variation induces a change of the behaviour. $\hat{u}(t : \tau)$ denotes a generic input signal in the interval $[t,\ \tau]$ that has to be changed to obtain an optimal, desired behaviour of the system. Moreover, in the interval $[t,\ t + h_p]$, $\hat{u}(\tau)$ is an open-loop that does not depend on $x(\tau)$. With reference to Fig. 5.1, the region of interest is the one spanning from k over the prediction horizon h_p. In this region, a variation of $\hat{u}$ brings a change of $\hat{x}$, the predicted state and consequently of $\hat{y} = h(\hat{x}, \hat{u})$.

[3]The prediction horizon h_p is such that $h_p \geq T_s$.

At each time interval t_k, look for an input signal $\hat{u}(t:\tau) = u^\star(t:\tau)$ such that the predicted output (Equation 5.7):

$$\hat{y}(\tau,\ x(t),\ u^\star(t:\tau)) \equiv \hat{y}(u^\star(t:\tau)) \tag{5.7}$$

has the desired behaviour for $\tau \in [t,\ t+h_p]$.

The concept of desired behaviour is formalized in the definition of the objective function (Equation 5.8):

$$J\left(\hat{u}\left(t:t+h_p\right)\right) \doteq \int_t^{t+h_p} \left(\|\tilde{y}_p(\tau)\|_Q^2 + \|\hat{u}(\tau)\|_R^2\right) d\tau + \|\tilde{y}_p(t+h_p)\|_P^2 \tag{5.8}$$

where:

- $\tilde{y}_p(\tau) \doteq r(\tau) - \hat{y}(\tau)$, predicted tracking error: difference between the reference and the predicted output.
- $r(\tau) \in \mathbb{R}^{n_y}$, reference to track.
- $\hat{u}(\tau)$, input given to the model.
- P, weight for the final tracking error.

and the signal is taken in the whole interval and it is integrated, the term outside the integral represents the final time instant and it penalizes the deviation of the predicted output at the end of the time horizon from the desired output.

Generally, the values of Q, R and P are changed according to a trial and error procedure by keeping in mind that:

- Increasing Q and P leads to decrease the energy of x and y thus reducing oscillations and convergence time.
- Increasing R leads to decrease the energy of u thus reducing command effort and energy consumption.

The input signal $u^\star(t:t+h_p)$ is chosen as one minimizing the objective function $J(\hat{u}(t:t+h_p))$ and the goal is to minimize, at each time t_k, the tracking error square norm $\|\tilde{y}_p(\tau)\|_Q^2$ over a finite time interval.

The minimization of the cost function J is subject to constraints (Equation 5.9):

$$\dot{\hat{x}}(\tau) = f(\hat{x}(\tau), \hat{u}(\tau)),\ \ \hat{x}(t) = x(t),\ \ \tau \in [t,\ t+h_p] \tag{5.9a}$$
$$\hat{y}(\tau) = h(\hat{x}(\tau), \hat{u}(\tau) \tag{5.9b}$$

Other constraints that can be present are:

- Constraints on the predicted state/output: $\hat{x}(\tau) \in X_c$, $\hat{y}(\tau) \in Y_c$, $\tau \in [t, t+h_p]$.
- Constraints on the input: $\hat{u} \in U_c$, $\tau \in [t,\ t+h_p]$.

Optimization Problem

At each time t_k, for $\tau \in [t,\ t+h_p]$, the optimization problem to be solved is reported in Algorithm 1 where the last line in the optimization algorithm is a further constraint employed to simplify calculations, reduce the computational complexity of the optimization problem. Instead, for the first two constraints, the variables have to be consistent with the state equations.

Algorithm 1 NMPC Optimization Problem

$u^\star(t : t+h_p) = \arg\min_{\hat{u}(.)} J(\hat{u}(t : t+h_p))$
subject to:
$\dot{\hat{x}}(\tau) = f(\hat{x}(\tau), \hat{u}(\tau)), \quad \hat{x}(t) = x(t)$
$\hat{y}(\tau) = h(\hat{x}(\tau), \hat{u}(\tau))$
$\hat{x}(\tau) \in X_c, \quad \hat{y}(\tau) \in Y_c, \quad \hat{u}(\tau) \in U_c$
$\hat{u}(\tau) = \hat{u}(t+h_c), \quad \tau \in [t+h_c, : t+h_p]$

Here, the following holds: $0 < T_s < h_c < h_p$. This optimization problem is in general non-convex[4] and must be solved online at each time t_k.

Moreover, the input signal $\hat{u}(t : t+h_p)$ can be seen as a vector with an infinite number of elements. Hence, the optimization involves an infinite number of decision variables. To overcome this problem, the signal can be parameterized. Typical parameterizations are:

- *Piece-wise Constant Input* (Equation 5.10) for which the command input is assumed to be constant on each sub-interval and minimization is easier because it occurs on the sub-intervals and not on the whole interval.

$$\hat{u}(\tau) = u_p(\tau) = c_i \quad \text{for} \quad \begin{cases} \tau \in [t+(i-1)T_s,\ t+iT_s] \\ i = 1,\ \ldots,\ m \doteq \frac{h_c}{T_s} \end{cases} \tag{5.10}$$

- *Polynomial* (Equation 5.11) for which the input is expressed in polynomial form: $\hat{u}$ is a polynomial in the time, the independent variable; c_i are the

[4]For non-convex functions it is possible to find only local minima, while it is hard to find global minima. Instead for a convex function, every local minimum is a global minimum.

coefficients.

$$\hat{u}(\tau) = u_p(\tau) = \sum_{i=1}^{m} c_i \cdot (\tau - t)^{i-1} \tag{5.11}$$

In this way, the optimization is performed with respect to the finite dimension matrix $c = [c_1, \ldots, c_m] \in \mathbb{R}^{n_u \times m}$. m represents the number of parameters and in general, a small value of m and thus of number of parameters, is enough to obtain a satisfactory control performance. Usually, $m = 1$ is fine in many situations and for which the command input is constant for the whole prediction interval. In fact, with $m = 1$ it is like having a constant input and a reduced computational effort. Instead, with values equal to 3, 4 strange behaviours start already to appear [19].

5.3.3 Multistage Nonlinear MPC

Multistage NMPC (msNMPC) is a type of control strategy used to optimize the control of systems with nonlinear dynamics over a finite prediction horizon. In Nonlinear Model Predictive Control, the optimization of a control sequence that minimizes a cost function subject to constraints on the system state and input is done. However, in a Multistage NMPC (msNMPC), this idea is extended by considering the system behaviour over a horizon of multiple stages. The basic idea behind NMPC is to solve an optimization problem at each sampling time to determine the optimal control actions to be applied to the system over the prediction horizon, subject to system constraints such as state and control input bounds. In the case of msNMPC, the optimization problem is solved iteratively, with the solution of the previous iteration used as the initial guess for the next iteration. This approach allows for the use of more accurate nonlinear models for prediction over longer time horizons.

More in depth, the working principle of the iterative optimization algorithm is hereafter described. At each time step, an optimization problem is solved to obtain a sequence of control actions that minimizes the cost function over the horizon of multiple stages. Then the first control action from the sequence is applied to the system and the process is repeated at the next time step using the updated system state as the initial condition for the next optimization problem. Multistage NMPC is particularly useful for systems with nonlinear dynamics, but it can also be computationally expensive due to the repeated solution of the optimization problem at each sampling time. Therefore, careful consideration of numerical stability and convergence of the iterative optimization solver is required. Various techniques have been developed to address these challenges, including the use of warm-start

techniques[5], continuation methods and regularization strategies.

The general steps involved in implementing a Multistage NMPC include:

- Definition of the system dynamics: the mathematical model describing the system behaviour has to be defined. The model is expressed in terms of state variables and inputs.
- Definition of the cost function: a cost function quantifying the performance of the system under consideration has to be defined. This cost function is expressed in terms of state variables and inputs.
- Definition of the constraints: the constraints imposed by system physical limitations on the state variables and inputs have to be defined.
- Choice of a solver: a suitable optimization solver has to be chosen to solve the optimization problem arising in the msNMPC.
- Implementation of the iterative optimization algorithm: the iterative optimization algorithm that repeatedly solves the optimization problem over the horizon of multiple stages has to be implemented.

In conclusion, some real-case scenarios in which a Multistage NMPC could be beneficial are found in the following sectors:

- Chemical processes: optimize the control of processes that have often complex, nonlinear dynamics and are subject to physical constraints such as temperature, pressure and concentration.
- Robotics: optimize the control of robotic systems that often have nonlinear dynamics and are subject to constraints such as joint limits, collision avoidance and task objectives.
- Automotive systems: optimize the control of automotive systems such as engines, transmissions and chassis systems that have nonlinear dynamics and are subject to constraints such as fuel economy, emissions and safety.

[5]Warm-start techniques are optimization methods using a previous solution as an initial guess for the next optimization problem. This can significantly reduce the computation time and increase the numerical stability of the solver. In the context of msNMPC, warm-start techniques are used to improve the convergence of the iterative solver, by using the solution of the previous iteration as the initial guess for the next iteration.

- Power systems: optimize the control of power systems such as wind turbines, solar panels and energy storage systems that have nonlinear dynamics and are subject to constraints such as power output, energy storage capacity and grid stability.

Chapter 6

Perpendicular Parking: Software Implementation

In this final chapter, the implementation of the autonomous parking algorithm for the three different scenarios earlier introduced and reported here for completeness:

- Perpendicular Parking using Nonlinear Model Predictive Control (Sec. 6.1).
- Perpendicular Parking using Multistage NMPC (Sec. 6.2).
- Perpendicular Parking using RRT* Trajectory Planner and NMPC Tracking Controller (Sec. 6.3).

will be discussed alongside the presentation of the MATLAB® packages employed. Additionally, the notation Case-Study (CS) 1, 2 or 3 will be respectively used to distinguish the figures and tables of the three different cases mentioned above.

For all the case studies presented hereafter, it is assumed that the perpendicular parking manoeuvre being considered involves the vehicle moving backward, which occurs after the on-board sensors have detected an empty parking spot, causing the vehicle to come to a halt and initiating the parking manoeuvre.

6.1 Perpendicular Parking using Nonlinear MPC

The first implementation that is going to be discussed regards the design of a perpendicular parking controller using a simple NMPC.

6.1.1 Parking Environment

To design a NMPC for parking manoeuvres, the first step is to define the parking environment, which will be then in common among the three case studies. This environment comprises the ego vehicle, three parking slots (one of which is empty), and eight static obstacles:

- Two perpendicularly parked vehicles.
- Six road curbsides delimiting the parking lot and the road borders, one of which is referred to as `upperRoadLine` in the algorithm of Appendix B and it represents the upper road line (the one having its middle axis in $y = 6$) in the parking environment of Figure 6.1.

As one can imagine, the goal of the ego vehicle is to complete a perpendicular parking manoeuvre without colliding with any of the obstacles.

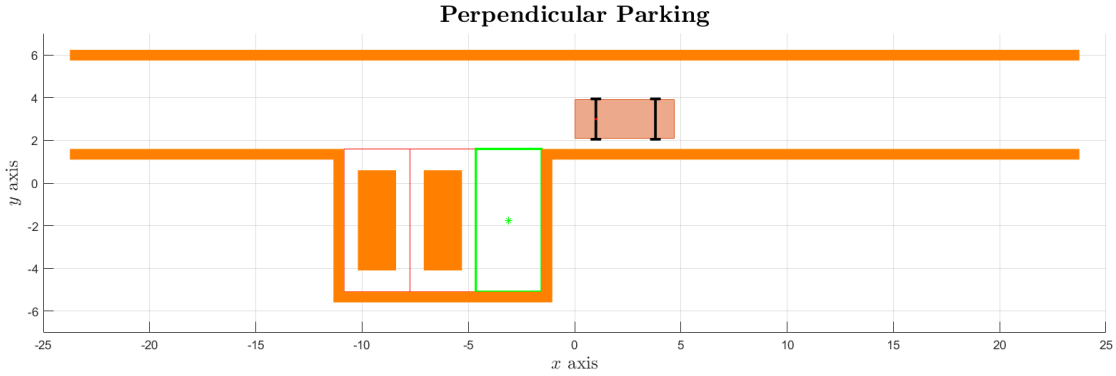


Figure 6.1: Perpendicular Parking Environment in MATLAB®.

The parking environment of Fig. 6.1, that is thus mainly constituted by the obstacles, is designed in MATLAB® function script called `helperSLVisualizeParking` that indeed creates and initializes a visualizer parking simulation. The visualizer shows the parking lot and vehicles, including the ego vehicle and obstacles. It takes the third pose entry of the ego vehicle, i.e. the yaw angle ψ and the steering angle as input arguments (App. B, lines $5 - 6$).

Then, it initializes and sets the properties of the figure handle and axes (App. B, lines $12 - 25$). The visualizer has a 2D representation of the parking lot, including the occupied parking lots and the target parking lot. It also shows the obstacles, including parked cars, road curbside and road line (App. B, lines $27 - 50$).

The inner function `createObstacles` creates collision boxes to represent the obstacles and transforms them to their respective positions. The obstacles are

returned as a cell array (App. B, lines $70 - 110$ and $40 - 43$). The function then creates a plot of the ego vehicle and sets its properties. The plot of the vehicle is then updated based on the input arguments (App. B, lines $53 - 68$).

The obstacles inside the parking environment were implemented using collision boxes created with the help of the MATLAB® function `collisionBox` (Robotics System Toolbox[1]). This function enables the creation of box collision geometries centred at the origin. A box primitive is defined by its three side lengths and is aligned with its own body-fixed frame, which has its origin at the center of the box[2].

To ensure the correct functioning of the algorithm, it is important to discuss the dimensions of the parking environment. Specifically, the design of the road curbsides and lane width is detailed in lines 84 to 106 of the algorithm provided in Appendix B. On the other hand, the dimensions of the parking spots and parked cars are designed in lines 30 to 36 and 76 to 82, respectively. To provide clarity, the dimensions are summarized in Table 6.1 and also displayed in Figure 6.2.

Description	**Dimension** $[m]$
Road Length	47.5
Lane Width	4.15
Road Curbside Width	0.5
Parking Slot Length	6.7
Parking Slot Width	3.1

Table 6.1: Relevant Dimensions of the Parking Scenario.

[1]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/robotics/index.html?s_tid=CRUX_lftnav`.

[2]**Syntax:** `BOX = collisionBox(X, Y, Z)`. For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/robotics/ref/collisionbox.html?searchHighlight=collisionBox&s_tid=srchtitle_collisionBox_1`.

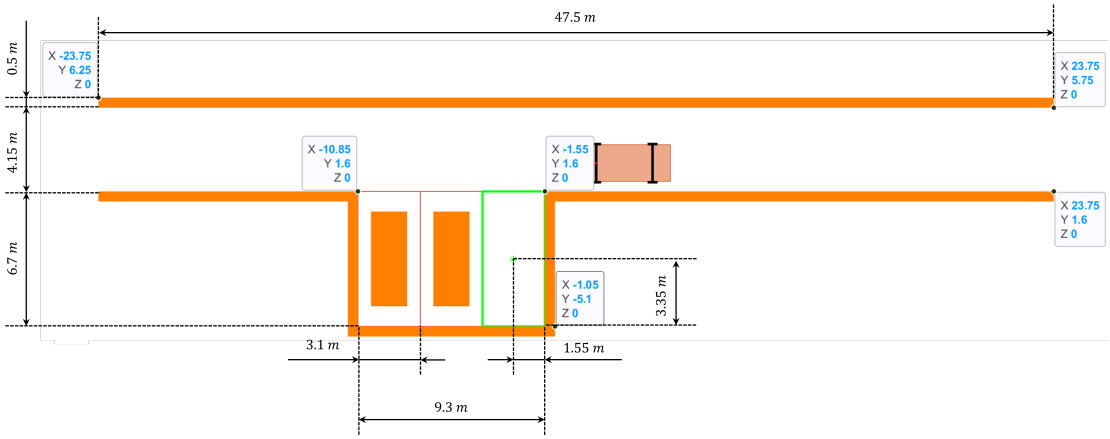


Figure 6.2: Parking Environment Dimensions.

