POLITECNICO DI TORINO

MECHATRONIC ENGINEERING

Master Thesis

Lane Keeping Algorithms and Intersection Management on a scaled mock-up Autonomous Vehicle

Supervisor:
Stefano Malan

Candidate:
Luca Di Biase

ACADEMIC YEAR 2023/2024

Contents

Abstract

The interest in autonomous driving has grown in the recent years, such that it is starting to be adopted in many applications, like in research, industries or in student works and competitions, as its use will probably spread more in the future if people will not fear letting go of the wheel.
The main topic of this thesis is the analysis and implementation of control algorithms for a scale model of a self-driving car that has to deal with specific scenarios: straight roads, crosses, curves and stop lines.
The information from the external world will come to the car from a camera, that is needed for the development of lane detection algorithms and to let the vehicle know in which situation it is involved.
The images captured by the device are later processed in such a way that lanes are recognized: from this, the vehicle trajectory can be generated and it will be used as a reference for lane keeping controls.
Hence, both lane detection and lane keeping instructions will be applied to our 1/10 scale on-road vehicle of the Bosch Future Mobility Challenge (BFMC), already provided with the needed hardware and that will constitute our starting point, from which then we will move away in terms of programming language and tasks, as well as of some electronic components.

Chapter 1

Introduction

Nowadays, Autonomous Vehicles (AVs) represent a progress in the transport industry because of their capacity of navigating and making decisions independently of human actions. They are equipped with particular devices like sensors, cameras, GPS, radar, LiDAR and advanced computing systems [1] that collaborate to discern the vehicle environment and let them make the right decision according to the situation they are involved in, since they act in a everyday-scenario, where pedestrians, objects and different road configurations exist.

This evolution improves mobility as, with this new technology, the risk of having or being victim of an accident is reduced because of the reduced possibility of experiencing any distraction or dangerous behavior. In fact, autonomous vehicles sense the environment, thanks to the connection to the Internet, obey traffic guidelines and make quick decisions to guarantee pedestrians and passengers' safety. This, however, still does not exclude the possibility of having an accident, but the percentage for this to happen is smaller than a normal human driving action, where the driver attention typically may be reduced by distraction, by the use of alcohol or drugs or simply by falling asleep at the wheel.

Furthermore, from an economic point of view, the advent of these new means of transport can reduce fuel consumption and transportation costs [2], [3]: according to some experts, in fact, this different type of transportation will become almost 50% cheaper than the normal public transport and this travel cost reduction may lead to a future increased use of these vehicles.

Lately, an increasing focus is on Full Autonomous Vehicles (FAVs), that, according to the Society of Automotive Engineers (SAE), represent the most

advanced form of vehicular automation, because they are able to drive everywhere and under all conditions, without the intervention of human actions and, consequently, without the presence of a steering wheel or of accelerator/brake pedals.
There exist 6 levels of driving automation, each described by a set of capabilities of the vehicle [4] according to the SAE International J3016-Standard:

1. **LEVEL 0 - NO AUTOMATION:** the driver has the complete control over the car. Here, only a few controls are implemented, like the Electronic Stability Control (ESC) and the Emergency Brake Assist (EBA), to provide warnings and momentary assistance.

2. **LEVEL 1 - DRIVER ASSISTANCE:** the driver has the control over the car, but she/he can also be assisted by some assistance systems concerning the steering **or** the braking/accelerating task. Lane centering or adaptive cruise control can be active one at a time.

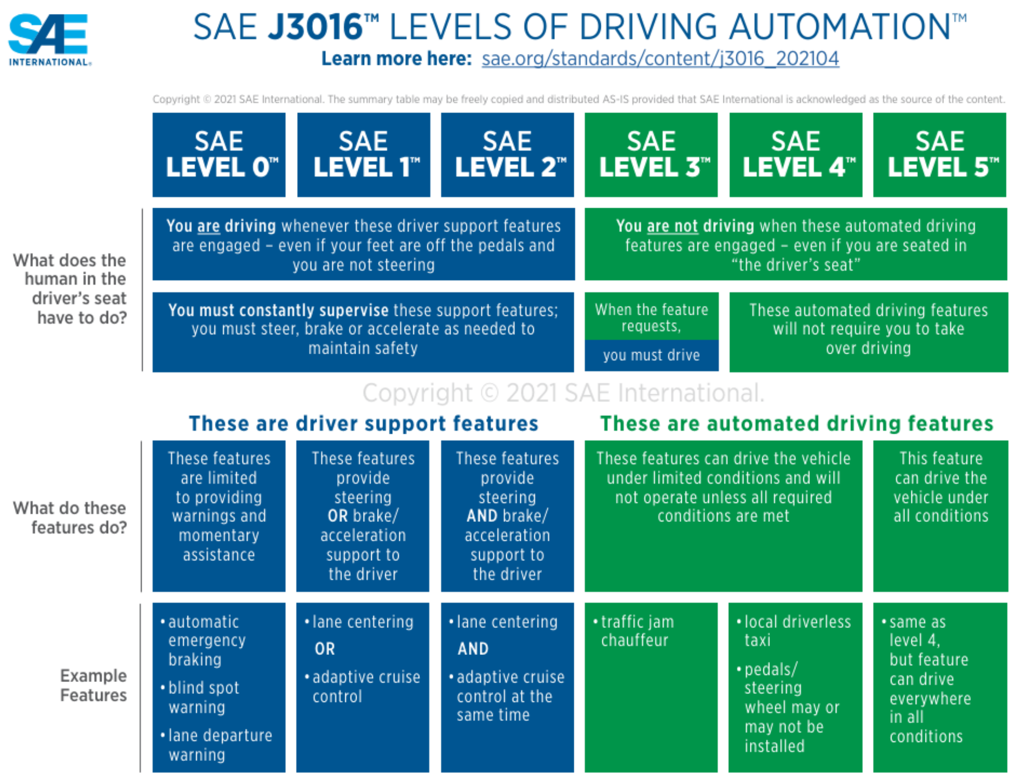
3. **LEVEL 2 - PARTIAL DRIVING AUTOMATION:** this is quite similar to level 1, the only difference is that the driver is assisted in both steering **and** braking/accelerating actions, leading to have the lane centering and the adaptive cruise control work at the same time.

4. **LEVEL 3 - CONDITIONAL DRIVING AUTOMATION:** the driver has the possibility of doing something else while driving thanks to the presence of systems that take over the driving action. However, the driver has to be reactive enough to come back to the driving task when a warning sign is sent from the autonomous systems.

5. **LEVEL 4 - HIGH DRIVING AUTOMATION:** this is similar to level 3, but in this scenario the driver no longer has to be ready to intervene in some driving situations.

6. **LEVEL 5 - FULL DRIVING AUTOMATION:** the vehicle drives on its own in every condition and situation. There is no need to install a steering wheel and throttle or braking pedals since every action is performed autonomously by the car.

These levels are also summed-up in Figure 1.1 [5] with some examples to better understand which actions the driver has to perform or does not have to.


SAE INTERNATIONAL

SAE **J3016**™ LEVELS OF DRIVING AUTOMATION™

Learn more here: sae.org/standards/content/j3016_202104

	SAE LEVEL 0™	SAE LEVEL 1™	SAE LEVEL 2™	SAE LEVEL 3™	SAE LEVEL 4™	SAE LEVEL 5™
What does the human in the driver's seat have to do?	**You <u>are</u> driving** whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering			**You <u>are not</u> driving** when these automated driving features are engaged – even if you are seated in "the driver's seat"		
	You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
	These are driver support features			**These are automated driving features**		
What do these features do?	These features are limited to providing warnings and momentary assistance	These features provide steering **OR** brake/ acceleration support to the driver	These features provide steering **AND** brake/ acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met		This feature can drive the vehicle under all conditions
Example Features	• automatic emergency braking • blind spot warning • lane departure warning	• lane centering **OR** • adaptive cruise control	• lane centering **AND** • adaptive cruise control at the same time	• traffic jam chauffeur	• local driverless taxi • pedals/ steering wheel may or may not be installed	• same as level 4, but feature can drive everywhere in all conditions

Figure 1.1: SAE J3016TM Levels Of Driving Automation.

Autonomous driving positively impacts on the environment, because fuel consumption can be reduced by optimizing the acceleration and braking actions, as well as by establishing a communication among vehicles and the environment, such that congestion zones can be avoided and fuel consumption cut down.
However, even though it seems that automated driving has a variety of advantages, to accept this new style, it is needed to solve several technical, legal and ethic issues [6], that consist in the assurance of road safety, by considering all possible situations (e.g. a pedestrian crosses when the traffic light is green for the car or a vehicle performs an overtake action when other road users are in the nearby), respect of public policies and technical standards.

In this work, we will focus on the description and behavior of a model car on which we will implement autonomous driving algorithms: in particular, to make it move and choose for the best action to perform, two main algorithms will be developed and they will be described in chapters 3 and 4:

1. **Lane Detection Algorithms:** they provide the car with the information about how the road in front of it is, by detecting left and right lane lines, stop lines and pedestrian crossings.
   This first step is crucial for the behavior of the vehicle, as the next controls strictly depend on it.

2. **Lane Keeping Algorithms:** they are important to keep the car between the two lane lines and to avoid any sort of skidding or loss of trajectory.
   The implementation is based on the control of the steering angle and of the vehicle velocity, given its lateral offset and its slope with respect to the road.

The model car, shown in Figure 1.2 and on which we do our studies, is provided by Bosch Future Mobility Challenge (BFMC), a competition among university students who have to realize a full autonomous driving on the scaled model they are given.
However, for this study, we start from the challenge premises to later move away from the competition itself to implement a quite different control, both in terms of programming languages and demands to be met. In particular, we will only make use of the C++ programming language and we will develop specific points to realize a full autonomous driving car in absence of other vehicles, external disturbances and obstacles, focusing on only what the car can meet during its run on the track, as already previously mentioned: straight roads, curves, crosswalks and stop lines.
Furthermore, changes will also be made in the hardware, since some components experienced some troubles in our application.