The green asterisk plotted at line 38 of App. B represents the center of the empty parking spot relative to which the ego vehicle will park.

6.1.2 Ego Vehicle Definition

In Section 4.3 it was stated that for parking problems the vehicle is driving at low speeds ($\sim$ 5 m/s) and for this, a kinematic single-track model of the vehicle with front steering angle is employed. The *Simplified Overall Equations of Motion* that describe the motion of the ego vehicle are presented in Equation (6.1), which is a restatement of Eqn. (4.41) for clarity.

$$\begin{cases} \dot{x} = v \cdot \cos(\psi) \\ \dot{y} = v \cdot \sin(\psi) \\ \dot{\psi} = \frac{v}{wb} \cdot (\tan \delta) \end{cases} \tag{6.1}$$

These state functions are implemented in a MATLAB® function script called `parkingVehicleStateFcn` (Appendix C) where the variables are defined and where the state functions are labelled as: $\dot{x}_1$, $\dot{x}_2$ and $\dot{x}_3$.

As regards the vehicle dimensions, this together with its initial (`egoInitialPose`) and target pose (`egoTargetPose`) are defined at the beginning of the `main.m` file (Algorithm 2).

Algorithm 2 Ego Vehicle Definition

```
vdims = vehicleDimensions;
egoWheelbase = vdims.Wheelbase;
distToCenter = 0.5 * egoWheelbase;
egoInitialPose = [1, 3, 0];
egoTargetPose = [-3.1, -1.75 - distToCenter, π/2];
```

Here, the `vehicleDimensions` (Automated Driving Toolbox[3]) object is used to define the vehicle dimensions. The syntax `vdims = vehicleDimensions` is chosen among the provided ones because this creates a vehicle with default dimensions. Figure 6.3[4] provides a graphical representation of the dimensions stored by the function while Table 6.2 summarizes their default values.

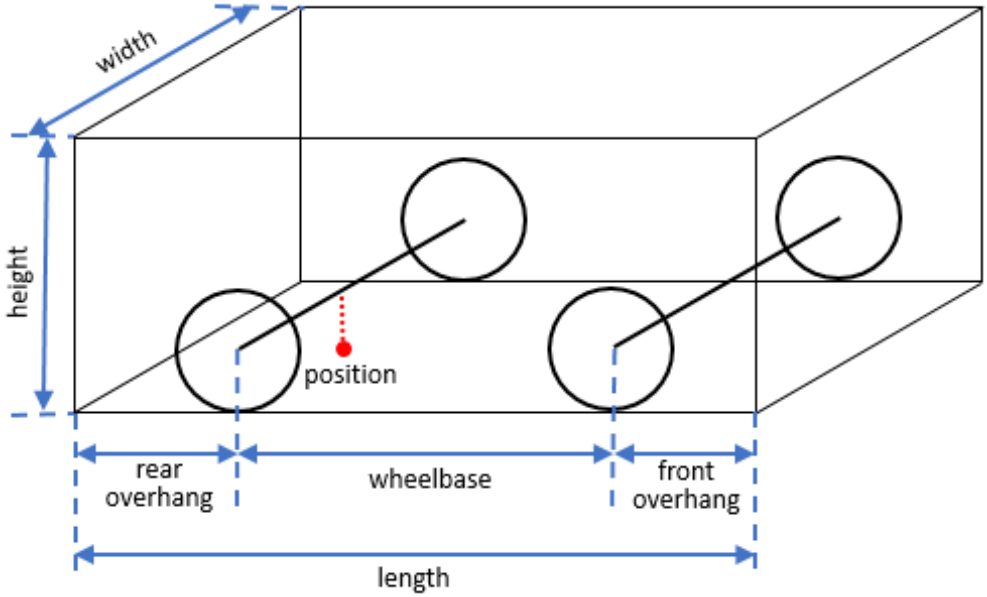


Figure 6.3: Vehicle Dimensions stored in the `vehicleDimensions` Object.

Notably, the vehicle position is defined as a single point on the ground at the centre of the rear axle, which corresponds to the vehicle natural centre of rotation [22]. This point serves as the reference point for the ego vehicle pose.

[3]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/driving/index.html?s_tid=CRUX_lftnav`.

[4]**Figure Credits:** Image adapted from The MathWorks, Inc. `https://it.mathworks.com/help/driving/ref/vehicledimensions.png`.

Description	**Dimension** $[m]$
Length	4.7
Width	1.8
Height	1.4
Wheelbase	2.8
Rear Overhang	1.0
Front Overhang	0.9

Table 6.2: Default Dimensions of the Ego Vehicle.

In terms of the initial and final poses, both are with respect to the *position* of the ego vehicle highlighted in Fig. 6.3. The initial pose is arbitrarily selected, while the target one is chosen by considering the location of the centre of the target parking spot, $(-3.1,\ -1.75)$. Since a transverse parking manoeuvre is being executed, the vehicles yaw angle must also be taken into account, as the vehicle undergoes a 90-degree turn to reach the final pose. Furthermore, being the vehicle position at the centre of the rear axle (Fig. 6.3), the "offset" with the vehicle centre (`distToCenter`) needs to be accounted for and results in being equivalent to half of the wheelbase, 1.4 m. As a result, the actual y-coordinate of the ego vehicle target pose must be less than -1.75 m, which is equivalent to -3.15 m.

6.1.3 Design the Nonlinear Model Predictive Controller

To design a Nonlinear Model Predictive Control[5] for parking purposes, the following assumptions have to be made [12]:

- The output of the vehicle state function is the same as the vehicle state, i.e. $(x,\ y,\ \psi)$, implying the NMPC object to be created with (App. F, lines $16 - 20$):
  - $n_x = 3$ states.
  - $n_y = 3$ outputs.
  - $n_u = 2$ manipulated variables (MV).
- The ego vehicle speed v is bounded between $[-2,\ 2]$ m/s (App. F, lines $37 - 38$).

[5]For more information on how to implement a NMPC in MATLAB®, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/mpc/ug/nonlinear-mpc.html`.

- The ego vehicle steering angle δ is bounded between $[-45,\ 45]$ deg (App. F, lines $39 - 40$).
- The NMPC controller uses a custom cost (Equation 6.2) similar to that presented in Eqn. (5.8) and defined in a MATLAB® function script called `parkingCostFcn` (Appendix D).

$$\begin{aligned} J = \int_0^d (s(t) - s_{ref})^T Q_p (s(t) - s_{ref}) + u(t)^T R_p u(t) dt + \\ + (s(d) - s_{ref})^T Q_t (s(d) - s_{ref}) + u(d)^T R_t u(d) \end{aligned} \tag{6.2}$$

  Here:

  - d is the simulation duration.
  - $s(t)$ is the states of the ego vehicle at time t.
  - s_{ref} is the target pose of the ego vehicle.
  - Q_p, R_p are the state and input process weight matrices, respectively.
  - Q_t, R_t are the state and input terminal weight matrices, respectively.

- To avoid collisions with obstacles, the NMPC must satisfy the inequality constraint of Equation 6.3 where the minimum distance to all obstacles, $dist_{min}$, must be greater than the safety distance $dist_{safety}$ (chosen to be $0.10\ m$).

$$dist_{min} \geq dist_{safety} \tag{6.3}$$

  Here, since the ego vehicle obstacles are modeled as `collisionBox` objects, the distance from the ego vehicle to the obstacles is computed in the MATLAB® function script called `parkingIneqConFcn` (Appendix E) using the `checkCollision`[6] (Robotics System Toolbox) function.

The Nonlinear Model Predictive Control is thus designed and reported completely, with the final values for each variable, in Appendix F. Moreover, the Jacobians of the state function (App. F line 43), cost function (App. F line 47) and inequality constraints (App. F line 50) are provided to the NMPC controller to improve the simulation efficiency.

The design starts with the specification of the sample time T_s, the prediction horizon h_p and the control horizon h_c by following the guidelines presented in Subsection 5.1.2 and resulting in (Table 6.3).

[6]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/robotics/ref/rigidbodytree.checkcollision.html?searchHighlight=checkCollision&s_tid=srchtitle_checkCollision_1`.

Parameter	**Value** $[s]$
Sample Time T_s	0.25
Prediction Horizon h_p	35
Control Horizon h_c	35

Table 6.3: CS 1: NMPC Design Parameters Values.

These design parameters are defined in App. F, lines $1-3$ and are specified for the controller at lines $22-24$.

The process and terminal weight matrices (Q_p, R_p and Q_t, R_t) are tuned through a trial and error procedure, which is discussed in detail in Subsection 6.1.4. This procedure takes into account the behavior of the ego vehicle„ i.e. how it drives to reach the target pose, as well as the results obtained from the MATLAB® `analyseParkingResults` function script (Appendix G), which has the following template:

1. **Valid/Invalid result**, it is based on the occurrence of a collision and analyses the data coming from the `parkingIneqConFcn.m` function. If no collisions are present, then the result is valid (App. G, lines $5-10$).
2. **Minimum distance to obstacles**, it is valid when greater than $dist_{safety}$ (App. G, lines $12-13$).
3. **Optimization exit flag**, it extracts the solution details from the variable `ExitFlag`[7] of the MATLAB® object `nlmpcmove`[8] (Model Predictive Control Toolbox). It is successful when positive (App. G, lines 12 and 14).
4. **Elapsed time** $[s]$ **for** `nlmpcmove`[9] (App. G, line 15).

[7]Optimization exit code, returned as one of the following:

- Positive integer: optimal solution found.
- 0: feasible suboptimal solution found after the maximum number of iterations.
- Negative integer: no feasible solution found.

[8]`nlmpcmove` computes the optimal control action for NMPC controller. For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/mpc/ref/nlmpc.nlmpcmove.html?searchHighlight=nlmpcmove&s_tid=srchtitle_nlmpcmove_1`.

[9]Actually this value is influenced by many factors e.g.: number of pages or applications opened on the laptop, CPU and so on.

5. **Final states error in** x $[m]$**,** y $[m]$ **and** ψ [deg]. It computes the error for the state variables (App. G, lines $17 - 19$), not in absolute terms, by taking the final values from the *optimal prediction model state sequence* `Xopt` (Equation 6.4).

$$\epsilon = \text{optimal} - \text{reference} \tag{6.4}$$

6. **Final control inputs speed** $[m/s]$ **and steering angle** [deg]. It takes the final values of from the *optimal manipulated variable sequences*, `MVopt`, for the two manipulated variables (App. G, lines $21 - 22$). These values correspond to the last control inputs provided to the actuator before the car has eventually brought to a halt.

The final values for the weight matrices are reported in App. F, lines $4 - 8$.

The next step is the definition of the safety distance, set at 0.1 m (App. F line 10), used by the controller when defining its constraints. After this, the `nlmpc`[10] (Model Predictive Control Toolbox) object is created alongiside with the definition of the constraints for the MV, as discussed earlier.

Then, from line 42 to line 50 of App. F, the followings are specified:

- Controller state function and Jacobian of the state function.
- Controller cost function and Jacobian of the cost function.
- Controller inequality constraints and Jacobian of the inequality constraints. The constraints compute the distance from the ego vehicle to all the obstacles in the environment and compare the distances with $dist_{safety}$.

Those specifications are followed by the configuration of the optimization solver (App. F, lines $52 - 56$) and the definition of the optimal state solution (App. F, lines $58 - 60$).

The last part of the NMPC controller design is related to the passing of the parameters to the function (App. F, lines $62 - 64$).

[10]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/mpc/ref/nlmpc.html?searchHighlight=nlmpc&s_tid=srchtitle_nlmpc_1`.

6.1.4 Controller Simulation in MATLAB®

To simulate the NMPC controller in MATLAB®, the `nlmpcmove` function is used inside the MATLAB® `runParkingAndPlot` function script (Appendix H). In this function script also the `buildMEX`[11] (Model Predictive Control Toolbox) function is used to improve the simulation efficiency. However, as can be seen from Algorithm 3, for this simulation a MEX file is not built (`useMEX` set to 0) because, as it will be shown later on in this section, the simulation duration is already small.

Algorithm 3 Controller Simulation in MATLAB®

```
useMEX = 0;
runParkingAndPlot
```

In App. H also the use of another MATLAB® function script is reported, i.e. `plotAndAnimateParking`, that animates and plots the results for the parking manoeuvre (Appendix I).

As stated in Subsec. 6.1.3, multiple simulations were conducted to determine the optimal set of weight matrices. The results of the first four simulations are presented in Table 6.5 which presents the various values for the Q_p weight matrix and brief comments on the outcomes. As concerns the other three weight matrices, namely R_p, Q_t and R_t, these are kept constant throughout those simulations with values reported in Table 6.4.

R_p	Q_t	R_t
$0.01 * \texttt{eye}(2)$	`diag([1 5 100])`	$0.1 * \texttt{eye}(2)$

Table 6.4: CS 1 (Trials 1 to 4): Values for the Weight Matrices R_p, Q_t and R_t.

Trial	Q_p	**Comment**		
1	`diag([0.1 0.1 0])`	Invalid: collisions	$dist_{min} = 0.0072$	`ExitFlag` $= -2$
2	`diag([1 1 0])`	Invalid: collisions	$dist_{min} = 0.0491$	`ExitFlag` $= -2$
3	`diag([10 10 1])`	Invalid: collisions	$dist_{min} = -10.0$	`ExitFlag` $= -2$
4	`diag([50 50 1])`	Invalid: collisions	$dist_{min} = -10.0$	`ExitFlag` $= -2$

Table 6.5: CS 1 (Trials 1 to 4): Tuning of the Process Weight Matrices.

[11] For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/mpc/ref/nlmpc.buildmex.html`.

Upon examining the ego vehicle behaviour in simulations (Figure 6.4), it became evident that the fourth trial performed the poorest, leading to abrupt steering of the ego vehicle, which resulted in collisions with the obstacles present in the scenario. This is highlighted by the path, represented as red dots, in Fig. 6.4.

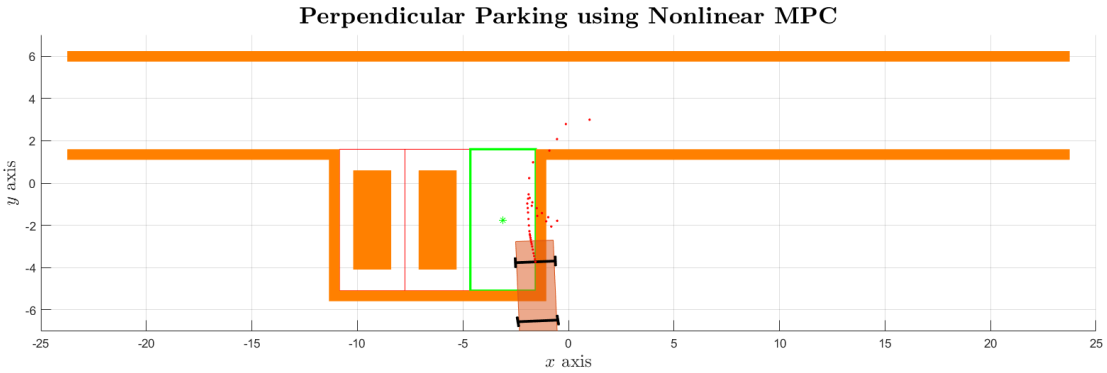


Figure 6.4: CS 1: Ego Vehicle Behaviour at the End of Trial 4.

Moreover, by looking at the results listed in Table 6.5, one can be influenced by the value of the $dist_{min}$ and think that the trial and error procedure should continue with the value of Q_p either from simulation 1 or from simulation 2. However, to determine the best choice, it is necessary to look at the complete *Parking Results Analysis*. For this reason, the weight matrix Q_p from trial 3 is chosen to continue with further simulations because this shows the best values in terms of final states error and final control inputs, besides a good simulation time (Table 6.6).

Trial	**Description and Value**		
	Elapsed time for `nlmpcmove`	Final states error	Final control inputs
Trial 1	15.8576 s	$x = 0.0339\ m$ $y = 0.2855\ m$ $\psi = -6.3237\ \text{deg}$	$v = -1.0560\ m/s$ $\delta = 43.7442\ \text{deg}$
Trial 2	7.0864 s	$x = -0.0514\ m$ $y = -0.1805\ m$ $\psi = 2.2082\ \text{deg}$	$v = -0.6897\ m/s$ $\delta = -45.0000\ \text{deg}$
Trial 3	9.5116 s	$x = -0.0117\ m$ $y = 0.0366\ m$ $\psi = 6.7945\ \text{deg}$	$v = 0.0113\ m/s$ $\delta = -0.1706\ \text{deg}$
Trial 4	7.7007 s	$x = 1.5329\ m$ $y = -0.5847\ m$ $\psi = 182.4180\ \text{deg}$	$v = 0.5470\ m/s$ $\delta = 6.9881\ \text{deg}$

Table 6.6: CS 1 (Trials 1 to 4): Comparison of Parking Results Analysis.

Hence, by keeping Q_p constant as per trial 3 and R_p, R_t as in Table 6.4, the next cluster of trials (reported in Table 6.7) aims at increasing the penalization on the yaw rate, i.e. the third diagonal value of Q_t is sequentially reduced.

Trial	Q_t	**Comment**		
5	`diag`([1 5 70])	Invalid: collisions	$\psi = -1.4056$ deg	$\delta = -19.0421$ deg
6	`diag`([1 5 50])	Invalid: collisions	$\psi = -11.4703$ deg	$\delta = 27.0794$ deg
7	`diag`([1 5 25])	Invalid: collisions	$\psi = -1.3970$ deg	$\delta = 8.2680$ deg
8	`diag`([1 5 15])	Invalid: collisions	$\psi = 6.0145$ deg	$\delta = -15.5749$ deg
9	`diag`([1 5 5])	Invalid: collisions	$\psi = 4.9160$ deg	$\delta = -10.2849$ deg

Table 6.7: CS 1 (Trials 5 to 9): Tuning of the Terminal Weight Matrices.

With this set of trials it was possible to prove that even though the simulation seems to be quite good (Figure 6.5), while turning the car collides with the obstacles. Therefore, by following a similar reasoning as done for the set $1-4$ and by considering that the goal of the last cluster was to obtain a good result by penalizing $\dot{\psi}$, the new objective is to increase the penalization on the steering angle δ to have a smoother steering during the parking manoeuvre thus tuning the R_t weight matrix by keeping Q_p, R_p and Q_t as in trial 9.

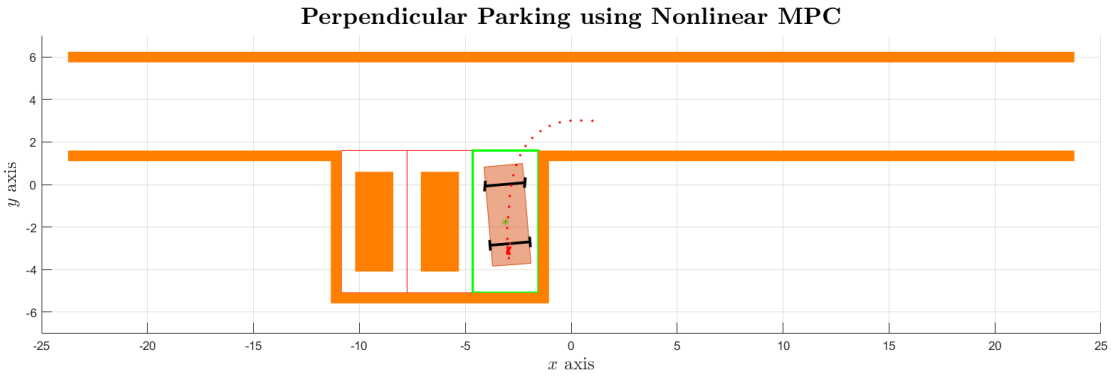


Figure 6.5: CS 1: Ego Vehicle Behaviour at the End of Trial 9.

The weight matrices values for the upcoming simulations are tabulated in Table 6.8, from where it will be clear that with trial 12 satisfactory results are reached (Figure 6.6). The complete *Parking Results Analysis* is reported in Table 6.9.

Trial	R_t	**Comment**
10	$0.08 * \texttt{eye}(2)$	Valid: no collisions $dist_{min} = 0.1091$ $\texttt{ExitFlag} = -2$
11	$0.06 * \texttt{eye}(2)$	Invalid: collisions
12	$0.04 * \texttt{eye}(2)$	Valid: no collisions $dist_{min} = 0.1000$ $\texttt{ExitFlag} = 0$
13	$0.02 * \texttt{eye}(2)$	Invalid: collisions

Table 6.8: CS 1 (Trials 10 to 13): Tuning of the Terminal Weight Matrices.

Description	**Value**
Result	Valid, no collisions.
Minimum distance to obstacles	$dist_{min} = 0.10\ m$
Optimization `ExitFlag`	0
Elapsed time for `nlmpcmove`	$14.9004\ s$
Final states error	$x = 0.0021\ m$ (highlighted in Fig. 6.6)
	$y = 0.0170\ m$ (highlighted in Fig. 6.6)
	$\psi = 0.9533$ deg
Final control inputs	$v = 0.1045\ m/s$
	$\delta = -23.3758$ deg

Table 6.9: CS 1 (Trial 12): Parking Results Analysis.

It is worth remarking that the speed in the final control inputs is not the speed at the end of the simulation, rather it is the last speed value that is reached by the vehicle before halting at the end of the simulation.

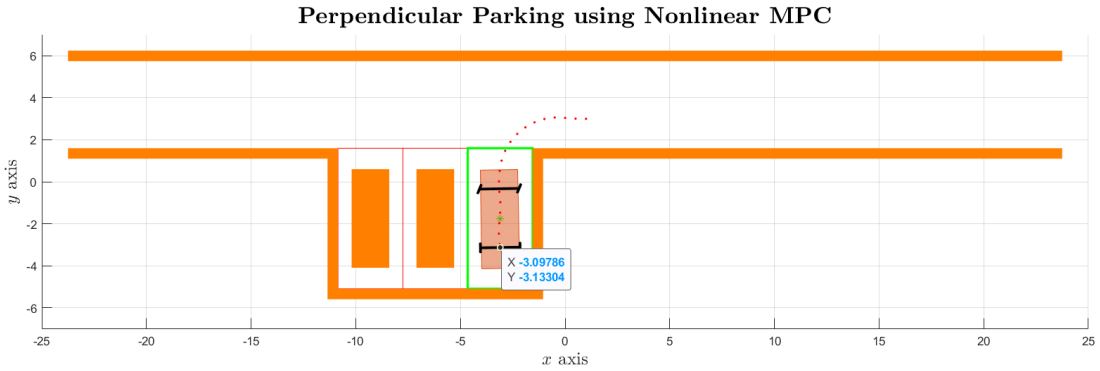


Figure 6.6: CS 1: Valid Result at Trial 12.

A comprehensive representation of the time evolution of the ego vehicle states and control inputs, spanning from 1 s to 36 s, with $h_p = 35\ s$, can be obtained by examining Figures 6.7 and 6.8. Fig. 6.7 shows the time evolution of the ego vehicle states from the initial to the target pose along with the previously discussed

errors, while Fig. 6.8 illustrates the satisfaction of constraints by both the speed v and the steering angle δ. During the initial non-turning phase, the speed remains constant (negative for backward motion), and minor adjustments are made to the steering wheel. However, when turning commences and then the ego vehicle aligns itself within the parking spot, both the speed and steering angle exhibit pulsating behaviour. The last plot shows also that the vehicle achieves a steering angle of $\delta = \pm 45$ deg during the manoeuvre. Although this angle may be relatively large and unsuitable for most normal driving situations, it is necessary to achieve a sharp turn to navigate the 90-degree curve and reach the parking spot. Additionally, the minor speed fluctuations after the 25th second correspond to the final adjustments made by the vehicle to achieve the optimal parking position.

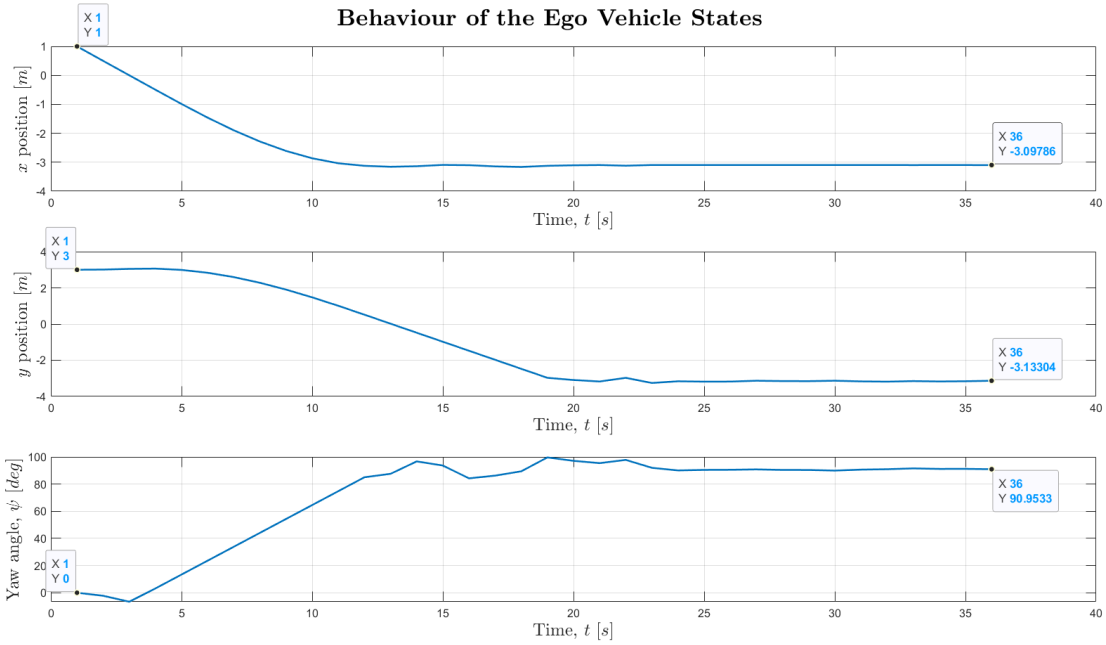


Figure 6.7: CS 1: Behaviour of the Ego Vehicle States at Trial 12.

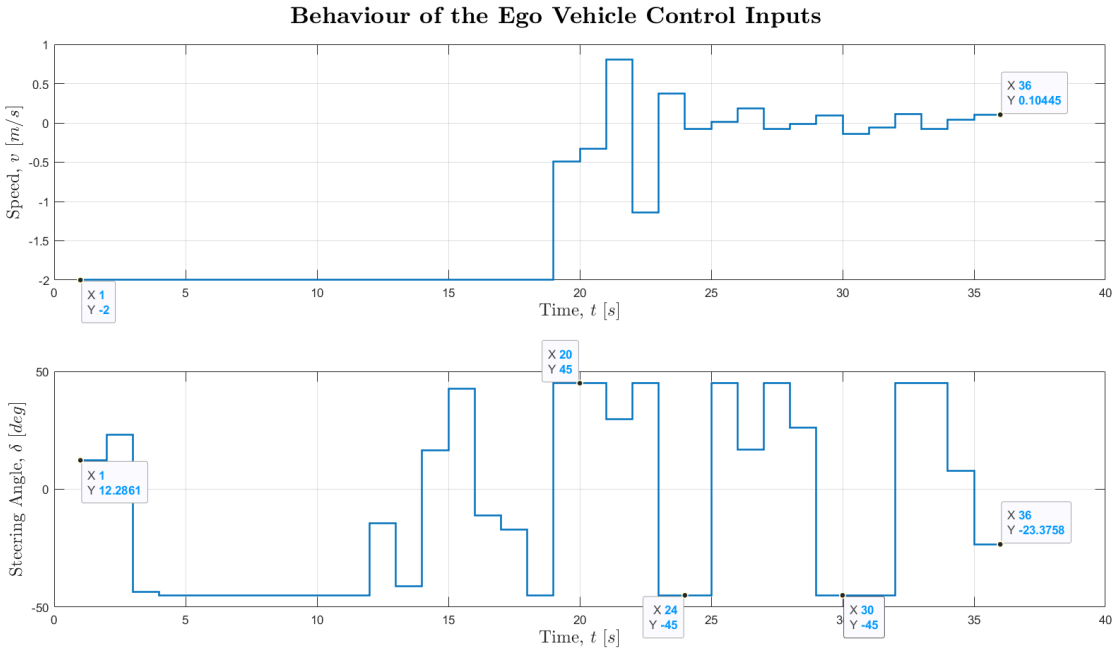


Figure 6.8: CS 1: Behaviour of the Ego Vehicle Control Inputs at Trial 12.

6.2 Perpendicular Parking using Multistage Nonlinear MPC

The second case study aims to perform a perpendicular parking manoeuvre utilizing a Multistage NMPC through a MATLAB® script (Appendix J) and a SIMULINK® model. The `Vehicle Path Planner System` block (Figure 6.9) is used to simulate a VPP system that plans a collision-free trajectory from `egoInitialPose` to `egoTargetPose` using a msNMPC [23].

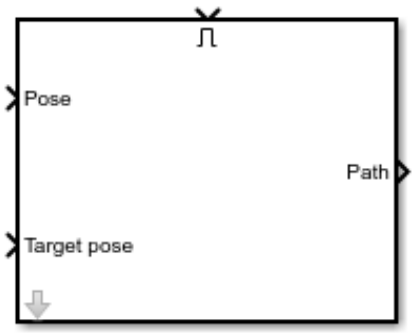


Figure 6.9: VPP System Block as per The MathWorks, Inc. Documentation.

Furthermore, it is noteworthy that this case study is much simpler and more straightforward than the previous case study since the VPP system already contains a pre-built msNMPC controller that requires only data input from the script. Therefore, with the parking environment being the same as in Subsec. 6.1.1 and the ego vehicle consideration in Subsec. 6.1.2, the focus can be directed on configuring the Vehicle Path Planner System block (Subsection 6.2.1).

6.2.1 Vehicle Path Planner System Block Configuration

Specifically, a modified version (compared to the The MathWorks, Inc. one) of the SIMULINK® `Vehicle Path Planner System` block is employed to realise the simulation. The inputs are the same: `egoInitialPose`, `egoTargetPose` and enabling signal[12]; the outputs are displayed in Figure 6.10. The parameters received by the VPP block are shown in Figure 6.11, which are all obtained from the `main.m` MATLAB® script.