The contribution of this thesis project is dual: the authors, Chiara Gentile and Luca Di Biase, have continuously collaborated and worked on the topic to satisfy all the requests needed to reach the final one, i.e. the lane keeping. However, a subdivision between the two candidates of the various control parts has been performed, even though they both were directly involved in

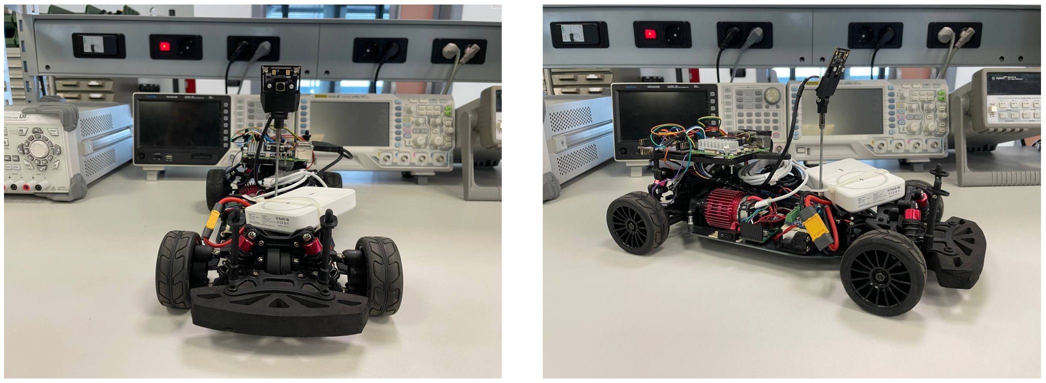

(a)Car from the front view. (b)Car from the lateral view.

Figure 1.2: Car from the front and lateral views.

each other's work to have a general view and a complete knowledge of the system. Chiara Gentile was responsible of the set-up and management of the serial communication between Raspberry Pi and the Nucleo Board, of the image processing algorithms for the frame acquisition and elaboration, and of the definition of the commands to send between the brain and the controller boards to run both the traction and the servo motors.
Luca Di Biase main tasks were the set-up of Raspberry Pi, focusing on the initialisation and installation of the libraries needed to write and execute the different controls, the implementation of the sliding windows algorithm, used to correctly identify at each time the lane lines, and, connected to this, the realization of the control on the lane trajectory, based on the central reference obtained from the just mentioned algorithm.
The final request, i.e. lane keeping, was handled together, but again with a subdivision of the management of the two states: the yaw angle was treated by Luca Di Biase, who focused on the control algorithm based on the inclination of the vehicle with respect to the central lane line. Chiara Gentile, instead, developed the algorithm based on the lateral offset with respect to the same line: all this performed to make the car run on its lane, always staying within the central line and without leaving the area defined by the two lateral lane lines.

1.1 Autonomous Driving Competitions

In recent years, the competitions of autonomous cars have become popular at the universities, where students have the possibility of learning new aspects of the possible-future mobility and developing new scientific and technical skills related to this new area, whose interest is growing up among young engineers.
This is also an opportunity to gaining new experiences and to make new people' acquaintance, from which students can learn by means of an exchange of knowledge and information.
Today, we can count different challenges over the world, that have the main aim to making a car completely autonomous, concerning both real and scaled models: real autonomous cars are obliged to follow specific rules to safeguard the safety of all the people around and working on the vehicles, while challenges based on scaled model cars have their own regulations.
Among this last category, the actual following championships are described:

1. **American Control Conference Self-Driving Car Student Competition**, powered by Quanser [7].
   The challenge highlights critical Control Systems concepts focusing on real-time decisions feedback control systems to obtain a fast and precise driving performance. The car will start from a known position on the track, traverse the outside path and then finish at the finish line, with every action performed by paying attention that the car always stays in the lane.

2. **VDI Autonomous Driving Challenge** [8].
   It takes place in Germany and it uses a model car of scale 1:8 that will autonomously run through a race track that features pit stops, parking actions and/or highway exits. To compete in the challenge, in fact, vehicles have to drive autonomously, complete different parking tasks and communicate with other participants.

3. **NXP Cup EMEA's Intelligent Car Racing** [9].
   This is an autonomous car competition that also makes use of a small model car provided to students. The racing procedure mainly deals with speed, precision and reliability and the vehicle has to be capable of overcoming an obstacle and lowering the speed in the presence of this object.

4. **Autonomous Driving Challenge - CARNET Barcelona** [10].
   Students have to develop a software solution for a scaled vehicle in 8 months. The challenge includes track localisation and navigation, perception of traffic signs, crosswalks and traffic lights and car manoeuvres, like overtaking, parking and overcoming obstacles.

5. **Bosch Future Mobility Challenge - BFMC** [11].
   This is a technical competition, where students have to develop their autonomous driving and connectivity algorithms on 1:10 model cars provided by Bosch itself. The vehicles have to run in a miniature smart city, where they have to deal with the presence of obstacles, pedestrians, traffic signs and other cars running on the same circuit.
   As already told previously, BFMC is the starting point for this thesis project.

The other category of competitions, instead, is for real cars that, from a mechanical point of view, have bigger dimensions (similar to go-karts or still bigger) and it is sometimes a task of the students to realize and build the entire vehicle.
Among this type of races, one can find:

1. **Self Driving Challenge** [12].
   Students have 6 months to develop their solution that will face six tasks: start on green traffic light, adhere to vehicle speed limit, identify traffic light stop and go signals, pay attention to pedestrian crossings, perform some overtaking and parallel parking manoeuvres.
   In this competition, students are asked to develop a software solution for their vehicle to achieve the best result.

2. **AutoDrive Challenge II** by SAE [13].
   It aims to letting team students develop and demonstrate an autonomous driving passenger vehicle, belonging to LEVEL 4, that can drive in urban scenarios according to what is written in SAE Standard rules. This competition lasts 4 years and it is in partnership with General Motors.

3. **Indy Autonomous Challenge** [14].
   This is the biggest and most important race competition linked to autonomous cars, whose body and design recall in some way to a structure similar to an old Formula 1 monocoque.
   The race cars that belong to this group are very fast and are equipped with a complex structure from every point of view.
   Teams competing in this challenge have to design a car that must be compliant with specific rules in terms of actions, performances and safety.

4. **Formula Student Driverless** [15].
   Formula Student Driverless asks university students to build, develop and run a full autonomous vehicle.
   Teams compete in three static events (Design, Business Plan Presentation and Cost), where they have to explain to the officials the choices made for their design in terms of used materials and equipment, implemented circuits and software solutions, as well as the followed building process and a detailed cost analysis related to every part of the car that is built and/or bought.
   Finally, there are dynamic events (Acceleration, Skid Pad, Autocross and Trackdrive) in which the car must be able to run autonomously, without going off the delimited road or hitting the cones.

Chapter 2

Set-Up

Bosch Future Mobility Challenge (BFMC) provides students with the needed equipment and basic software to be run on the electronic boards they give. However, for this thesis we decided to keep only part of the hardware as a starting point, changing a few components to achieve better performance, while for the software some modifications are made on the one running on the Nucleo board, compared to what is already implemented by the challenge itself. The pieces of code, instead, inherent to the Raspberry are developed by us full custom according to our application in C++.

2.1 Improvements from the initial model

To improve performance and reliability, we decided to make some upgrades concerning the major elements of this car: the BFMC proposes a Raspberry Pi 4 and its Raspi Cam 2 for the frame acquisition and image processing. Although, we adopted the latest version of the Raspberry board and a USB camera, which is compatible and easier to manage and to adapt to our system. The just mentioned devices will be described in section 2.2.
These changes were necessary for two reasons: on one side because Raspberry operating system, i.e. raspbian, has changed with the latest updates and has lost compatibility with its camera (connected by means of a flat cable). On the other side, image processing requires a huge amount of computational power and, for this reason, it needs to take place at a very high frequency to allow the car to see at every moment what is present in front of it and to go as fast as possible, according to the received information, without affecting the

control. These goals are achieved by adopting Raspberry Pi 5 [16], thanks to its increased computational speed, that is up to three times faster than the previous version.
Finally, from a mechanically point of view, it has also been necessary to fix the camera better to the chassis of the car, as the support was a bit peeled, providing in that way incorrect camera positioning and inclinations.

2.2 Connection diagram and Hardware description

Figure 2.1 represents the connection diagram of all the components in the car, while Figure 2.2 shows the inner part of our vehicle.

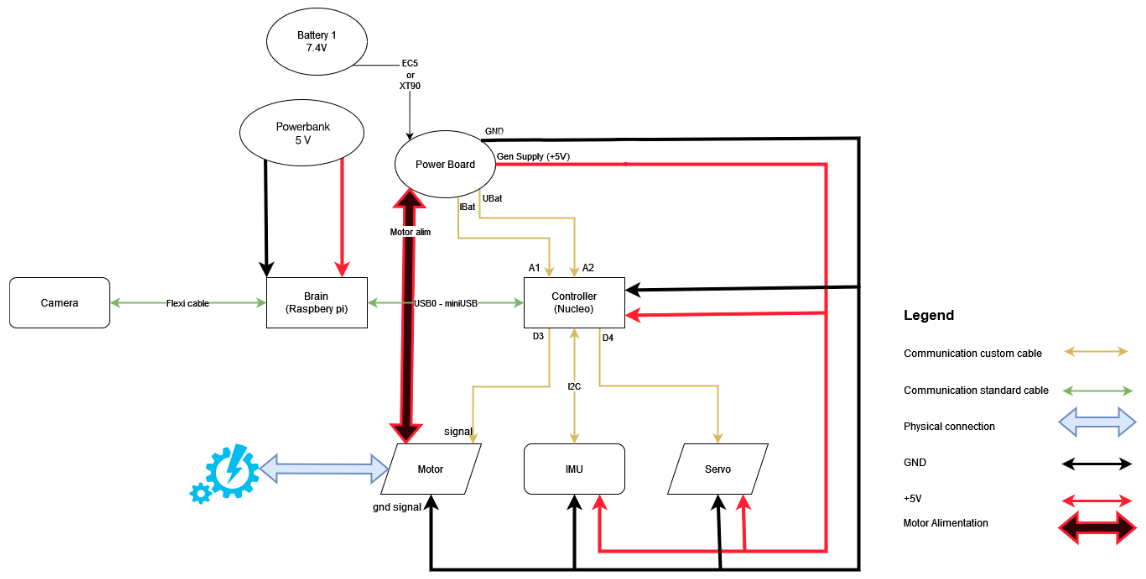


Figure 2.1: Connection diagram of all the HW components in the car.