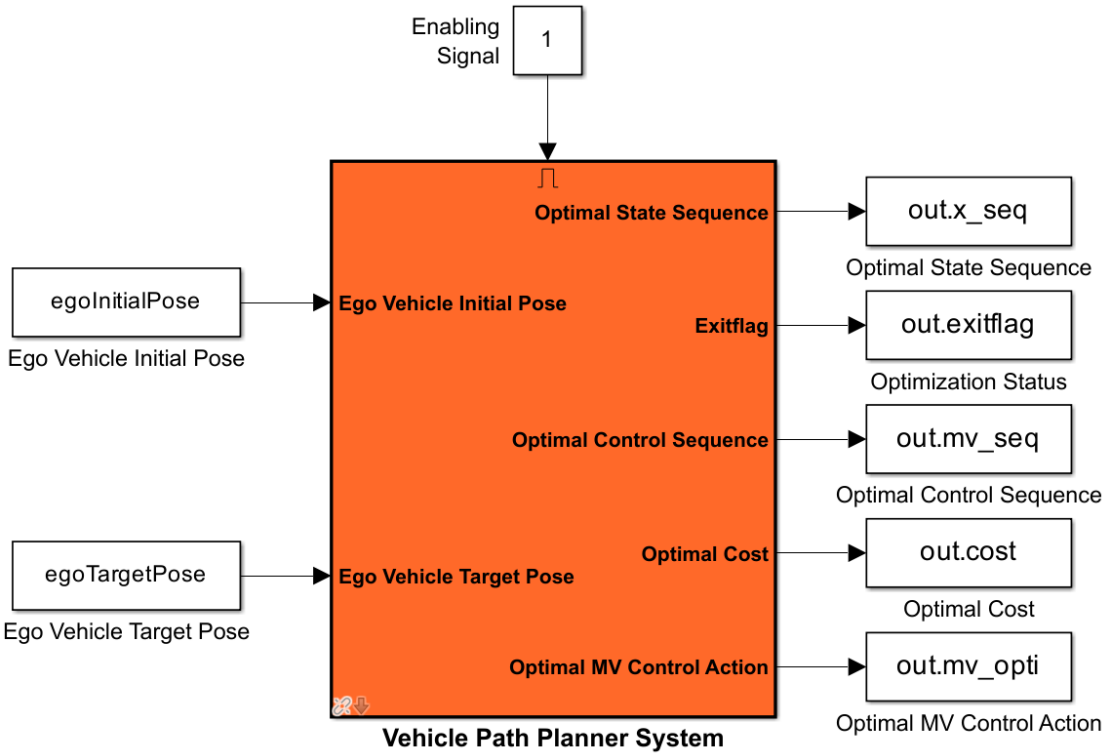


Figure 6.10: VPP SIMULINK® Model.

[12]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/mpc/ref/vehiclepathplannersystem.html`.

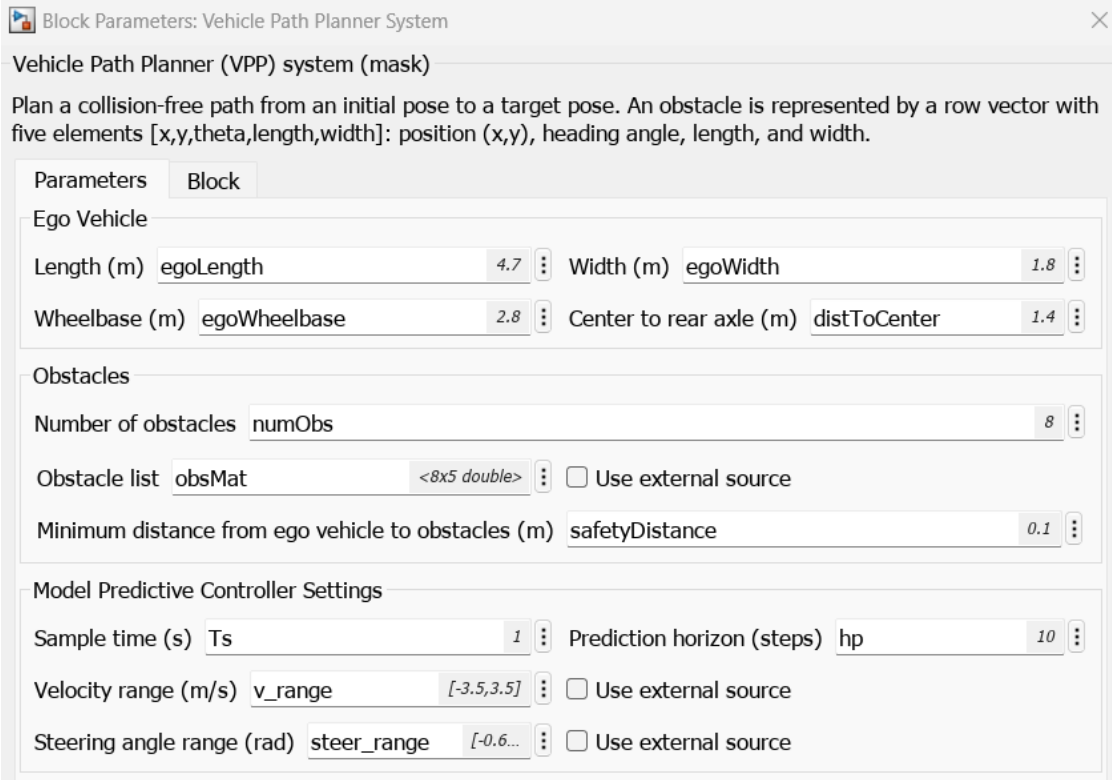


Figure 6.11: VPP System Block Parameters.

To enable additional outports, the VPP must be navigated until reaching the Multistage NMPC Controller block (2nd level inside the starting block), as shown in Figure 6.12.

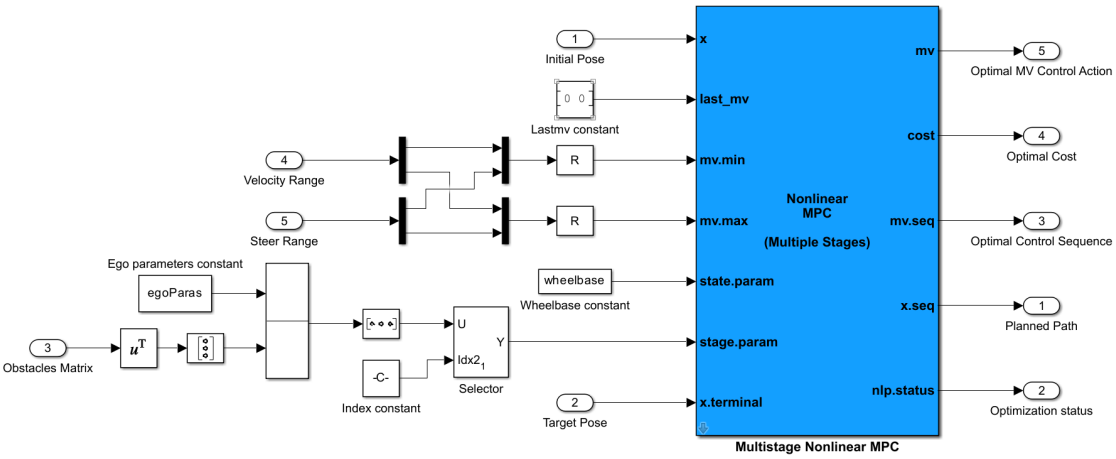


Figure 6.12: msNMPC Controller Model.

The msNMPC Controller block simulates a multistage nonlinear model predictive controller and at each interval, the block computes optimal control moves by solving a nonlinear programming problem in which different cost functions and constraints are defined for different predictions stages [24]. To enable additional outports, it is sufficient to open the block parameters window and tick the needed outputs. Those enabled for this CS are categorized as follows:

- **Required**:
  - Optimal MV Control Action `out.mv_opti`. If the solver converges to a local optimum solution, i.e. if `out.exitflag` > 0, then `out.mv_opti` contains the optimal solution.
- **Additional**:
  - Objective Function Cost `out.cost`, enabled via the *Optimal Cost* parameter. The cost quantifies the degree to which the controller has achieved its objective and this is meaningful only when `out.exitflag` ≥ 0.
  - Optimization Status `out.exitflag` be positive (convergence to optimal solution), null (no convergence to optimal solution) or negative (solver failed).
- **Optimal Sequences**:
  - Optimal MV Sequence `out.mv_seq`, the first h_p rows contain the computed optimal MV values from current time k to $k + h_p - 1$, while the first row contains the current MV values.
  - Optimal Prediction Model State Sequence `out.x_seq`, the first row contains the current estimated state values, while the next h_p rows contain the calculated optimal state values from $k + 1$ to $k + h_p$.

6.2.2 Controller Simulation in MATLAB® and Simulink®

To simulate the controller in both MATLAB® and Simulink® the following design parameters are needed and chosen as from the The MathWorks, Inc. example:

- Sample Time $T_s = 1\ s$.
- Prediction Horizon $h_p = 10\ s$.

Furthermore, as there are no weight matrices to tune in the current CS (as they are integrated within the msNMPC controller block), and the simulation outcomes are already collisions-free, the only remaining objective is to obtain an

optimal solution, achievable by modifying the constraints on the inputs. Loosening these constraints can enhance the feasibility of the solution. Additionally, due to collision-free outcomes, the *Parking Results Analysis* described in Subsec. 6.1.3 is shorter since the first two points are not evaluated. Moreover, as the `nlmpcmove` is not present in the script anymore, the simulation time is now computed using the `Simulink.SimulationMetadata` class[13]. Specifically, the field `TotalElapsedWallTime`[14] (of the `TimingInfo` property) is considered, which is given by the sum of the following three fields, measured in seconds $[s]$:

- `InitializationElapsedWallTime`, time spent before simulation execution.
- `ExecutionElapsedWallTime`, time spent during simulation execution.
- `TerminationElapsedWallTime`, time spent after simulation execution.

Concerning the simulations, these have been carried out starting from the bounds provided by the The MathWorks, Inc. example. Thus, initially the bounds on the speed have been varied while those on the steering angle have been kept fixed at ± 30 deg (Table 6.10).

Trial	**Speed Range** $[m/s]$	**Comment**
1	$[-2.5,\ 2.5]$	`ExitFlag` $= -2$
2	$[-3.0,\ 3.0]$	`ExitFlag` $= -2$
3	$[-3.5,\ 3.5]$	`ExitFlag` $= -2$
4	$[-4.0,\ 4.0]$	`ExitFlag` $= -2$

Table 6.10: CS 2 (Trials 1 to 4): Loosening of the Speed Bounds.

After this first cluster of simulations, it is possible to state that the trial in which the car performed the best, i.e. the car travels more in the parking spot before the simulations ends (Figure 6.13), is in trial 3; while the worst in trial 4. The Parking Results Analysis are reported in Table 6.11.

[13]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/simulink/slref/simulink.simulationmetadata-class.html`.

[14]As for the value of `nlmpcmove`, also this value is influenced by many factors external to the mere script/model.

Trial	**Description and Value**			
	Elapsed simulation time	Final states error		
Trial 1	5.6244 s	$x = 0\ m$	$y = 1.8490\ m$	$\psi = -14.7675\ \text{deg}$
Trial 2	6.0894 s	$x = 0\ m$	$y = 1.4051\ m$	$\psi = -11.3564\ \text{deg}$
Trial 3	5.7216 s	$x = 0\ m$	$y = 0.8835\ m$	$\psi = -7.3862\ \text{deg}$
Trial 4	3.8931 s	$x = 0\ m$	$y = 3.8142\ m$	$\psi = -56.5411\ \text{deg}$

Table 6.11: CS 2 (Trials 1 to 4): Comparison of Parking Results Analysis.

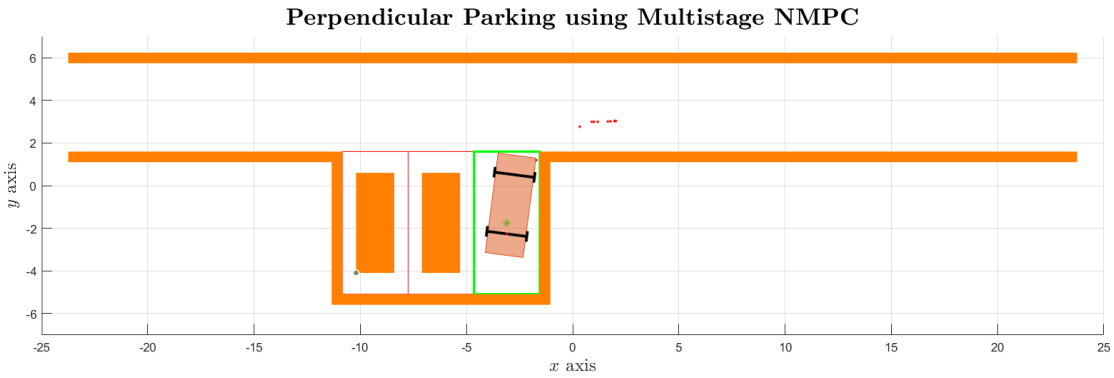


Figure 6.13: CS 2: Ego Vehicle Behaviour at the End of Trial 3.

It follows that trial 3 should represent a good compromise and so, the speed range of this simulation is kept to proceed with the relaxation of the steering angle range because, as also explained for CS 1, a high steering angle is required to turn sharp angles. For this reason, a simulation is run (trial 5) with a steering angle range $[-\pi/4,\ \pi/4]$ and the results are collected in Table 6.12.

Description	**Values**
Optimization `ExitFlag`	2
Optimal MV control action	$v = 0.5264\ m/s$
	$\delta = 0.3591\ \text{deg}$
Objective function cost	≈ 29
Elapsed simulation time	7.9835 s
Final states error	$x = 0\ m$
	$y = 0\ m$
	$\psi = 0\ \text{deg}$
Final control inputs	$v = 0\ m/s$
	$\delta = 0\ \text{deg}$

Table 6.12: CS 2 (Trial 5): Parking Results Analysis.

However, as discussed in CS 1, a steering angle $\delta = \pm 45$ deg is relatively large and unsuitable for many driving situations while a suitable (and more real) range is $30 \div 40$ deg. Therefore, another simulation is attempted (trial 6) with a steering angle range $[-\pi/5,\ \pi/5]$ with results collected in Table 6.13.

Description	**Values**
Optimization `ExitFlag`	1
Optimal MV control action	$v = 0.7574\ m/s$
	$\delta = 0.6000$ deg
Objective function cost	≈ 32
Elapsed simulation time	$4.2473\ s$
Final states error	$x = 0\ m$
	$y = 0\ m$
	$\psi = 0$ deg
Final control inputs	$v = 0\ m/s$
	$\delta = 0$ deg

Table 6.13: CS 2 (Trial 6): Parking Results Analysis.

Hence, by considering all the results (apart from the simulation time), this second simulation is a bit worse than the previous one. However, an angle of 36 deg is closer to the reality and so it is chosen as final value for this case study. The vehicle behaviour is shown in Figure 6.14, while in Figures 6.15 and 6.16 it is possible to see the time evolution of the states and of the control inputs, respectively.

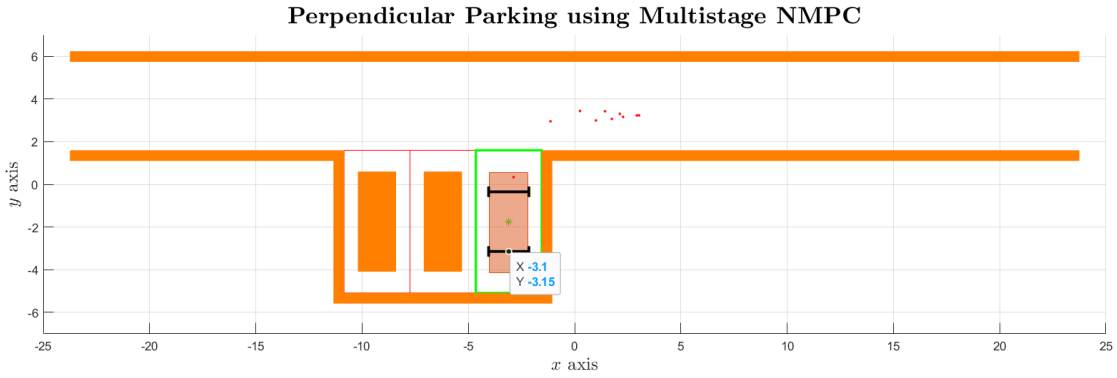


Figure 6.14: CS 2: Valid Result at Trial 6.

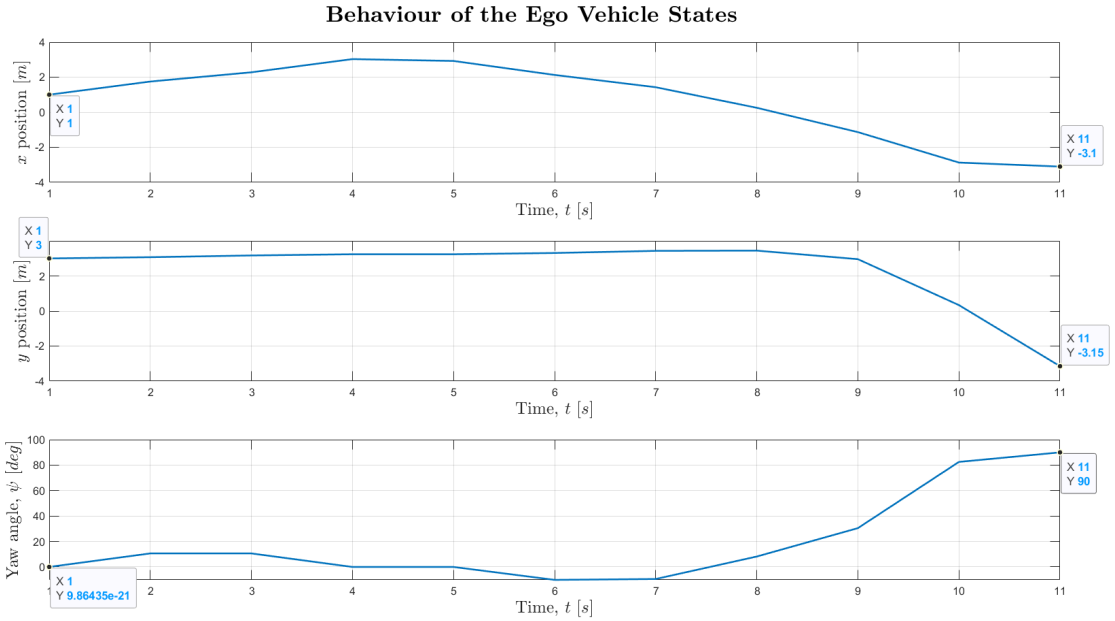


Figure 6.15: CS 2: Behaviour of the Ego Vehicle States at Trial 6.

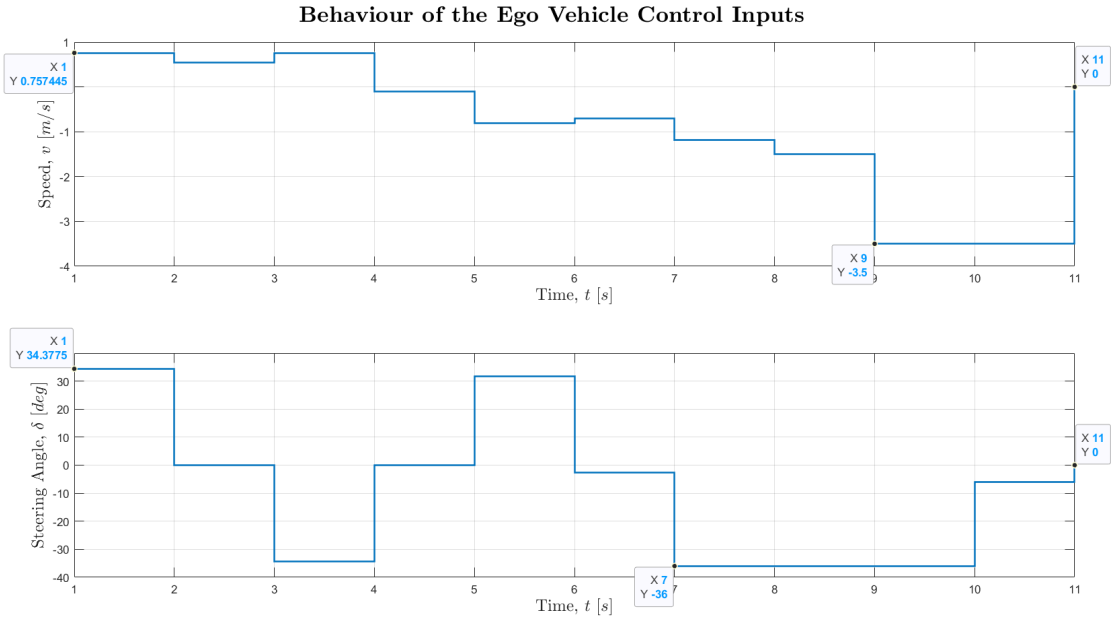


Figure 6.16: CS 2: Behaviour of the Ego Vehicle Control Inputs at Trial 6.

Based on the plots, it is evident that the ego vehicle to navigate into the parking spot performs a wider curve: it slightly accelerates (positive velocity in the speed plot of Fig. 6.16) thus to increase both its x and y positions. The x coordinate

increases more prominently as shown by the hump at the beginning of the first plot in Fig. 6.15, then it starts the reverse motion to drive into the parking spot. As it is clear from the plot, the vehicle does not execute any additional *positioning* manoeuvres to align itself at the center of the parking space, unlike in the CS 1. Furthermore, the simulation results demonstrate that the overall process is both faster and more precise than in the simple NMPC scenario. This approach is also more straightforward as only a few parameters need to be defined in the MATLAB® script, and there is no need to generate many function scripts as everything is handled within the msNMPC block. It is also good to point out that with this approach it is possible to obtain an `ExitFlag` = 1 (positive integer: optimal solution found), that was not possible to obtain in CS 1.

6.3 Perpendicular Parking using RRT* Planner and NMPC Tracking Controller

The aim of this last case study is to realize a perpendicular parking manoeuvre by path-generation with the RRT* planner algorithm and trajectory-tracking with the NMPC controller. Also for this case, the parking environment is the same as in the previous two case studies as well as the ego vehicle poses and dimensions.

This section starts with a brief overview on the trajectory planning and the RRT, RRT* algorithms and then it proceeds with the implementation of the RRT* planner and the NMPC tracking controller in the MATLAB® environment.

The complete script is reported in Appendix K.

6.3.1 Trajectory Planning and the RRT, RRT* Algorithms

The goal of trajectory planning is to generate the reference inputs to the motion control system which ensures that the robot executes the planned trajectory. The user specifies a number of parameters to describe the desired trajectory. Planning consists of generating a time sequence of the values attained by an interpolating function of the desired trajectory [25].

Many motion planning algorithms exist in literature but these have to be differentiated according to the assumption of whether the workspace under examination is empty or obstacles are present. In this latter case, it is necessary to plan motions allowing the robot to accomplish its task without colliding with with the obstacles.

This is the case of an autonomous vehicle[15] that has to perform a parking manoeuvre without colliding with road-users or cars or other obstacles.

The RRT algorithm is a probabilistic algorithm that can help accomplish the task of autonomous parking without collision. It relies on randomized sampling of the configuration space and memorizes the samples that do not cause a collision between the robot and the obstacles. This algorithm is an example of a *single-query*[16] probabilistic planner that incrementally expands the tree, T, using randomized procedures.

The algorithm first generates a random configuration, q_{rand}, according to a uniform probability distribution. When a configuration q_{near} in T, close to q_{rand} is found, a new candidate configuration, q_{new}, is generated at a distance δ from q_{near} on the segment joining q_{near} to q_{rand}. A collision check is then run to verify that both q_{new} and the segment belong to C_{free}. If they do, T is expanded by incorporating q_{new} and the segment.

The RRT is designed for efficiently searching non-convex high-dimensional spaces and it is incrementally constructed to reduce the expected distance of a randomly-chosen point to the tree. These properties make such algorithm particularly well-suited for path planning problems involving obstacles and nonholonomic constraints. It can be regarded as a tool for generating open-loop trajectories for nonlinear systems with state constraints [26].

The RRT*, on the other hand, is nothing but an extension of the RRT that addresses some limitations of the original algorithm, like the non-convergence to an optimal value. Instead, RRT* is asymptotically optimal, meaning that it guarantees that the solution found approaches the optimal solution as the number of iterations goes to infinity.

In summary, RRT is a simple and efficient algorithm for addressing feasible-path-finding problem in high-dimensional configuration spaces, while RRT* is just an extension of RRT that is asymptotically optimal and finds higher quality solutions.

[15]Considered as a *wheeled* robot and which kinematic constraints arising from the pure rolling of the wheels are referred to as *nonholonomic* constraints.

[16]Single-query probabilistic methods aim at solving particular cases of motion planning problems and they explore only a subset of the free configuration space, C_{free} relevant for solving the problem.

6.3.2 Path Planning from RRT* in MATLAB®

To generate a suitable algorithm that involves both the trajectory planning and the tracking controller, the first thing to develop concerns the path planning algorithm. Therefore, the state space model for the planner is configured (App. K, lines $11 - 15$) by considering the ego vehicle states, Eq. (6.1). Here, the bounds for the ego vehicle states have to be carefully chosen so to drive the generated tree inside the available parking spot.

The state space model is built via the state space for Reeds-Shepp vehicles[17], `stateSpaceReeds-Shepp`[18] (Navigation Toolbox, Motion Planning), which stores parameters and states in the Reeds-Shepp state space constituted by state vectors of the kind: $[x\ y\ \theta]$, with x and y Cartesian coordinates, and θ orientation angle. Here, a `MinTurningRadius`[19] of 2.3 m is chosen since the vehicle is supposed to turn a very sharp curve with the smallest possible amount of interventions at the steering wheel.

Once the state space model is defined, the planner requires a customized *state validator* (Appendix L) to enable the collision checking between ego vehicle and obstacles. Here, after the configuration of the custom state validator for RRT* (made by The MathWorks, Inc.), the parking environment obstacles are generated in a way similar to how they were generated in the `helperSLVisualizerParking` function (App. B) for the previous two case studies.

Next, the configuration of the path planner is done via the `plannerRRTStar`[20] object (Navigation Toolbox, Motion Planning) that creates an asymptotically-optimal RRT planner, the RRT* (App. K, lines $17 - 20$). Then, by using the configured path planner, it is possible to plan the path from `egoInitialPose` to `egoTargetPose` (App. K, lines $22 - 23$).

Finally, from line 25 to line 31 of App. K the tree expansion (*yellow line* in

[17]Reeds-Shepp vehicles are wheeled vehicles with a steering mechanism designed to minimize the turning radius and maximize the vehicle manoeuvrability. These vehicles are often used in applications where precise manoeuvring is required, such as autonomous vehicles.

[18]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/nav/ref/statespacereedsshepp.html?searchHighlight=stateSpaceReedsShepp&s_tid=srchtitle_stateSpaceReedsShepp_1`.

[19]Minimum turning radius, in meters, the vehicle can cover with a maximum steer in a single direction.

[20]For more information, please refer to the documentation provided by The MathWorks, Inc. at `https://it.mathworks.com/help/nav/ref/plannerrrtstar.html?searchHighlight=plannerRRTStar&s_tid=srchtitle_plannerRRTStar_1`.

Figure 6.17) in the parking environment and trajectory generated (*blue line* in Figure 6.17) are plotted in the environment.

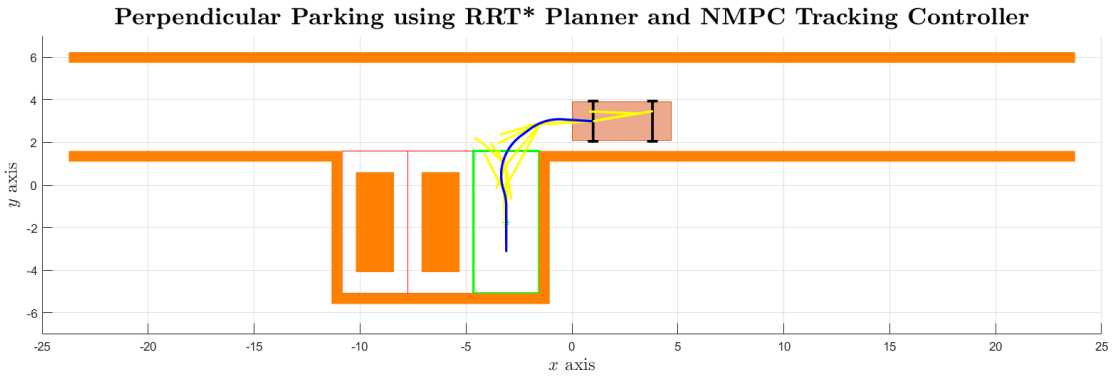


Figure 6.17: CS 3: Tree Expansion and Trajectory Generation in the Parking Environment.

6.3.3 Design of the NMPC Tracking Controller

The NMPC tracking controller design is done similarly to how it has been discussed in Subsec. 6.1.3. The design parameters this time are (Table 6.14):

Parameter	**Value** $[s]$
Sample Time T_s	0.1
Prediction Horizon h_p	10
Control Horizon h_c	10

Table 6.14: CS 3: NMPC Tracking Controller Design Parameters Values.

Instead the constraints on the control inputs are the same as in CS 1, i.e. $v \in [-2,\ 2]\ m/s$ and $\delta \in [-\pi/4,\ \pi/4]\ rad$.

On the other hand, a substantial difference with CS 1 lies in how the weight matrices are handled. In the former example, a custom cost function was used, this time instead a standard cost function is employed and thus, there is no more distinction between process and terminal weight matrices for the output and manipulated variables, but just between output and manipulated variables weight matrices (App. K, lines $46 - 47$), with respect to which the tuning is carried on.

6.3.4 Controller Simulation in MATLAB®

To simulate the controller in MATLAB®, the first thing to do is to specify the initial values for both the ego vehicle state and the control inputs and then, build a `MEX` function for simulating the controller and speed up the simulation (App. K, lines $56 - 58$) since it can take quite a long time.

Then, some variables are initialized before the simulation, like the history of the plant states and control inputs, besides of setting the duration of the parking manoeuvre and the computation of number of steps (App. K, lines $60 - 62$).

Finally, the controller is run in closed-loop and the simulation results are plotted (App. K, lines $63 - 73$):

- Figure 6.18 shows the accomplishment of the manoeuvre by the ego vehicle.
- Figure 6.19 shows the goodness of the tracking controller, i.e. its ability to track the desired trajectory.
- Figure 6.20 shows the behaviour of the ego vehicle states versus the time steps.
- Figure 6.21 shows the behaviour of the control inputs versus the time steps.