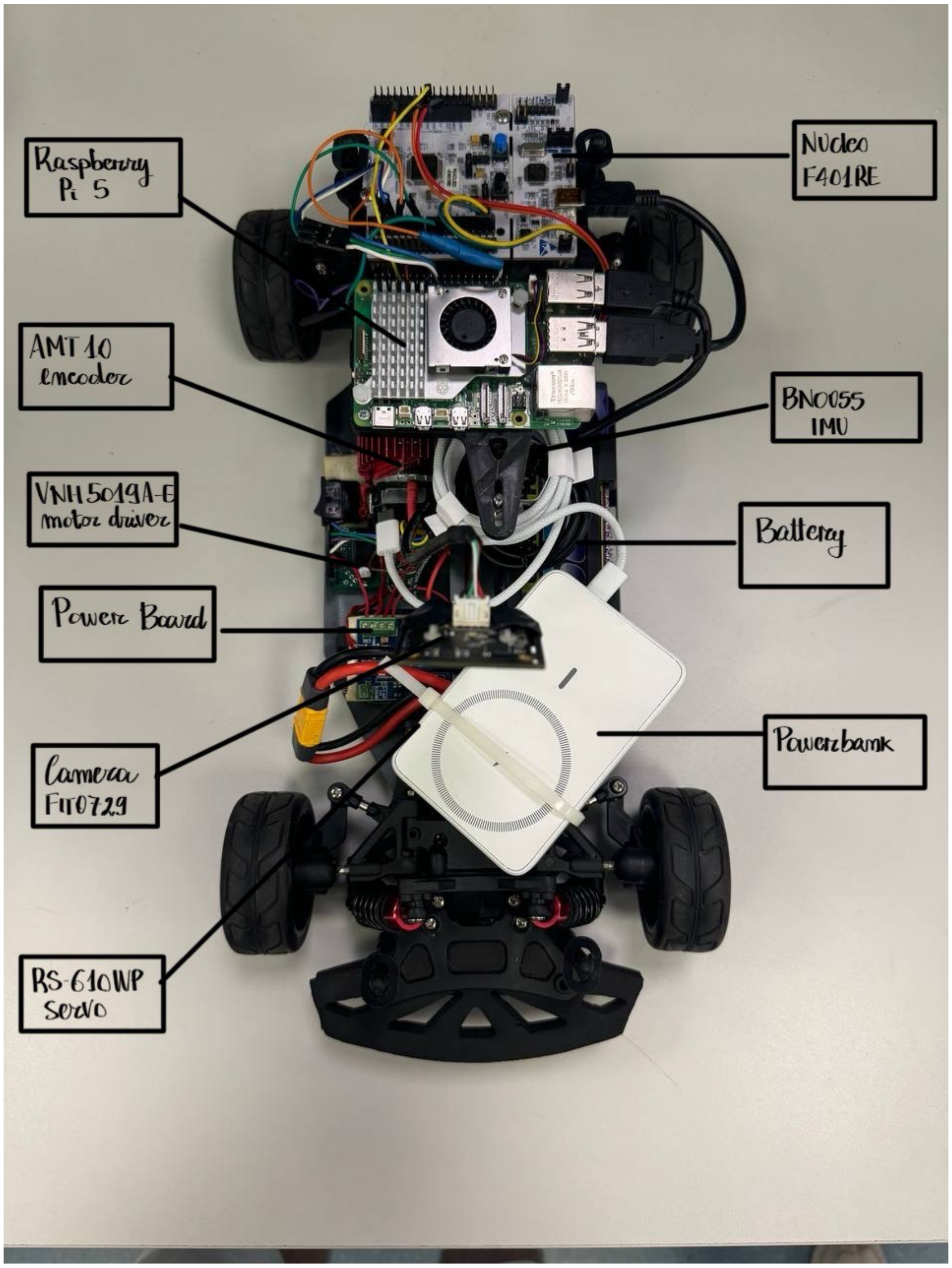


Figure 2.2: Top view of the car.

1. **Chassis:** base structure of the car, comes already mounted.
2. **Servo:** reely standard servo RS-610WP. It is the actuator for the steering mechanism.
3. **Motor equipment:** AMT10 modular incremental encoder and VNH5019A-E automotive fully integrated H-bridge motor driver. The motor instead is 531009 produced by Reely.
4. **Brain:** Raspberry Pi 5: faster compared to previous generations thanks to its processor.
5. **Controller:** Nucleo F401RE used to control the motors and to read data from the sensors and power board.
6. **Camera:** DFRobot FIT0729, distortion-less with 8 Megapixel resolution and 75-degrees camera parameter.
7. **Battery:** LiPo 55C 2s 7.4 V and capacity 6500 mAh.
8. **Inertial Measurement Unit (IMU):** sensor Bosch BNO055 shuttle board 3.0 flyer. It measures the acceleration, the angular rate and the orientation of the car, as well as other gravitational forces.
9. **Power Board:** Bosch custom board that delivers power to the entire vehicle and it returns feedbacks regarding the voltage and current consumption.
10. **Powerbank:** an additional power source is needed to feed the brain board independently from the battery, as it would suffer otherwise from voltage drops due to motor peaks or battery discharge.

The main electronic board used to process the information coming from the camera is Raspberry Pi, that communicates with the controller by means of a mini USB cable at 19200 *bps*, establishing a UART communication.
To decide which programming language was the most suitable for our application, a comparison was made between the different styles and a summary is reported in tables 2.1, 2.2, 2.3: as a result, according to our goals and needs, for Raspberry Pi, the C++ programming language resulted to be the most suitable compared to the others [17], [18], [19], thanks to its velocity in compiling as well as to its simplicity in working with classes and objects,

while for the controller we kept the already implemented C++ code with some modifications made by us.
Moreover, Object Oriented Programming (OOP) [20] focuses on the objects we want to manipulate and this approach is commonly used for complex and large software that needs to be continuously updated.

Table 2.1: Comparison between C language and Python.

C	**Python**
Variable declaration	NO variable declaration
NO OOP (Object Oriented Programming)	Has OOP
Use of pointers	NO pointers
C is a compiled language	Python is just an interpreted language
Has a limited number of built-in functions	Has a large library of built-in functions
Is compiled directly to machine code	Is first compiled to byte-code and then interpreted
Has no complex data structures	Has some complex data structures
Faster	Slower

Table 2.2: Comparison between C++ language and Python.

C++	**Python**
Is pre-compiled	Is interpreted
Is faster once it is compiled	Is slower since it uses an interpreter
Powerful OOP features	OOP features
Is statically typed	Is dynamically typed
Function type and return values must be type-coherent	No restrictions on the type of the returned value
Explicit memory management	Automatic memory management
Variables defined in loops are available locally	Variables are accessible also outside the loop

Table 2.3: Comparison between C language and C++.

C	**C++**
No polymorphisms, encapsulation, inheritance, OOP	Supports polymorphism, encapsulation, inheritance
Is a subset of C++	Is a superset of C
Supports procedural programming	Supports both procedural and OOP paradigms
No operator overloading	Supports operator overloading
Is a function-driven language	Is an object-driven language
No reference variables	Supports reference variables
Does not have namespace features	Supports namespace features to avoid name collisions
Focuses on method or process	Focuses on data
Cannot throw exceptions	Can throw exceptions
Complex memory allocation	Easier memory allocation and de-allocation

2.2.1 Boards usage

1. **Raspberry Pi 5:** it constitutes a bridge between what the car sees and which action it has to perform.
   The board, in fact, receives the information from the camera and processes the frames with the use of *OpenCV libraries* [21], that are essential to perform the Lane Detection part.
   After having identified which road features are important to isolate lane lines, additional controls are implemented to tell the car what to do according to the scenario it is involved in.

2. **NUCLEO F401RE:** the controller receives the commands from Raspberry Pi (e.g. set a particular speed or a turn angle) and drives either the motor or the servo to perform the activity that the main board tells according to what it sees.
   Furthermore, there is a dual exchange of information between the two boards: while the brain sends the command to set, the controller reacts not only on the car, but also by sending back an acknowledgment to Raspberry to let it know whether the message is received, and consequently if the action is performed successfully, or not.

Figure 2.3 shows how the controller and the brain exchange information with each other and with the remaining parties.
To guarantee that both the frame acquisition and the sending of commands are performed simultaneously and correctly, the thread class has been adopted: its objects allow, in fact, the execution of two or more parts of a program in parallel for maximum usage of the CPU.

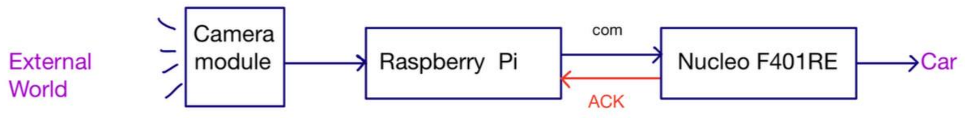


Figure 2.3: Communication between the boards.

2.2.2 Nucleo F401RE pinout

Pinout Nucleo - Encoder

The rotation of the encoder is of rotational type and this is identified by the number 3 in the Part Number (PN): AMT103 [22]. Moreover, the information from this device is sent to the Nucleo board by means of two channels.
Table 2.4 shows the pinout between the Nucleo board and the encoder.

Table 2.4: Pinout Nucleo - Encoder.

Pin Nucleo STM32F401	**Modular Incremental Encoder AMT103**	**Wire color**
PB7	Channel B	yellow
PB6	Channel A	green-brown

Pinout Nucleo - Motor driver

This is a full bridge motor driver [23], typically used in automotive applications. Its output current is limited at 30 A and the PWM can operate up to 20 kHz.
Table 2.5 shows the pinout between the Nucleo board and the motor driver.

Table 2.5: Pinout Nucleo - Motor driver.

Pin Nucleo STM32F401	**VNH5019A-E**	**Wire color**
D3	PWM	blue-green
D2	INA	white-blue
D7	INB	green-white
A2	CS	orange-violet

Brief pin descriptions:

1. INA - INB: they are input signals that allow to select the motor direction and the brake condition. In particular, for our application, INA is a clockwise input, while INB is a counter-clockwise input.

2. CS: allows to monitor the motor current by delivering a current proportional to its value when CS DIS pin is driven low or left open. The information can be read as an analog voltage across an external resistor.

3. CS DIS: this is an active high CMOS compatible pin to disable the current sense pin.

4. PWM: it lets control the speed of the motor in all conditions.

Pinout Nucleo - IMU

This is a shuttle board designed by Bosch and it communicates through I2C Connection.
Table 2.6 shows the pinout between the Nucleo board and the IMU board.

Table 2.6: Pinout Nucleo - IMU.

Pin Nucleo STM32F401	**BNO055**	**Wire color**
D15/SCL	SCL	yellow
D14/SDA	SDA	green
5V Power	5V	red
GND Power	GND	orange

Pinout Nucleo - Servo

This is the servo motor [24] used for steering actions. Only one analog signal is sent to the controller board.
Table 2.7 shows the pinout between the Nucleo board and the servo motor.

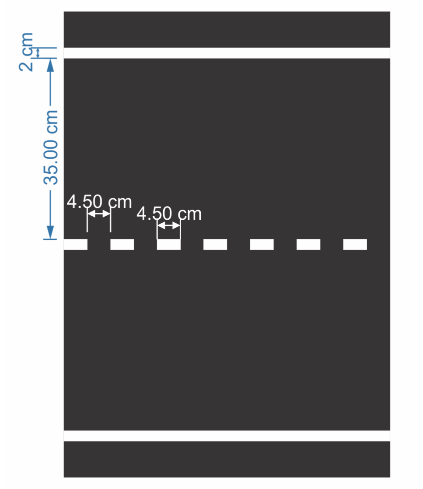
Table 2.7: Pinout Nucleo - Servo.

Pin Nucleo STM32F401	**RS-610WP MG**	**Wire color**
D4	signal	white-orange

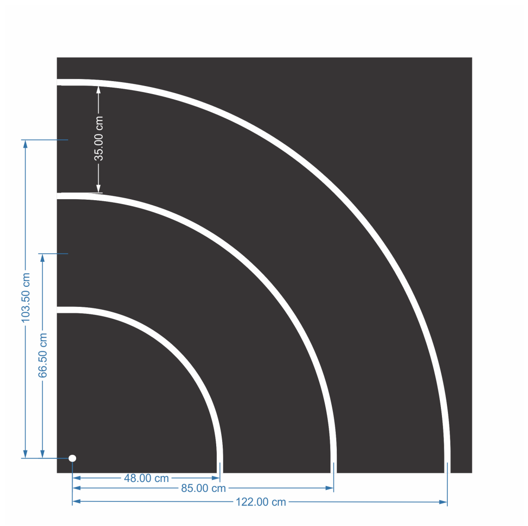
2.3 Race-track

The car has to run on a track whose lane markings may be dashed or continuous and whose line is 2 *cm* wide. Lane width instead measures 35 *cm* and dashed markings are 4.5 *cm* long as the distance between them.
Tight curves and intersections are also present, and the vehicle must be able to drive properly within its lane, maintain its trajectory or stop in the case many roads meet at a certain point.
Stop line is 42 *cm* wide and 4 *cm* long.
An example of the possible road configurations is shown in Figure 2.4.

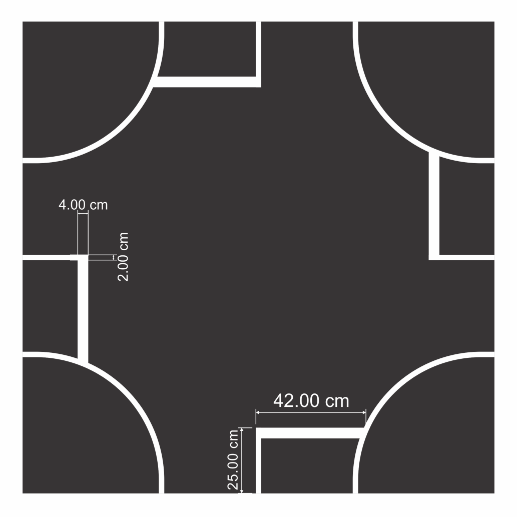
Once the dimensions of the track are known, the camera has to be set-up in a way that it is able to capture continuously the area in front of it, including the two lane lines, and without any loss of information, as it will be crucial for the development of lane detection algorithms.



(a)Straight road.



(b)Curve.



(c)Intersection.

Figure 2.4: Different road configurations.

2.4 Software constraints

The car we adopted to perform our studies is a model and, because of this, it has some physical limitations from a mechanical point of view. They concern mainly the maximum angle at which the vehicle can turn left or right, as well as the maximum speed that can be reached. To avoid to set these parameters to the wrong values, software constraints are imposed by means of the Mbed Studio platform and the used programming language, also in this case, is C++.

2.4.1 Constraint on the steering angle

The interval in which the steering angle can assume positive and negative values is [-25, 25] degrees, as shown in listing 2.1. Outside that region, the car is not able to perform any steering action.
The instruction that applies this condition is:

```
    drivers::CSteeringMotor g_steeringDriver(D4, -25.0,
      25.0);
```

Listing 2.1: Constraint on the steering angle.

where:

1. *drivers:* is a namespace that includes the definition of different classes. It initializes the PWM parameters and it sets the speed reference to zero.

2. *CSteeringMotor:* is a class used to control the servo motor connected to the steering wheels.

3. *g_steeringDriver:* is a variable that refers to

   (a) pin D4: enabled for setting the angle of the servo motor on the Nucleo board.

   (b) -25.0: is the highest angle for letting the car turn left.

   (c) 25.0: is the highest angle for letting the car turn right.

2.4.2 Constraint on the speed

The speed values are also constrained to avoid to reach large values in magnitude that may provoke some dangerous situations and consequences.
The allowed interval in this case is a bit bigger, as the traction motor is not subjected to the same limitations as the servo, and this region is [-50, 50] cm/s, as shown in the listing 2.2 that corresponds to 1.8 km/h.
The instruction to write is:

```
    drivers::CSpeedingMotor g_speedingDriver(D3, D2, D7,
        -50.0, 50.0); //speed in cm/s
```

Listing 2.2: Constraint on the speed.

where *drivers* and *CSpeedingMotor* are defined as in the previous subsection, but this time the second refers to a class used to control the brush-less motor connected to the driving shaft.
g_speedingDriver is a variable that refers to pins enabled on the Nucleo board:

1. pin D3: enabled for setting the speed.
2. pin D2: is associated to the INA pin of the motor driver that makes the wheels rotate clockwise, hence the car moves forward.
3. pin D7: is associated to the INB pin of the motor driver that makes the wheels rotate counterclockwise, hence the car moves backward.
4. -50: is the highest speed for letting the car move backwards.
5. 50: is the highest speed for letting the car move forward.

The direction of the car for which it moves for- or backwards is imposed with the PWM: if the $PWM > 0$, then the vehicle has a positive speed, negative otherwise.

The function that implements what has just been mentioned is called *setSpeed* and it is defined in listing 2.3:

```
    void CSpeedingMotor::setSpeed(float f_speed)
     {
     if(f_speed < 0){
         m_ina = 0;
         m_inb = 1;
     }
     else {
         m_ina = 1;
         m_inb = 0;
     }
     m_pwm_pin = std::abs(f_speed);
};
```

Listing 2.3: Definition of forward and backwards directions.

2.5 Periodic messages

In Mbed Studio platform, some instructions that send back to Raspberry Pi periodic messages about the instant current consumption and the battery voltage are also present, as well as measured values from the Inertial Measurement Device. This allows the users to monitor in real-time those quantities to avoid, for example, to discharge completely the battery or to be in presence of too high currents.

2.5.1 Instant current consumption

The task to send back the information about the instant current consumption of the battery is given in listing 2.4,

```
periodics::CInstantConsumption g_instantconsumption(0.2 /
    g_baseTick, A2, g_rpi); \\ g_baseTick = 0.0001;
```

Listing 2.4: Message for the instant current consumption.

where

1. *periodics:* is a namespace and it is used for all periodic messages.
2. *CInstantConsumption:* is a class for the current.
3. *g_instantconsumption:* is a variable that refers to
   (a) $\frac{0.2}{g\_baseTick}$: is the frequency with which the message is sent.
   (b) A2: is the analog pin to enable to send the message from the Nucleo to Raspberry.
   (c) g_rpi: is a variable used in the serial interface with another device, Raspberry in our case.

2.5.2 Battery voltage

The instruction to send periodically the information about the battery voltage, to know when it is discharging, is shown in listing 2.5,

```
periodics::CTotalVoltage g_totalvoltage(3.0 / g_baseTick,
    A1, g_rpi); \\ g_baseTick = 0.0001;
```

Listing 2.5: Message for the battery voltage.

where

1. *CTotalVoltage:* is a class for the voltage.
2. *g_totalvoltage:* is a variable that refers to the same parameters as the current function, but this time the frequency of the message is increased, and the enabled pin for the UART communication from the Nucleo and Raspberry is A1.

2.5.3 IMU measurement

The last periodic message exchanged between the brain and the controller is related to the measurements made by the inertial device.
The task that performs this part is written in listing 2.6,

```
periodics::CImu g_imu(0.1 / g_baseTick, g_rpi, I2C_SDA,
   I2C_SCL); \\ g_baseTick = 0.0001;
```

Listing 2.6: Message for the inertia measurement.

where

1. *CImu:* is a class for the inertia.

2. *g_imu:* is the variable that refers to the frequency of the message and to the serial interface as the other periodic messages, plus

   (a) I2C_SDA: is the Serial Data pin.

   (b) I2C_SCL: is the Serial Clock pin.

   Both pins are used in the I2C protocol and they have to be configured in open-drain operation with the suitable bits. *Open-drain* refers to an output that can drive the bus low or floating, and in this last case the bus requires a pull-up resistor to pull it up to a certain voltage.