The Parking Results Analysis for this third case study is collected in Table 6.15.

Description	**Values**
Optimization `ExitFlag`	2
Tracking error in infinity norm	$\|x\|_\infty = 0.2007\ m$
	$\|y\|_\infty = 0.0790\ m$
	$\|\psi\|_\infty = 7.3293\ \text{deg}$
Final states error	$x = -0.0341\ m$
	$y = 0\ m$
	$\psi = 0.0179\ \text{deg}$
Final control inputs	$v = 0\ m/s$
	$\delta = 0\ \text{deg}$

Table 6.15: CS 3: Parking Results Analysis.

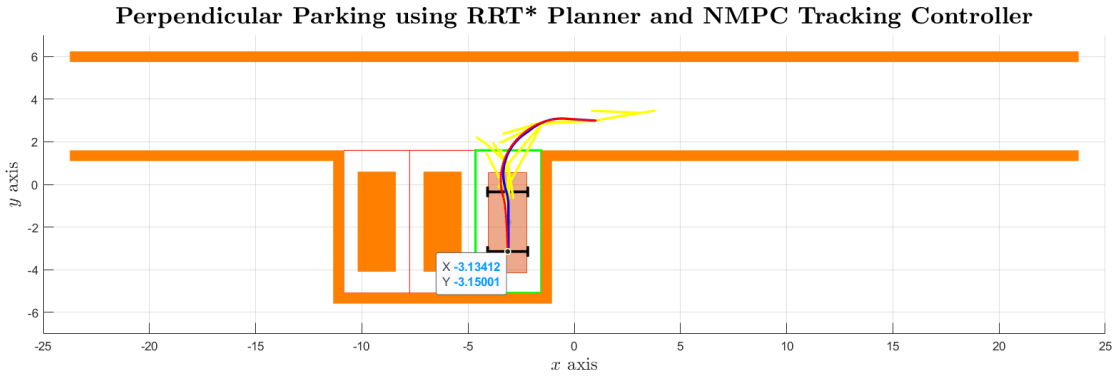


Figure 6.18: CS 3: Ego Vehicle Behaviour at the End of the Simulation.

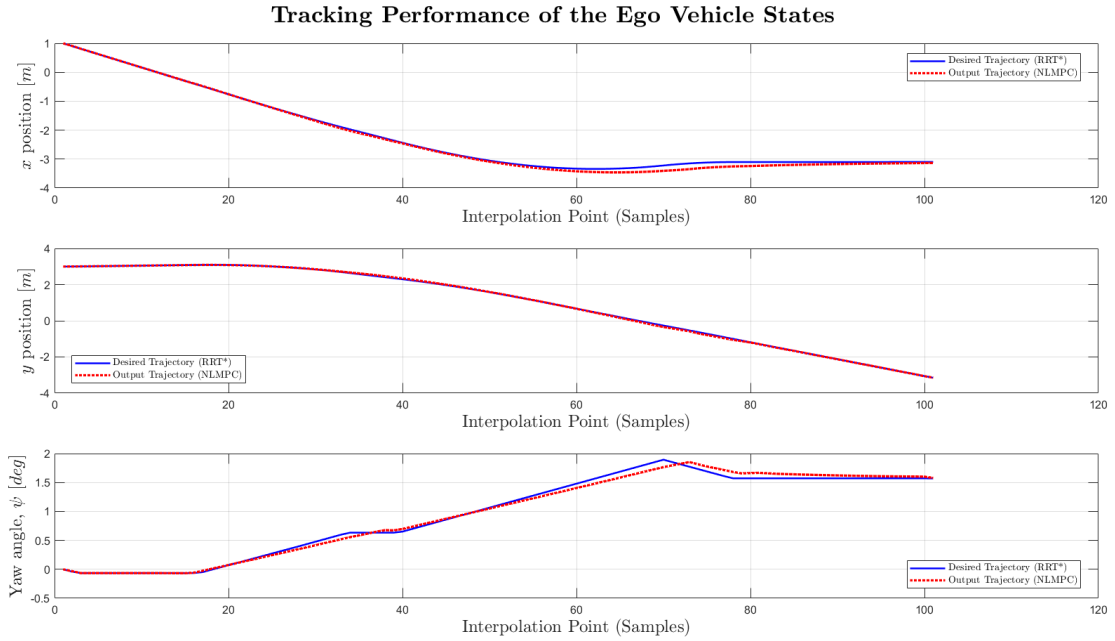


Figure 6.19: CS 3: Goodness of the NMPC Tracking Controller.

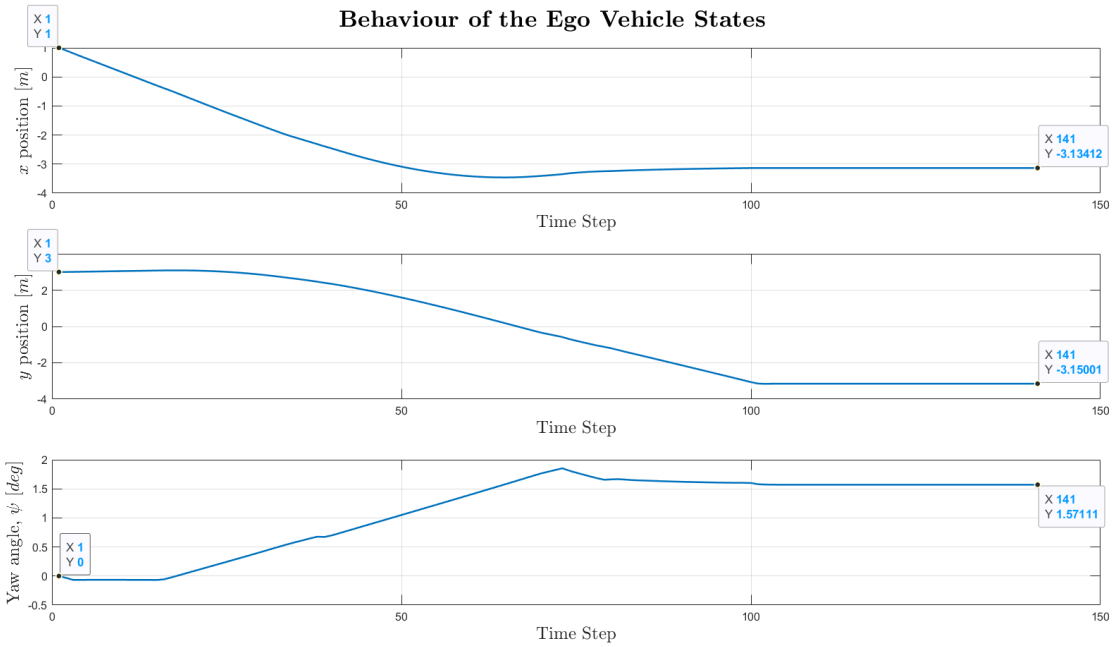


Figure 6.20: CS 3: Behaviour of the Ego Vehicle States.

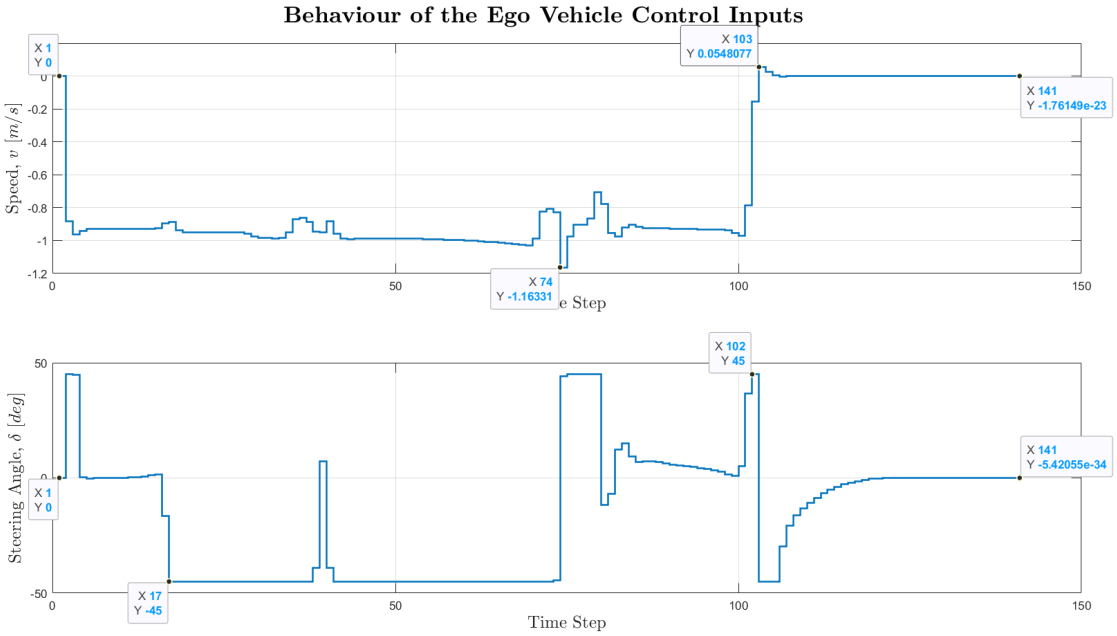


Figure 6.21: CS 3: Behaviour of the Ego Vehicle Control Inputs.

From these last two plots it is possible to say that after the 100th time step, the

parking manoeuvre is practically terminated and what follows are just small corrections in the longitudinal orientation of the vehicle. After these adjustments, the ego vehicle comes to a halt in the correct position as also demonstrated by Fig. 6.18.

In conclusion, the results proved that all three approaches could achieve successful manoeuvres, with some differences in terms of execution time and computational complexity. In particular, the second control strategy (Section 6.2) showed the best overall performance, with the shortest execution time and the most stable control output, besides of being more close to the reality: the steering angle value is in the range $30 \div 40$ deg.

Chapter 7

Conclusion

Autonomous driving technology has surfaced as a viable option to tackle the obstacles of urban transportation in the past few years. Due to the growing need for secure and productive mobility, various organizations and scholars are striving to advance autonomous driving technology. A significant contest in this area is the Bosch Future Mobility Challenge, conducted by the Bosch Engineering Centre Cluj, that seeks to promote inventiveness and originality and create awareness regarding the potential of these emerging technologies.

The goal of this Master's Thesis project is to aid in the progress of autonomous driving technology by concentrating on a crucial element - autonomous parking, with specific emphasis on perpendicular parking. While this manoeuvre may appear effortless for human drivers, it can pose a difficult challenge for self-driving vehicles. This is because it requires accurate perception, planning, and control to manoeuvre the vehicle into a tight parking spot without colliding with other vehicles or obstacles.

To tackle this challenge, The MathWorks, Inc. has already developed a solution for parallel parking. To expand upon this, three separate case studies were conducted to explore various methods to accomplish successful perpendicular parking for autonomous vehicles. The first case study used a Nonlinear Model Predictive Control, which is a type of control algorithm that can optimize the vehicle trajectory based on a predictive model of its dynamics. The second case study used a Multistage NMPC, which is an extension of Nonlinear Model Predictive Control that allows for the use of more accurate nonlinear models for prediction over longer horizons. The third case study used an optimized Rapidly-exploring Random Tree path planning combined with Nonlinear Model Predictive Control tracking controller, which is an approach that uses a probabilistic algorithm to generate a feasible trajectory for the vehicle.

The results showed that all three approaches could achieve successful perpendicular parking manoeuvres, with some differences in terms of the execution time and computational complexity. In particular, the msNMPC approach showed the best overall performance, with the shortest execution time and the most stable control output, besides of being more close to the reality. However, each approach has its advantages and limitations, and the choice of the best method will depend on the specific requirements and constraints of the application.

Overall, this research contributes to the growing body of knowledge on autonomous driving technology by demonstrating the feasibility of using advanced control and planning techniques to solve the problem of perpendicular parking. The hope is that such work can inspire and serve as a valuable resource for researchers and engineers working in this field, providing insights into the design, implementation and evaluation of autonomous driving systems.

Appendix A

Autonomous Parking Algorithm @ BFMC22

```
# PARKING MANOEUVRE
elif PARKING_MANOEUVRE:
    cnt = cnt + 1
    if cnt == 1:
        print("Parking manoeuvre initiated...")
        self.Car_detected = 2
        actual_yaw = self.Yaw
    if cnt <= 10:
    # Drive forward without steering --> straight.
        value = self._straight_correction(actual_yaw)
    elif 17 < cnt < 27:
    # The first slot is empty, the ego vehicle is in correspondence
    # of the second slop, thus it comes to a halt.
        value = 999
    elif 27 <= cnt < 60:  # decreased by 10
    # Steer the wheels and drive backward.
        value = 1000
    elif 60 <= cnt < 93:  # decreased by 20
    # Steer the wheels on the other side and drive backward.
        value = 1001
    elif 93 <= cnt < 130 and not SECOND_PARKING:
    # If there is an obstacle in front, and the ego vehicle is close
    # to this, the LiDAR sensor will brake the car.
        if self.Lidar == 99:
            value = 999
        else:
        # Otherwise, drive forward, without steering, to reach the
        # parking spot's center.
```

```
29              value = self._straight_correction(actual_yaw)
30      elif 93 <= cnt < 130 and SECOND_PARKING:
31      # If the first slot is occupied, drive forward.
32          if cnt < 100:
33              value = self._straight_correction(actual_yaw)
34          else:
35              value = 999
36      elif 130 <= cnt < 155:
37      # Standstill in the parking spot.
38          value = 999
39      elif 155 <= cnt < 170:
40      # Move back to the road.
41          value = 1000
42      elif 170 <= cnt < 180:
43      # Steer to the left and drive forward.
44          value = 2001
45      elif cnt >= 180:
46      # End of the parking manoeuvre.
47          cnt = 0
48          PARKING_MANOEUVRE = False
49          NORMAL = True
50          DISABLE_LIDAR = True
51          print("Parking manoeuvre completed!")
52      else:
53          pass
```

Appendix B

helperSLVisualizeParking.m

```
function helperSLVisualizeParking(pose, steer)

persistent vehicleBodyHandle axesHandle vehicleDims

pose(3) = rad2deg(pose(3));
steer = rad2deg(steer);

if isempty(vehicleDims)
    vehicleDims = vehicleDimensions;
end

if isempty(axesHandle) || ~isvalid(axesHandle)
    fh1 = figure('Visible', 'off');
    fh1.Name        = 'Automated Perpendicular Parking';
    fh1.NumberTitle = 'off';
    fh1.WindowState = 'Maximized';
    axesHandle      = axes(fh1);
    legend off, axis equal, grid on
    title(axesHandle, '\bf Perpendicular Parking using Nonlinear MPC'
    , 'FontSize', 20, 'Interpreter', 'latex')
    xlabel('$x$ axis', 'FontSize', 15, 'Interpreter', 'latex')
    ylabel('$y$ axis', 'FontSize', 15, 'Interpreter', 'latex')
    hold(axesHandle, 'on')

    axesHandle.XLim = [-25 25];
    axesHandle.YLim = [-7 7];

    width = 3.1;
    length = 6.2;

    rectangle('Position', [-10.85 -5.1 width length+0.5], ...,
```

```
                'EdgeColor', 'r')
    rectangle('Position', [-7.75 -5.1 width length+0.5], ...,
                'EdgeColor', 'r')

    rectangle('Position', [-4.65 -5.1 width length+0.5], ....,
                'EdgeColor', 'g', 'LineWidth',2)

    plot(-3.1, -1.75, '*', 'color', 'g')

    obstacles = createObstacles();
    for ct = 1:numel(obstacles)
        show(obstacles{ct})
    end

    ax = gca(fh1);
    upperRoadLine = ax.Children(1);
    upperRoadLine.LineStyle = 'none';
    curbside = ax.Children(2);
    curbside.LineStyle = 'none';
end


if isempty(vehicleBodyHandle) || any(~isvalid(vehicleBodyHandle))
    vehicleBodyHandle = helperPlotVehicle(pose, vehicleDims, ...,
                        steer, 'Parent', axesHandle);
else
    vehicleShapes = helperVehiclePolyshape(pose, vehicleDims, steer);
    for n = 1:numel(vehicleBodyHandle)
        vehicleBodyHandle(n).Shape = vehicleShapes(n);
    end
end

plot(axesHandle, pose(1), pose(2), '.', 'Color', 'r')

fh1.Visible = 'on';
drawnow('limitrate');

end

function obstacles = createObstacles()

obsLength = 6.2;
egoLength = 4.7;
egoWidth = 1.8;

obs1 = collisionBox(egoWidth, egoLength, 0);
T1 = trvec2tform([-9.3, -1.75, 0]);
obs1.Pose = T1;