Chapter 3

Methodology

The main objective of this chapter is to describe the specific procedures and techniques adopted to develop our research in the autonomous driving field. In particular, we will start explaining the performed steps to finally give a complete overview of the implemented algorithms for the image vision part.

3.1 Image processing

Image processing [25], [26] is a technique applied to extract meaningful data from the frame acquired by an external camera, whose main function is to identify lane lines, traffic signs and obstacles to control the behavior of an autonomous car.
Vision-based lane detection approach can be of two types:

1. **Feature-based techniques:** deal with the detection of features describing lane markings (e.g. color, edges).
   These methods have good computational efficiency and work very well when lane markings are clear, but they can be easily affected by robustness losses.

2. **Model-based techniques:** they involve the training of the algorithm with a series of images aimed to recognize the various elements that the vehicle may encounter along its route: thus, unlike the previous model, it is no longer the size of the objects that needs to be recognized, but the shapes and features that make them up.

An on-board vision system is important for the generation of a reference path, as well as to obtain the vehicle states (position and inclination with respect to the lane) and to control the car in the desired path. In this work, it consists of a camera [27] that continuously acquires frames in real-time, and these last ones will be subjected to multistage pre-processing steps to detect clearly and accurately lane lines, as they will be essential later for the lane keeping part to help the car to follow the desired path. To this purpose, from a programming point of view, both objectives, i.e. lane detection and lane keeping, can be executed in parallel thanks to the utilisation of threads [28], that let both parts run continuously and independently at the same time: the acquisition and elaboration of the frame, and the sending of commands to the Nucleo board.

3.2 Multistage pre-processing steps for Lane Detection

3.2.1 Frame capture

The frame is captured continuously in real-time to know, at each moment, what the car sees in front of it and to take the right decision consequently.

```
void captureFrames(VideoCapture &cap) {
    Mat frame;
    while (!stopCapture)
    {
        if (cap.read(frame))
        {
            lock_guard<mutex> lock(queueMutex);
            frameQueue.push(frame.clone());
        }
    }
}
```

Listing 3.1: Thread for the image acquisition.

In listing 3.1, it is implemented the image acquisition thread: it acquires frames continuously as long as it collects data from the camera, as it is shown in picture 3.1.

Inside this void function, a variable of type `Mat` is defined, in which the acquired frame will be stored, while the mutex and queue of Mat functions `lock_guard` and `frameQueue`, respectively, ensure that the frame has its own memory allocation used by only one thread at a time.

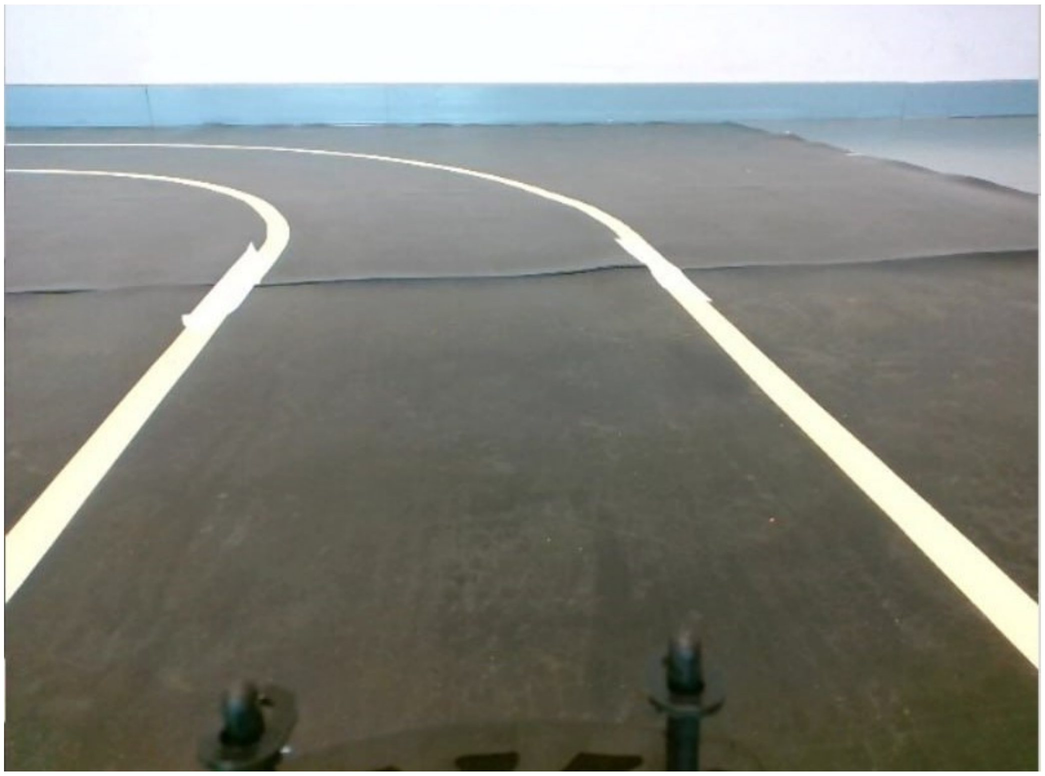


Figure 3.1: Acquired frame.

3.2.2 Gray scale conversion

The captured frame is in RGB format, but it needs to be converted to gray scale as lane marking colors can be distinguished better compared to the other shades. Moreover, a gray image is uni-dimensional (while RGB is tri-dimensional) and this reduces the size of the processed image.

```
Mat imageGray;
cvtColor(frame, imageGray, COLOR_BGR2GRAY);
```

Listing 3.2: Gray scale conversion.

In listing 3.2, it is created a `Mat` object called "*imageGray*" in which we store the initial frame converted to gray for a better edge recognition of the image objects and for the unfolding of the sliding windows algorithm. The result of this transformation is depicted in Figure 3.2.

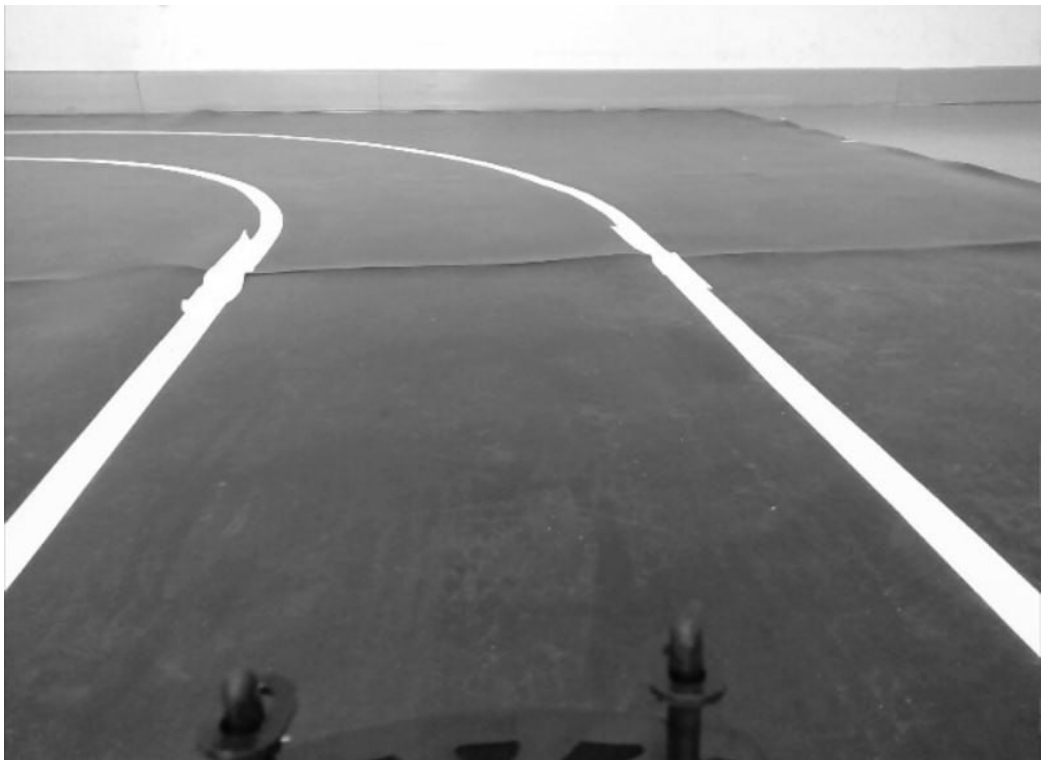


Figure 3.2: Grayscale conversion.

3.2.3 White and Yellow pixels extraction

The road markings can be either yellow or white. In our simulation environment, however, we only have white lines, and, for this reason, we decided to perform a gray-scale transformation and to create then a mask collecting those pixels. Listing 3.3 contains the piece of code implementing this step.

If we also wanted to create an algorithm working on both white and yellow lines, we would need to create two separate masks, one for the white and one for the yellow, and mix them together with the `bitwise_or` and the `bitwise_and` functions.

```
Mat maskWhite;

inRange(imageGray, Scalar(200, 200, 200), Scalar(255,
    255, 255), maskWhite);
```

Listing 3.3: White pixels extraction.

3.2.4 Gaussian blur

This is a low-pass filter adopted to remove any noise from the image, and it is implemented by calling the function *GaussianBlur* that applies the transformation to the processed image, as reported in listing 3.4.
However, this algorithm reduces the resolution, having consequently not perfectly sharp edges, in particular when the kernel size of the function is increased more than needed. For our application, it is set equal to 5 (its value must be odd and equal or greater than 3).

```cpp
int kernelSize = 5;

GaussianBlur(processed, processed, Size(kernelSize,
    kernelSize), 0);
```

Listing 3.4: Gaussian blur.

3.2.5 Gap fillings and gray-scale saturation

Gap-filling operations are performed to close all small holes within the image: at this stage, it is created a 3x3 matrix full of ones used for the `dilation` part, which expands the bright areas of the image.

```cpp
Mat kernel = Mat::ones(3, 3, CV_8U);
dilate(processed, processed, kernel);
```

Listing 3.5: Image dilatation.

The next step consists in the application of the `erosion`, which closes this time the small holes within the image.

```cpp
erode(processed, processed, kernel);
morphologyEx(processed, processed, MORPH_CLOSE, kernel);
```

Listing 3.6: Erosion of the small noise.

Finally, all the pixels are saturated: with the threshold function, we separate out parts of the image that correspond to objects we want to analyze. To make this distinction among interesting and uninteresting pixels, it is needed to compare their intensity with respect to a threshold, set for us to 180: if the pixel intensity is higher than the threshold, they are saturated to the maximum value, that is 255 for the white, otherwise they are set to the minimum, 0 for the black, as shown in equation 3.1:

$$dst(x,y) = \begin{cases} maxVal, & src(x,y) > thresholdVal \\ 0, & otherwise \end{cases} \tag{3.1}$$

```
const int thresholdVal = 180;
threshold(processed, processed, thresholdVal, 255,
    THRESH_BINARY);
```

Listing 3.7: Pixel saturation.

An example of how this function works is shown in Figure 3.3, while listings 3.5, 3.6 and 3.7 show how to implement the above mentioned functions for the gap fillings and gray saturation.

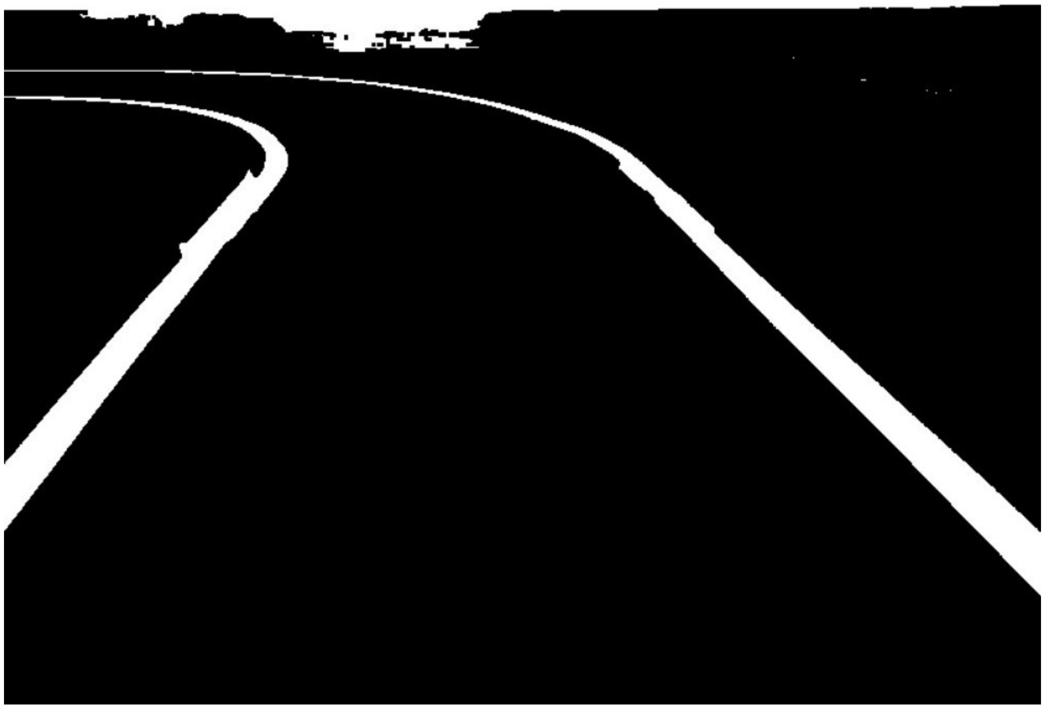


Figure 3.3: Gap fillings and pixel saturation.

3.2.6 Bird's eye view

After having acquired and processed the image, it has to be transformed from the frontal view into the bird's eye view by performing an Inverse Perspective Mapping (IPM).
The bird's eye view consists in taking a rectangular image and *spreading* it either on the top or on the bottom edges, so as to eliminate the perspective. As a result, the lane lines become parallel to the vertical direction of the frame and the vehicle positioning more precise: by performing this transformation, in fact, it is possible to search for white peaks more accurately.
The transformation in the bird's eye view is carried out in three steps:

1. Definition of the points of the original image.
2. Definition of the target points.
3. Processing of the image using the `warpPerspective` function.

The points of the original frame are selected as written in listing 3.8:

- 0: upper left corner (at 40% of the frame height from the top)
- 1: upper right corner (at 40% of the frame height from the top)
- 2: lower right corner
- 3: lower left corner

Thanks to the `.cols` and `.rows` functions, it is possible to take the extreme points, regardless of the image size.
Furthermore, they are stored in a variable of type `Point2f`, that is used for 2-dimensional points whose coordinates are float.

```
    Point2f OriginalImage_Points[4] = {
    Point2f(0, processed.rows * 0.4),
    Point2f(processed.cols, processed.rows * 0.4),
    Point2f(processed.cols, processed.rows),
    Point2f(0, processed.rows)
    };
```

Listing 3.8: Initial points of Bird's eye view.

In the same way, we select the points of the final frame in listing 3.9, that is the result of the bird's eye view transformation:

- 0: upper left corner
- 1: upper right corner
- 2: lower right corner
- 3: lower left corner

Here we take the upper points nearest to the center to obtain a trapezoidal perspective. In particular, *point 2* is taken after the 32% of the length of the image, starting from the left edge, while *point 3* after the 68%. These parameters must be changed if the angular position of the camera changes.

```
Point2f FinalImage_Points[4] = {
  Point2f(0, 0),
  Point2f(processed.cols, 0),
  Point2f(processed.cols * 0.68, processed.rows),
  Point2f(processed.cols * 0.32, processed.rows)
  };
```

Listing 3.9: Final points of Bird's eye view.

Finally, it is computed the perspective matrix, passing from the points of the original image to those of the final one by means of the `getPerspectiveTransform` function.
The result is stored in the variable `Mat birdsEyeView`, after the application of the `warpPerspective` function, that performs the perspective transformation to the image (for us, it is the *processed* Mat object), as shown in listing 3.10.

```
Mat perspectiveMatrix = getPerspectiveTransform(
    OriginalImage_Points, FinalImage_Points);
Mat birdsEyeView;
warpPerspective(processed, birdsEyeView,
    perspectiveMatrix, frame.size());
```

Listing 3.10: Bird's eye view transformation.

The output of this modification is represented in picture 3.4.

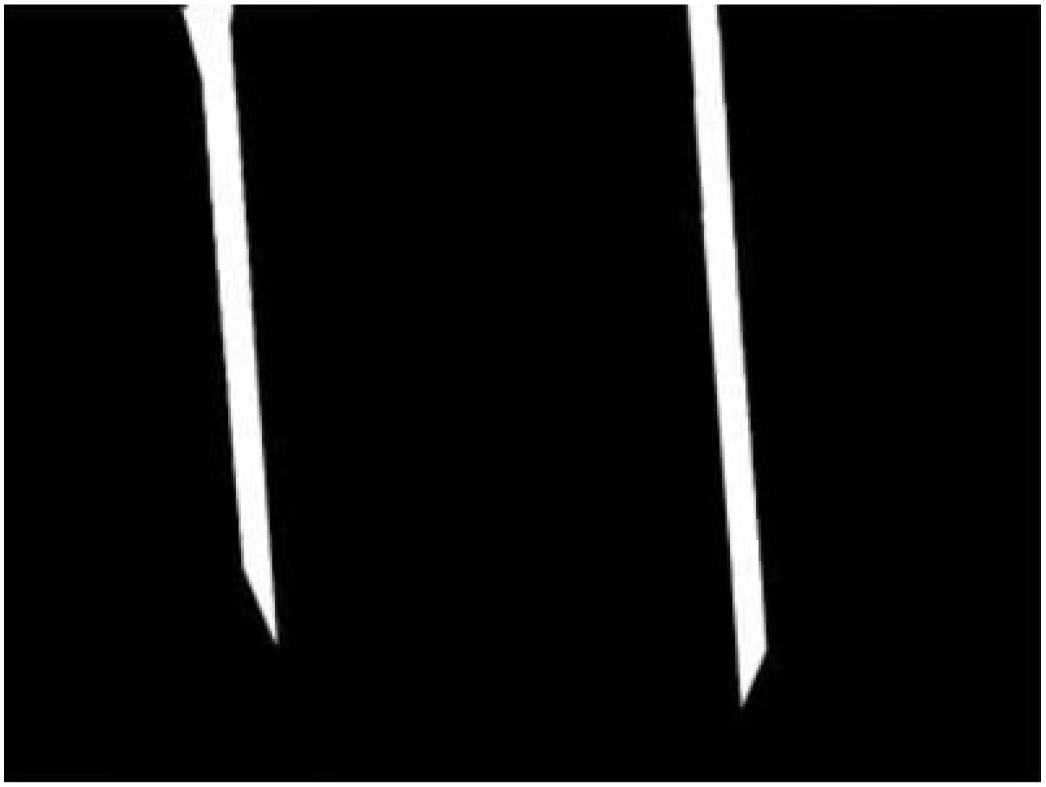


Figure 3.4: Bird's eye view.

3.2.7 Hough transform

The Hough transform [29] is a mathematical function used to identify lane lines on an image. More precisely, it transforms lines in the image space into points in the parameter space (or vice-versa): in polar coordinates, the line equation

$$r = xcos(a) + ysin(a)$$

is transformed to a point having coordinates (r, a), with r being the distance from the origin to the closest point on the line and a the angle between the x-axis and r.

In our algorithm, `lines` are stored in a vector of vectors of 4 integer elements as in listing 3.11, because they have to contain two couple of points: the initial points having coordinates (x_1, y_1) and the final points (x_2, y_2), both belonging to the lines.

```
vector<Vec4i> lines;
```

Listing 3.11: HoughLines vector.

The Hough transform is a very computationally onerous operation. To alleviate the power required for this operation, the dimensions of the image to which this transformation is applied have been reduced: height and width have been halved as written in listing 3.12.

```
Mat resizedImage;
resize(birdsEyeView, resizedImage, Size(birdsEyeView.cols
    /2, birdsEyeView.rows/2));
```

Listing 3.12: Resizing of the input frame.