```

```
80 obs2 = collisionBox(egoWidth, egoLength, 0);
81 T2 = trvec2tform([-6.2, -1.75, 0]);
82 obs2.Pose = T2;
83
84 obs3 = collisionBox(4*obsLength, 0.5, 0);
85 T3 = trvec2tform([11.35, 1.35, 0]);
86 obs3.Pose = T3;
87
88 obs4 = collisionBox(0.5, obsLength+1, 0);
89 T4 = trvec2tform([-1.3, -2, 0]);
90 obs4.Pose = T4;
91
92 obs5 = collisionBox(10.3, 0.5, 0);
93 T5 = trvec2tform([-6.2, -5.35, 0]);
94 obs5.Pose = T5;
95
96 obs6 = collisionBox(0.5, obsLength+1, 0);
97 T6 = trvec2tform([-11.1, -2, 0]);
98 obs6.Pose = T6;
99
100 obs7 = collisionBox(2*obsLength, 0.5, 0);
101 T7 = trvec2tform([-17.55, 1.35, 0]);
102 obs7.Pose = T7;
103
104 obs8 = collisionBox(47.5, 0.5, 0);
105 T8 = trvec2tform([0, 6, 0]);
106 obs8.Pose = T8;
107
108 obstacles = {obs1, obs2, obs3, obs4, obs5, obs6, obs7, obs8};
109
110 end
```

Appendix C

parkingVehicleStateFcn.m

```
function xdot = parkingVehicleStateFcn(x, u, ref, Qp, Rp, Qt, Rt,
    distToCenter, safetyDistance)

wb = 2.8;

psi = x(3);
v = u(1);
delta = u(2);

xdot = zeros(3, 1);
xdot(1) = v * cos(psi);
xdot(2) = v * sin(psi);
xdot(3) = v / wb * tan(delta);
```

Appendix D

parkingCostFcn.m

```
function cost = parkingCostFcn(X, U, e, data, ref, Qp, Rp, Qt, Rt,
    distToCenter, safetyDistance)

hp = data.PredictionHorizon;

cost = 0;
for idx = 1:hp
runningCost = (X(idx+1,:) - ref) * Qp * (X(idx+1,:) - ref)' + U(idx
    ,:) * Rp * U(idx,:)';
cost = cost + runningCost;
end

terminal_cost = (X(hp+1,:) - ref) * Qt * (X(hp+1,:) - ref)' + U(hp,:)
     * Rt * U(hp,:)';

cost = cost + terminal_cost;

end
```

Appendix E

parkingIneqConFcn.m

```
function cineq = parkingIneqConFcn(X, U, e, data, ref, Qp, Rp, Qt, Rt
    , distToCenter, safetyDistance)

persistent obstacles ego

if isempty(obstacles)

    vdims = vehicleDimensions;
    egoLength = vdims.Length;
    egoWidth = vdims.Width;
    ego = collisionBox(egoLength, egoWidth, 0);

    obsLength = 6.2;

    obs1 = collisionBox(egoWidth, egoLength, 0);
    T1 = trvec2tform([-9.3, -1.75, 0]);
    obs1.Pose = T1;

    obs2 = collisionBox(egoWidth, egoLength, 0);
    T2 = trvec2tform([-6.2, -1.75, 0]);
    obs2.Pose = T2;

    obs3 = collisionBox(4*obsLength, 0.5, 0);
    T3 = trvec2tform([11.35, 1.35, 0]);
    obs3.Pose = T3;

    obs4 = collisionBox(0.5, obsLength+1, 0);
    T4 = trvec2tform([-1.3, -2, 0]);
    obs4.Pose = T4;

    obs5 = collisionBox(10.3, 0.5, 0);
```

```
31      T5 = trvec2tform([-6.2, -5.35, 0]);
32      obs5.Pose = T5;
33
34      obs6 = collisionBox(0.5, obsLength+1, 0);
35      T6 = trvec2tform([-11.1, -2, 0]);
36      obs6.Pose = T6;
37
38      obs7 = collisionBox(2*obsLength, 0.5, 0);
39      T7 = trvec2tform([-17.55, 1.35, 0]);
40      obs7.Pose = T7;
41
42      obs8 = collisionBox(47.5, 0.5, 0);
43      T8 = trvec2tform([0, 6, 0]);
44      obs8.Pose = T8;
45
46      obstacles = {obs1, obs2, obs3, obs4, obs5, obs6, obs7, obs8};
47
48  end
49
50  hp = data.PredictionHorizon;
51  numObstacles = numel(obstacles);
52  allDistances = zeros(hp*numObstacles,1);
53
54  for i = 1:hp
55
56      x = X(i,1) + distToCenter * cos(X(i,3));
57      y = X(i,2) + distToCenter * sin(X(i,3));
58      T = trvec2tform([x,y,0]);
59      H = axang2tform([0 0 1 X(i,3)]);
60      ego.Pose = T*H;
61
62      distances = zeros(numObstacles, 1);
63      for ct = 1:numObstacles
64          [~, dist, ~] = checkCollision(ego, obstacles{ct});
65          distances(ct) = max(dist, -10);
66          allDistances((1+(i-1)*numObstacles):numObstacles*i, 1) =
    distances;
67      end
68  end
69
70  cineq = -allDistances + safetyDistance;
71
72  end
```

Appendix F

Nonlinear Model Predictive Controller

```
Ts = 0.25;
hp = 35;
hc = 35;

Qp = diag([10 10 1]);
Rp = 0.01 * eye(2);
Qt = diag([1 5 5]);
Rt = 0.04 * eye(2);

safetyDistance = 0.1;

maxIter = 40;

mpcverbosity('off');

nx = 3;
ny = 3;
nu = 2;

nlobj = nlmpc(nx, ny, nu);

nlobj.Ts = Ts;
nlobj.PredictionHorizon = hp;
nlobj.ControlHorizon = hc;

nlobj.States(1).Units = "m";
nlobj.States(2).Units = "m";
nlobj.States(3).Units = "rad";
```

```
29
30 nlobj.OutputVariables(1).Units = "m";
31 nlobj.OutputVariables(2).Units = "m";
32 nlobj.OutputVariables(3).Units = "rad";
33
34 nlobj.ManipulatedVariables(1).Units = "m/s";
35 nlobj.ManipulatedVariables(2).Units = "rad";
36
37 nlobj.MV(1).Min = -2;
38 nlobj.MV(1).Max = 2;
39 nlobj.MV(2).Min = -pi/4;
40 nlobj.MV(2).Max = pi/4;
41
42 nlobj.Model.StateFcn = "parkingVehicleStateFcn";
43 nlobj.Jacobian.StateFcn = "parkingVehicleStateJacobianFcn";
44
45 nlobj.Optimization.CustomCostFcn = "parkingCostFcn";
46 nlobj.Optimization.ReplaceStandardCost = true;
47 nlobj.Jacobian.CustomCostFcn = "parkingCostJacobian";
48
49 nlobj.Optimization.CustomIneqConFcn = "parkingIneqConFcn";
50 nlobj.Jacobian.CustomIneqConFcn = "parkingIneqConFcnJacobian";
51
52 nlobj.Optimization.SolverOptions.FunctionTolerance = 0.01;
53 nlobj.Optimization.SolverOptions.StepTolerance = 0.01;
54 nlobj.Optimization.SolverOptions.ConstraintTolerance = 0.01;
55 nlobj.Optimization.SolverOptions.OptimalityTolerance = 0.01;
56 nlobj.Optimization.SolverOptions.MaxIter = maxIter;
57
58 opt = nlmpcmoveopt;
59 opt.X0 = [];
60 opt.MV0 = zeros(hp, nu);
61
62 paras = {egoTargetPose, Qp, Rp, Qt, Rt, distToCenter, safetyDistance
       }';
63 nlobj.Model.NumberOfParameters = numel(paras);
64 opt.Parameters = paras;
```

Appendix G

analyseParkingResults.m

```
function analyseParkingResults(nlobj, info, ref, Qp, Rp, Qt, Rt, ...
    distToCenter, safetyDistance, timeVal)

data.PredictionHorizon = nlobj.PredictionHorizon;
cineq = parkingIneqConFcn(info.Xopt, info.MVopt, [], data, ref, Qp,
   Rp, Qt, Rt, distToCenter, safetyDistance);
if all(cineq <= 0)
    fprintf('Valid results. No collisions.\n')
else
    fprintf('Invalid results. Collisions.\n')
end

minObsDist = min(-cineq + safetyDistance); flag = info.ExitFlag;
fprintf('Minimum distance to obstacles = %.4f (Valid > dist_safety)\n
    ', minObsDist);
fprintf('Optimization exit flag = %d (Successful when > 0)\n', flag);
fprintf('Elapsed time for nlmpcmove = %.4f\n', timeVal);

e1 = info.Xopt(end,1) - ref(1); e2 = info.Xopt(end,2) - ref(2);
e3 = rad2deg(info.Xopt(end,3) - ref(3));
fprintf('Final states error in x, y, and psi: %2.4f, %2.4f, %2.4f\n',
    e1, e2, e3);

vFinal = info.MVopt(end,1); deltaFinal = rad2deg(info.MVopt(end,2));
fprintf('Final control inputs v and delta: %2.4f %2.4f\n', vFinal,
   deltaFinal);

end
```

Appendix H

runParkingAndPlot.m

```
x0 = egoInitialPose';
u0 = [0;0];

if useMex
    [coredata,onlinedata] = getCodeGenerationData(nlobj,x0,u0,paras);
    mexfcn = buildMEX(nlobj,'parkingMex',coredata,onlinedata);
end

if useMex
    tic;
    [mv,onlinedata,info] = mexfcn(x0,u0,onlinedata);
    timeVal = toc;
else
    tic;
    [mv,nloptions,info] = nlmpcmove(nlobj,x0,u0,[],[],opt);
    timeVal = toc;
end

plotAndAnimateParking(info.Xopt, info.MVopt);

analyzeParkingResults(nlobj,info,egoTargetPose,Qp,Rp,Qt,Rt,
    distToCenter,safetyDistance,timeVal);
```

Appendix I

plotAndAnimateParking.m

```
function plotAndAnimateParking(xHistory, uHistory)

xLength = size(xHistory, 1);
uLength = size(uHistory, 1);
if xLength ~= uLength
    xHistory = xHistory';
    uHistory = uHistory';
end

timeLength = size(xHistory, 1);
for ct = 1:timeLength
    helperSLVisualizeParking(xHistory(ct, :), uHistory(ct, 2));
    pause(0.05);
end

figure()
subplot(311), plot(xHistory(:, 1), 'linewidth', 1.5), grid on
xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
ylabel('$x$ position [$m$]', 'FontSize', 15, 'Interpreter', 'latex')
subplot(312), plot(xHistory(:, 2), 'linewidth', 1.5), grid on
xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
ylabel('$y$ position [$m$]', 'FontSize', 15, 'Interpreter', 'latex')
subplot(313), plot(rad2deg(xHistory(:, 3)), 'linewidth', 1.5), grid
    on
xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
ylabel('Yaw angle, $\psi$ [$deg$]', 'FontSize', 15, 'Interpreter', '
    latex')
sgtitle('\bf Behaviour of the Ego Vehicle States', 'FontSize', 20, '
    Interpreter', 'latex')

figure()
```

```
29 subplot(211), stairs(uHistory(:, 1), 'linewidth', 1.5), grid on
30 xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
31 ylabel('Speed, $v$ [$m/s$]', 'FontSize', 15, 'Interpreter', 'latex')
32 subplot(212), stairs(rad2deg(uHistory(:, 2)), 'linewidth', 1.5), grid
       on
33 xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
34 ylabel('Steering Angle, $\delta$ [$deg$]', 'FontSize', 15, '
       Interpreter', 'latex')
35 sgtitle('\bf Behaviour of the Ego Vehicle Control Inputs', 'FontSize'
       , 20, 'Interpreter', 'latex')
```

Appendix J

Perpendicular Parking with Multistage NMPC

```
vdims = vehicleDimensions;
egoLength     = vdims.Length;
egoWidth      = vdims.Width;
egoWheelbase = vdims.Wheelbase;

obsLength = 6.2;
distToCenter = 0.5*egoWheelbase;

egoInitialPose = [1, 3, 0];
egoTargetPose = [-3.1, -3.15, pi/2];

helperSLVisualizeParking(egoInitialPose, 0);

numObs = 8;

% obsMat(a, b) = [x-pos, y-pos, heading angle, length, width]
obsMat = [-9.3       , -1.75, 0,       egoLength,      egoWidth ; ...
          -obsLength, -1.75, 0,       egoLength,      egoWidth ; ...
           11.35     ,  1.35, 0,  4 * obsLength, 0.5            ; ...
          -1.3       , -2   , 0,  0.5          , 1 + obsLength; ...
          -obsLength, -5.35, 0, 10.3          , 0.5            ; ...
          -11.1      , -2   , 0,  0.5          , 1 + obsLength; ...
          -17.55     ,  1.35, 0,  2 * obsLength, 0.5            ; ...
            0        ,  6   , 0, 47.5          , 0.5          ];

Ts = 1;
hp = 10;

```

```
v_range = [-3.5, 3.5];
steer_range = [-pi/5, pi/5];

safetyDistance = 0.1;

Tsim = 1/hp;

open_system("VehiclePathPlannerSystem.slx")
out = sim("VehiclePathPlannerSystem.slx");

out_x_seq = out.x_seq;
out_ExitFlag = out.exitflag;
out_mv_seq = out.mv_seq;
out_mv_opti = out.mv_opti;
out_cost = out.cost;

timeLength = size(out_x_seq, 1);
for ct = 1:timeLength
    helperSLVisualizeParking(out_x_seq(ct,:), 0);
    pause(0.1);
end

figure()
subplot(311), plot(out_x_seq(:, 1), 'linewidth', 1.5), grid on
xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
ylabel('$x$ position [$m$]', 'FontSize', 15, 'Interpreter', 'latex')
subplot(312), plot(out_x_seq(:, 2), 'linewidth', 1.5), grid on
xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
ylabel('$y$ position [$m$]', 'FontSize', 15, 'Interpreter', 'latex')
subplot(313), plot(rad2deg(out_x_seq(:, 3)), 'linewidth', 1.5)
grid on
xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
ylabel('Yaw angle, $\psi$ [$deg$]', 'FontSize', 15, 'Interpreter', '
    latex')
sgtitle('\bf Behaviour of the Ego Vehicle States', 'FontSize', 20, '
    Interpreter', 'latex')

figure()
subplot(211), stairs(out_mv_seq(:, 1), 'linewidth', 1.5), grid on
xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
ylabel('Speed, $v$ [$m/s$]', 'FontSize', 15, 'Interpreter', 'latex')
subplot(212), stairs(rad2deg(out_mv_seq(:, 2)), 'linewidth', 1.5)
grid on
xlabel('Time, $t$ [$s$]', 'FontSize', 15, 'Interpreter', 'latex')
ylabel('Steering Angle, $\delta$ [$deg$]', 'FontSize', 15, '
    Interpreter', 'latex')
sgtitle('\bf Behaviour of the Ego Vehicle Control Inputs', 'FontSize'
    , 20, 'Interpreter', 'latex')