The parameters to define for the Hough transform are:

- **rho:** it is the pixel resolution of the Hough transform accumulator that refers to the distance.
- **threshold:** it corresponds to the minimum value to accept a line. Only lines having at least a number of points equal or greater than the threshold are valid.
- **minLength:** it is the minimum length to consider the object as a line. Below that value, we ignore it.
- **maxLineGap:** it is the maximum distance between two segments to consider them as a part of a single line. If the value is higher than this quantity, lines are considered as two distinct.

```
int rho = 1;
int threshold = 50;
int minLineLength = 80;
int maxLineGap = 20;

HoughLinesP(resizedImage, lines, rho, PI/60, threshold,
    minLineLength, maxLineGap);
```

Listing 3.13: Parameters of the Hough transform.

The function we use to implement the Hough transform is `HoughLinesP`, that finds line segments in the binary image *resizedImage* with a probabilistic approach and whose parameters are:

- **resizedImage:** it is the image in which we detect the lines.
- **lines:** it is where the identified lines are stored.
- **rho:** it refers to the distance, as previously mentioned.
- **PI/60:** it is the resolution in radians for the angle.
- **threshold**, **minLineLength** and **maxLineGap**, as already described.

The above mentioned variables and function are shown in listing 3.13.
We then create another variable, identical to *birdsEyeView*, where we draw the lines found by the function. To do this, however, it is needed to double the coordinates of all the lines, as they come from a scaled image, to represent them on the final one and this is represented in listing 3.14.

```
Mat HoughInBirds = birdsEyeView.clone();
vector<Vec4i> scaledLines;
for (const auto& l : lines) {
    Vec4i scaledLine;
    scaledLine[0] = l[0] * 2;
    scaledLine[1] = l[1] * 2;
    scaledLine[2] = l[2] * 2;
    scaledLine[3] = l[3] * 2;
    scaledLines.push_back(scaledLine);
}
```

Listing 3.14: Lines scaling.

Finally, the lines are plotted in the clone image `HoughInBirds` depicted in Figure 3.5.

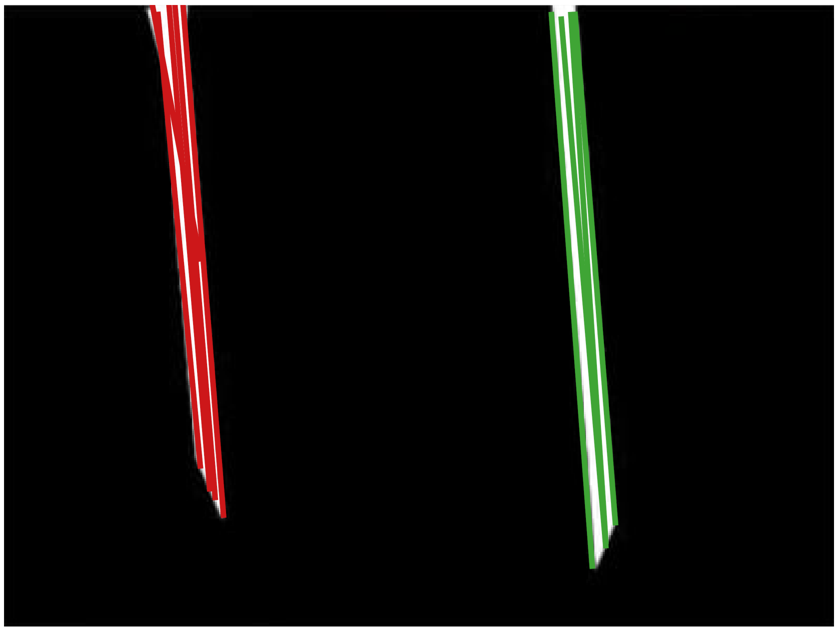


Figure 3.5: Hough lines.

3.2.8 Lane recognition

This part is useful to distinguish points belonging to the right lane or to the left one. What we do, is a loop over all the lines detected with the Hough transform algorithm to compute the slope of each and, according to the value, classify the points as part of:

1. the left, if the presence of the line is in a window at the bottom left of the image. Points are stored in the variable `vector<Point2f> leftLineControlPoints`.

2. the right, if the presence of the line is in a window at the bottom right of the image. In this case points are collected in `vector<Point2f> rightLineControlPoints`.

3. the stop line (horizontal), if the slope is under a threshold set to 0.15. Points are accumulated in `vector<Point2f> crossWalkControlPoints`.

```
vector<Point2f> leftLineControlPoints,
    rightLineControlPoints, crossWalkControlPoints;
float slope = 0;
float slope_threshold = 0.15;
```

Listing 3.15: Variable and threshold definition for the lane recognition algorithm.

After the declaration of the necessary variables done in listing 3.15, it is extracted the *i-th* line in `scaledLine` and it is saved in the new vector `line`, from which we draw out the extremities of the lane that are stored in `Point2f p1, p2`, and whose points are used to compute the slope of each line, as shown in listing 3.16.
If the slope goes to infinity, we saturate it at 999 to avoid the division by 0.

```
for (size_t i = 0; i < scaledLines.size() ; ++i)
{
    Vec4i line = scaledLines[i];

    Point2f p1(line[0], line[1]);
    Point2f p2(line[2], line[3]);

    if(p2.x - p1.x == 0)
    {
        slope = 999;
    }
    else
    {
        slope = (p2.y - p1.y)/(p2.x - p1.x);    //(y2-y1)
           /(x2-x1)
    }
```

Listing 3.16: Slope computation.

As already mentioned, if the slope is greater than a certain threshold, it is considered as a lane line, otherwise it is classified as horizontal.
The piece of code shown in 3.17 is useful for the implementation of the sliding window algorithm, because it looks for both left and right lines to be later represented in the two windows at the bottom corners of the image.
The dimensions of windows are taken 40% wide and 30% high of the total frame.
If the lines are identified inside that area, the `left_lane` and-or the `right_lane` become *true*, depending on which line is present.

```
        if ((abs(slope)>slope_threshold))
        {
            if((p1.y> birdsEyeView.rows*0.7)||(p2.y >
                birdsEyeView.rows*0.7))
            {
                if((p1.x< birdsEyeView.cols*0.4)||(p2.x <
                    birdsEyeView.cols*0.4))
                {
                    left_lane = true;
                }

            }

            if((p1.y > birdsEyeView.rows*0.7)||(p2.y >
                birdsEyeView.rows*0.7))
            {
                if((p1.x > birdsEyeView.cols*0.6)||(p2.x >
                    birdsEyeView.cols*0.6))
                {
                    right_lane = true;
                }

            }

        }
```

Listing 3.17: Check for the presence of the left and right lines.

On the other hand, if the slope of the lines is less than 0.15, it means we are in the presence of a stop sign and it is necessary to check that the lines have a certain length, in this case 60 pixels, and that they are positioned in a window placed in the middle of the frame, as it often happens that there is noise on the edges that leads to the presence of pedestrian crossings or stop signs being identified even when they are not there.
Starting from the top, the window is set in height from the 20% to the 50% of the image.
The part of algorithm related to the presence of horizontal lines is written in listing 3.18.

```
        else if (sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2))>60)
            {
            crossWalkControlPoints.push_back(p1);
            crossWalkControlPoints.push_back(p2);

            if((p1.y> birdsEyeView.rows*0.2 && p1.y<
                birdsEyeView.rows*0.5)||(p2.y > birdsEyeView.
                rows*0.2 && p2.y<birdsEyeView.rows*0.5)) {
            crosswalk_stop = true;
            }
            else {
            crosswalk_stop = false;
            }
        }
    }
```

Listing 3.18: Presence of the crosswalk or stop lines.

Finally, we look for the presence of both lines if flags `right_lane` and `left_lane` are true at the same time: this will be important for the future controls on the car.
Listing 3.19 represents this last check.

```
if((right_lane == true) && (left_lane == true))
{
    both_lines = true;
}
else
    both_lines=false;
```

Listing 3.19: Check for the presence of both lines.

Figures 3.6 and 3.7 show the result of listings 3.17, 3.18 and 3.19.

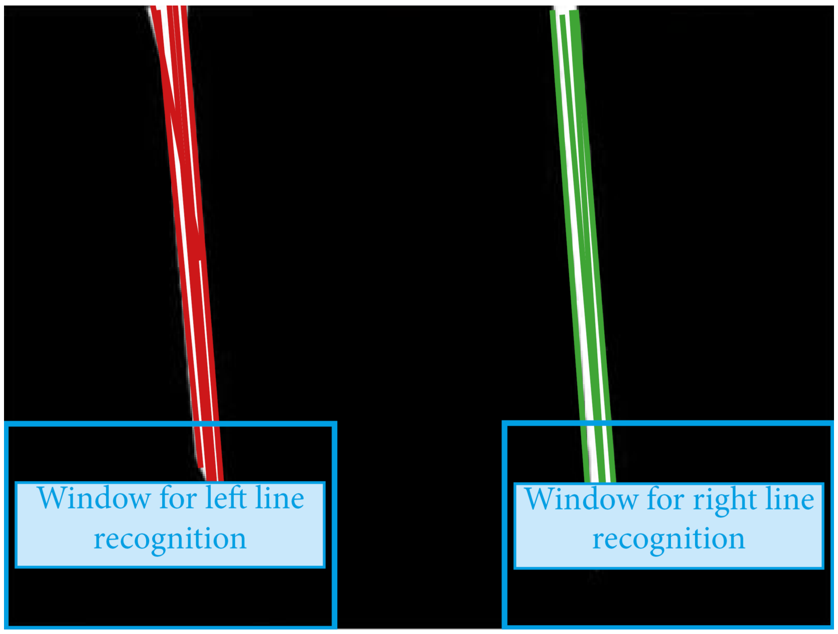

Window for left line recognition
Window for right line recognition

Figure 3.6: Windows for line recognition.