```

```
74 fprintf('1) Optimization exit flag = %d (Successful when positive)\n'
       , out_ExitFlag)
75 if out_ExitFlag > 0
76     fprintf('1.1) Optimal solution for speed [m/s] and steering angle
       [deg]: %2.4f %2.4f\n', out_mv_opti(1), out_mv_opti(2))
77 end
78 fprintf('2) Optimal Cost = %i\n', out_cost)
79 fprintf('3) Elapsed Simulation Time = %.4f\n', out.SimulationMetadata
       .TimingInfo.TotalElapsedWallTime)
80
81 e1 = out_x_seq(end, 1) - egoTargetPose(1);
82 e2 = out_x_seq(end, 2) - egoTargetPose(2);
83 e3 = rad2deg(out_x_seq(end, 3) - egoTargetPose(3));
84 fprintf('4) Final states error in x [m], y [m], and psi [deg]: %2.4f,
       %2.4f, %2.4f\n', e1, e2, e3)
85
86 vFinal = out_mv_seq(end, 1);
87 deltaFinal = rad2deg(out_mv_seq(end, 2));
88 fprintf('5) Final control inputs speed [m/s] and steering angle [deg
       ]: %2.4f %2.4f\n', vFinal, deltaFinal)
```

Appendix K

Perpendicular Parking using RRT* and NMPC

```
vdims = vehicleDimensions;
egoLength = vdims.Length;              egoWidth = vdims.Width;
egoWheelbase = vdims.Wheelbase;        distToCenter = 0.5*egoWheelbase;

obsLength = 6.2;

egoInitialPose = [1, 3, 0]; egoTargetPose = [-3.1, -3.15, pi/2];

Tv = 0.1;    helperSLVisualizeParking(egoInitialPose,0);      pause(1)

xlim = [-5 5];    ylim = [-6 6];      yawlim = [-pi pi];
bounds = [xlim; ylim; yawlim];
stateSpace = stateSpaceReedsShepp(bounds);
stateSpace.MinTurningRadius = 2.3;
stateValidator = parkingStateValidator(stateSpace);

planner = plannerRRTStar(stateSpace, stateValidator);
planner.MaxConnectionDistance = 4;
planner.ContinueAfterGoalReached = true;
planner.MaxIterations = 2e3;

rng(9, 'twister')
[pathObj, solnInfo] = plan(planner, egoInitialPose, egoTargetPose);

f = findobj('Name', 'Automated Perpendicular Parking');
ax = gca(f);
hold(ax, 'on');
plot(ax, solnInfo.TreeData(:,1), solnInfo.TreeData(:,2), 'y.-')
```

```
p = 100;     pathObj.interpolate(p+1);    xRef = pathObj.States;

plot(ax, xRef(:,1), xRef(:,2), 'b-', 'LineWidth',2),      pause(5)

mpcverbosity('off');

nx = 3;      ny = 3;      nu = 2;
nlobjTracking = nlmpc(nx, ny, nu);

Ts = 0.1;    hp = 10;     hc = hp;
nlobjTracking.Ts = Ts;
nlobjTracking.PredictionHorizon = hp;
nlobjTracking.ControlHorizon = hc;

nlobjTracking.MV(1).Min = -2;        nlobjTracking.MV(1).Max = 2;
nlobjTracking.MV(2).Min = -pi/4;     nlobjTracking.MV(2).Max = pi/4;

nlobjTracking.Weights.OutputVariables = [3 3 3];
nlobjTracking.Weights.ManipulatedVariablesRate = [0.1 0.005];

nlobjTracking.Model.StateFcn = "parkingVehicleStateFcnRRT";
nlobjTracking.Jacobian.StateFcn = "parkingVehicleStateJacFcnRRT";

nlobjTracking.Optimization.CustomEqConFcn = "parkingTerminalConFcn";

validateFcns(nlobjTracking, randn(3,1), randn(2,1));

x = egoInitialPose';     u = zeros(2,1);
[coredata, onlinedata] = getCodeGenerationData(nlobjTracking, x, u);
mexfcn = buildMEX(nlobjTracking,'parkingRRTMex',coredata,onlinedata);

xTrackHistory = x;       uTrackHistory = u;       mv = u;
Duration = 14;           Tsteps = Duration/Ts;
Xref = [xRef(2:p+1,:); repmat(xRef(end,:),Tsteps-p,1)];
for ct = 1:Tsteps
    xk = x;
    onlinedata.ref = Xref(ct:min(ct+hp-1,Tsteps),:);
    [mv,onlinedata,info] = mexfcn(xk,mv,onlinedata);
    ODEFUN = @(t,xk) parkingVehicleStateFcnRRT(xk,mv);
    [TOUT,YOUT] = ode45(ODEFUN,[0 Ts], xk);
    x = YOUT(end,:)';
    xTrackHistory = [xTrackHistory x];
    uTrackHistory = [uTrackHistory mv];
end
plotAndAnimateParkingRRT(p, xRef, xTrackHistory, uTrackHistory)
```

Appendix L

parkingStateValidator.m

```
classdef parkingStateValidator < nav.StateValidator

    properties
        Obstacles
        EgoCar
        ValidationDistance
    end

    methods
        function obj = parkingStateValidator(ss)
            obj@nav.StateValidator(ss);
            obj.Obstacles = createObstacles();
            obj.EgoCar = collisionBox(4.7, 1.8, 0);
            obj.ValidationDistance = 0.1;
        end

        function isValid = isStateValid(obj, state)
            isValid = true(size(state,1), 1);
            for k = 1:size(state, 1)
                relPose = [1.4, 0, 0];
                st = robotics.core.internal.SEHelpers.
    accumulatePoseSE2(state(k,:), relPose);
                pos = [st(1), st(2), 0];
                quat = eul2quat([st(3), 0, 0]);
                for i = 1:length(obj.Obstacles)
                    [~, dist] = ...
                    robotics.core.internal.intersect(obj.Obstacles{i
    }.GeometryInternal, ...
                        obj.Obstacles{i}.Position, ...
                        obj.Obstacles{i}.Quaternion,...
                        obj.EgoCar.GeometryInternal,...
```

```
30                          pos, quat, 1);
31                      if dist <= 0.15
32                          isValid(k) = false;
33                          return;
34                      end
35                  end
36              end
37          end
38
39          function [isValid, lastValid] = isMotionValid(obj, state1,
    state2)
40              if ~obj.isStateValid(state1)
41                  isValid = false;
42                  lastValid = nan(1,obj.StateSpace.NumStateVariables);
43                  return
44              end
45              dist = obj.StateSpace.distance(state1, state2);
46              interval = obj.ValidationDistance/dist;
47              interpStates = obj.StateSpace.interpolate(state1, state2,
     [0:interval:1 1]);
48              interpValid = obj.isStateValid(interpStates);
49              if nargin == 1
50                  if any(~interpValid)
51                      isValid = false;
52                  else
53                      isValid = true;
54                  end
55              else
56                  firstInvalidIdx = find(~interpValid, 1);
57                  if isempty(firstInvalidIdx)
58                      isValid = true;
59                      lastValid = state2;
60                  else
61                      isValid = false;
62                      lastValid = interpStates(firstInvalidIdx-1, :);
63                  end
64              end
65          end
66          function copyObj = copy(obj)
67
68          end
69      end
70  end
71
72  function obstacles = createObstacles()
73
74  obsLength = 6.2;     egoLength = 4.7;     egoWidth = 1.8;
75
76  obs1 = collisionBox(egoWidth, egoLength, 0);
```

```
77 T1 = trvec2tform([-9.3, -1.75, 0]);
78 obs1.Pose = T1;
79
80 obs2 = collisionBox(egoWidth, egoLength, 0);
81 T2 = trvec2tform([-6.2, -1.75, 0]);
82 obs2.Pose = T2;
83
84 obs3 = collisionBox(4*obsLength, 0.5, 0);
85 T3 = trvec2tform([11.35, 1.35, 0]);
86 obs3.Pose = T3;
87
88 obs4 = collisionBox(0.5, obsLength+1, 0);
89 T4 = trvec2tform([-1.3, -2, 0]);
90 obs4.Pose = T4;
91
92 obs5 = collisionBox(10.3, 0.5, 0);
93 T5 = trvec2tform([-6.2, -5.35, 0]);
94 obs5.Pose = T5;
95
96 obs6 = collisionBox(0.5, obsLength+1, 0);
97 T6 = trvec2tform([-11.1, -2, 0]);
98 obs6.Pose = T6;
99
100 obs7 = collisionBox(2*obsLength, 0.5, 0);
101 T7 = trvec2tform([-17.55, 1.35, 0]);
102 obs7.Pose = T7;
103
104 obs8 = collisionBox(47.5, 0.5, 0);
105 T8 = trvec2tform([0, 6, 0]);
106 obs8.Pose = T8;
107
108 obstacles = {obs1, obs2, obs3, obs4, obs5, obs6, obs7, obs8};
109
110 end
```

Bibliography

[1] Wikipedia. *Self-driving car*. URL: https://en.wikipedia.org/wiki/Self-driving_car (cit. on p. 1).

[2] Synopsys. *What is an Autonomous Car?* URL: https://www.synopsys.com/automotive/what-is-autonomous-car.html (cit. on pp. 2, 8).

[3] European Parliament and Council of the European Union. «Regulation (EU) 2019/2144 of the European Parliament and of the Council». In: *Official Journal of the European Union* (Dec. 2019). URL: https://eur-lex.europa.eu/eli/reg/2019/2144/oj (cit. on p. 2).

[4] Barbara Wendling (SAE J3016 Standard Committee chairperson). *Amending the automated-driving 'Constitution'*. URL: https://www.sae.org/news/2021/06/sae-revises-levels-of-driving-automation (cit. on p. 3).

[5] Rambus Press. *SAE levels of automation in cars simply explained (+Image)*. URL: https://www.rambus.com/blogs/driving-automation-levels/ (cit. on p. 5).

[6] World Healt Organization. *Road traffic injuries*. URL: https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries (cit. on p. 7).

[7] European Parliament. *Self-driving cars in the EU: from science fiction to reality*. URL: https://www.europarl.europa.eu/news/en/headlines/economy/20190110STO23102/self-driving-cars-in-the-eu-from-science-fiction-to-reality (cit. on pp. 7, 9).

[8] European Commission. *Saving Lives: Boosting Car Safety in the EU*. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:52016DC0787&from=EN (cit. on p. 8).

[9] Synopsys. *What is ADAS?* URL: https://www.synopsys.com/automotive/what-is-adas.html (cit. on p. 8).

[10] European Commission. *Advanced Driver Assistance Systems*. URL: https://road-safety.transport.ec.europa.eu/document/download/93f6a003-83dc-4e66-ae14-852d93236e57_en?filename=ersosynthesis2018-adas.pdf (cit. on p. 8).

[11] Bosch Engineering Center Cluj. *Bosch Future Mobility Challenge*. URL: https://boschfuturemobility.com/ (cit. on p. 12).

[12] The MathWorks, Inc. «Parallel Parking Using Nonlinear Model Predictive Control». In: *MathWorks Documentation* (2023). URL: https://it.mathworks.com/help/mpc/ug/parallel-parking-using-nonlinear-model-predictive-control.html (cit. on pp. 25, 74).

[13] The MathWorks, Inc. «Plan Parallel Parking Path Using Multistage Nonlinear Model Predictive Control». In: *MathWorks Documentation* (2023). URL: https://it.mathworks.com/help/mpc/ug/plan-parallel-parking-path-using-multistage-nonlinear-model-predictive-control.html (cit. on p. 25).

[14] The MathWorks, Inc. «Parallel Parking Using RRT Planner and MPC Tracking Controller». In: *MathWorks Documentation* (2023). URL: https://it.mathworks.com/help/mpc/ug/parallel-parking-using-rrt-planner-and-mpc-tracking-controller.html (cit. on p. 25).

[15] Bosch Sensortec. *Smart sensor: BNO055*. URL: https://www.bosch-sensortec.com/products/smart-sensors/bno055/ (cit. on p. 25).

[16] Rajesh Rajamani. *Vehicle Dynamics and Control*. Ed. by Frederick F. Ling. Mechanical Engineering Series. United States of America: Springer, 2006, pp. 15, 20–33, 95–120 (cit. on p. 32).

[17] Luca Venturi and Krishtof Korda. *Hands-On Vision and Behavior for Self-Driving Cars*. Ed. by Kunal Chaudhari. Birmingham, United Kingdom: Packt Publishing Ltd., 2020, pp. 283, 290–291 (cit. on pp. 52, 54).

[18] Eduardo F. Camacho and Carlos Bordons. *Model Predictive Control*. Ed. by Michael J. Grimble and Michael A. Johnson. Advanced Textbooks in Control and Signal Processing. London, United Kingdom: Springer-Verlag, 1999, p. 1 (cit. on p. 53).

[19] Carlo Novara. *Nonlinear Model Predictive Control*. Slides from the Nonlinear Control and Aerospace Applications course. Politecnico di Torino. 2022 (cit. on pp. 55, 56, 66).

[20] The MathWorks, Inc. «Automated Driving Using Model Predictive Control». In: *MathWorks Documentation* (2021). URL: https://it.mathworks.com/help/mpc/ug/automated-driving-using-model-predictive-control.html (cit. on p. 57).

[21] Basil Kouvaritakis and Mark Cannon. *Model Predictive Control: Classical, Robust and Stochastic*. Ed. by Michael J. Grimble and Michael A. Johnson. Advanced Textbooks in Control and Signal Processing. Switzerland: Springer International Publishing AG Switzerland, 2016, p. 13 (cit. on p. 59).

[22] The MathWorks, Inc. «Vehicle Dimensions». In: *MathWorks Documentation* (2023). URL: https://it.mathworks.com/help/driving/ref/vehicledimensions.html?searchHighlight=vehicleDimensions&s_tid=srchtitle_vehicleDimensions_1 (cit. on p. 73).

[23] The MathWorks, Inc. «Vehicle Path Planner System». In: *MathWorks Documentation* (2023). URL: https://it.mathworks.com/help/mpc/ref/vehiclepathplannersystem.html (cit. on p. 83).

[24] The MathWorks, Inc. «Multistage Nonlinear MPC Controller». In: *MathWorks Documentation* (2023). URL: https://it.mathworks.com/help/mpc/ref/multistagenonlinearmpccontroller.html (cit. on p. 86).

[25] B. Siciliano, L. Sciavicco, L. Villani, G. Oriolo. *Robotics: Modelling, Planning and Control*. Ed. by Michael J. Grimble and Michael A. Johnson. Advanced Textbooks in Control and Signal Processing. London, United Kingdom: Springer-Verlag London Limited, 2010, pp. 161, 469, 523, 543 (cit. on p. 91).

[26] Steve LaValle. «About RRTs». In: *The RRT Page* (2023). URL: http://lavalle.pl/rrt/about.html (cit. on p. 92).