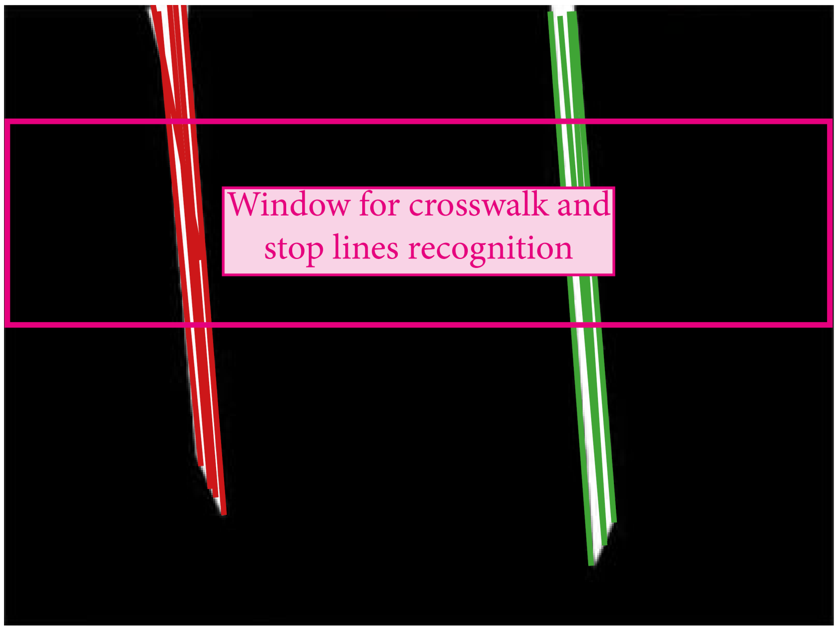


Figure 3.7: Window for crosswalk and stop lines.

3.2.9 Sliding Windows

The sliding window algorithm is a technique used to identify the lines of a lane. It consists of placing two windows at the beginning of the two lines at the bottom of the frame, using the methodology of searching for the peaks in the histogram of the density of white pixels, and, from there, delimit immediately above each of the two windows another window in which the white peaks will be searched again to place again the new window. In this way it is possible to follow the course of the lane lines with a good accuracy.
The **histogram peak detection** is a fundamental step for the development of this algorithm, as we need to get first the histogram of the image with respect to the x-axis to know how many white pixels are present in each column of the image. We then look for the highest peaks in both sides that will correspond to the center points of the lines: in this way, in fact, it is possible to locate the horizontal positions of the left and right lane markings x_L and x_R.
To implement these actions, all the pixels from half of the image to the bottom are analyzed and we also choose a narrow region of interest to avoid areas where there may be white pixel noise.
Finally, the histogram is calculated using the `reduce` function, which returns the sum of white pixels in the applied range, that is the column of interest where there are the peaks.
This initial part of code is reported in listing 3.20.

```cpp
int start_row = static_cast<int>(binary_warped.rows *
    0.5);

Mat roi = binary_warped(Range(start_row, binary_warped.
    rows), Range::all());

Mat histogram_mat;
reduce(roi, histogram_mat, 0, cv::REDUCE_SUM, CV_32S);

vector<int> histogram(histogram_mat.begin<int>(),
    histogram_mat.end<int>());
```

Listing 3.20: Histogram for white pixels.

The next step to perform is the seek for the peaks of the left and right halves of the histogram that will correspond to the starting point for the respective lines.

In particular, in listing 3.21, the left peak is searched in the initial 40% of the image width, while the right peak in the final 40%: this is to create a window in the middle, equal to the 20% of the frame width, where the peaks cannot be found, in a way to avoid an overlap of them when the car is close to the line.

```
int midpoint = binary_warped.cols*0.5;
int leftx_base = distance(histogram.begin(), max_element(
    histogram.begin(), histogram.begin() + midpoint*4/5));
int rightx_base = distance(histogram.begin() + midpoint,
    max_element(histogram.begin() + midpoint*6/5,
    histogram.end())) + midpoint;
```

Listing 3.21: Histogram peak detection.

After having found the starting point for the first left and right windows, it is needed to define the number of windows and their height to continue with the implementation of the algorithm, as in listing 3.22.

```
int nwindows = 8;
int window_height = binary_warped.rows/nwindows*4/3;
```

Listing 3.22: Choice of number of windows and their height.

As next step reported in listing 3.23, the x and y positions of all the non-zero pixels in the image are identified and saved in two vectors of coordinates to easily find the peaks in the subsequent windows.

```
vector<Point> nonzero_points;
findNonZero(binary_warped, nonzero_points);
vector<int> nonzeroy, nonzerox;

for (const auto& point : nonzero_points) {
    nonzeroy.push_back(point.y);
    nonzerox.push_back(point.x);
}
```

Listing 3.23: Storage of nonzero pixels.

The current positions of each window in listing 3.24 have to be updated and they make reference to the quantities defined previously in the search for the peaks.

```
int leftx_current=leftx_base;
int rightx_current=rightx_base;
```

Listing 3.24: Current position update.

As final variable definitions in 3.25, it is needed to decide a certain margin and the minimum number of pixels on the peak to recenter the window, as well as empty vectors to store the indexes corresponding to where pixels in the left and right lines are. Finally, a vector of points, as large as the number of windows, will contain the center points of the lane and they will constitute the trajectory followed by the vehicle.

```
int margin = 40;
size_t minpix = 20;

vector<vector<int>> left_line_inds, right_line_inds;
vector<Point> center_line(nwindows);
```

Listing 3.25: Window parameters.

Finally we step through the windows one by one: starting from the initial position, the first window counts how many pixels are inside it: if the number of pixels exceeds a given threshold, the window is re-centred where the new peak is located, otherwise it remains in the same position.
The sides of both the left and right windows are defined as shown in 3.26:

```
for (int window = 0; window < nwindows; window++)
{
int win_y_low = binary_warped.rows - (window + 1) *
    window_height/2;
int win_y_high = binary_warped.rows - window *
    window_height/2;

int win_xleft_low = leftx_current - margin;
int win_xleft_high = leftx_current + margin;

int win_xright_low = rightx_current - margin;
int win_xright_high = rightx_current + margin;
```

Listing 3.26: Definition of the margins of the window.

The step after their identification is the drawing of the windows themselves, according to which line is present, if only one or both.
Furthermore, in this control, the centre of the lane is also marked as the average between the positions of the two windows, or of the single window plus/minus half the width of the lane, depending on whether only the left or only the right line is present. The result of this calculation is stored in `center_lane`.
Half of the lane is computed each time the algorithm is run: the variable `CenterLane`, in fact, is a vector of 100 elements representing it. At the beginning of the program, it is empty and, during the execution, it fills up with elements when both lines are identified. When it is full, the oldest elements are discarded to make room for new ones, if and only if the new values fall within a tolerance of 10% of the average of the existing values.
The value used to calculate the centre of the lane is the average value among the 100 elements of the vector, by means of the custom function `computeMean` and whose result is stored in the variable `meanCenterLane`.
The just described part of the algorithm is shown in listing 3.27.

```
if (both_lines==0) {
    if(left_lane == true){
        rectangle(birdsEyeView, Point(win_xleft_low,
           win_y_low), Point(win_xleft_high, win_y_high),
            Scalar(255, 255, 255), 2);
        center_lane.push_back(Point(leftx_current+
           meanCenterLane,win_y_low));
    }
    if(right_lane == true){
        rectangle(birdsEyeView, Point(win_xright_low,
           win_y_low), Point(win_xright_high, win_y_high)
           , Scalar(255, 255, 255), 2);

        center_lane.push_back(Point(rightx_current-
           meanCenterLane,win_y_low));
    }
}
else {
    rectangle(birdsEyeView, Point(win_xleft_low,
       win_y_low), Point(win_xleft_high, win_y_high),
       Scalar(255, 255, 255), 2);
    rectangle(birdsEyeView, Point(win_xright_low,
       win_y_low), Point(win_xright_high, win_y_high),
       Scalar(255, 255, 255), 2);

    center_lane.push_back(Point((leftx_current +
       rightx_current)/2,win_y_low));

    if ((((rightx_current - leftx_current)/2>0.95*
       meanCenterLane)&&((rightx_current - leftx_current)
       /2<1.05*meanCenterLane))||(allNonZero(CenterLane)
       ==0)) {
        rotate(CenterLane.begin(), CenterLane.begin() +
           1, CenterLane.end());
        CenterLane.back() = (rightx_current -
           leftx_current) * 0.5;
    }
}

if (!center_lane.empty() && center_lane[0] == Point(0, 0)
   ) {
    center_lane.erase(center_lane.begin());
}
```

Listing 3.27: Window design and lane centre definition.

After having identified the non-zero pixels inside the windows following the first one, we had to cycle both left and right lines in parallel using the `Pragma` directive to find the final peak in both sides and to mark consequently the final and correct coordinates of lane lines. This is shown in listing 3.28.

```
    vector<int> good_left_inds, good_right_inds;

    #pragma omp parallel for
    for (size_t i = 0; i < nonzeroy.size(); ++i)
    {
        if (nonzeroy[i] >= win_y_low && nonzeroy[i] <
           win_y_high &&
        nonzerox[i] >= win_xleft_low && nonzerox[i] <
           win_xleft_high) {
        #pragma omp critical
        good_left_inds.push_back(i);
        }
        if (nonzeroy[i] >= win_y_low && nonzeroy[i] <
           win_y_high &&
        nonzerox[i] >= win_xright_low && nonzerox[i] <
           win_xright_high) {
        #pragma omp critical
        good_right_inds.push_back(i);
        }
    }
```

Listing 3.28: Peak research.

Indexes associated to the new peaks are added to vectors because, if at the next loop more pixels than the minpix threshold are found, the next window is re-centered based on their average position, as written in listing 3.29.

```
        if (good_left_inds.size() > minpix) {
            int mean_x = 0;
            for (int i : good_left_inds){
                mean_x += nonzerox[i];
            }
            leftx_current = mean_x / good_left_inds.size();
        }

        if (good_right_inds.size() > minpix) {
            int mean_x = 0;
            for (int i : good_right_inds) {
                mean_x += nonzerox[i];
            }
            rightx_current = mean_x / good_right_inds.size();
        }
    }
```

Listing 3.29: Comparison with the minimum threshold for the found peaks.

The final result of the applied sliding window algorithm is shown in Figure 3.8, where it is visible the formation of the central reference as the union of all central points formed by the average distance between the left and right windows placed at a certain height.

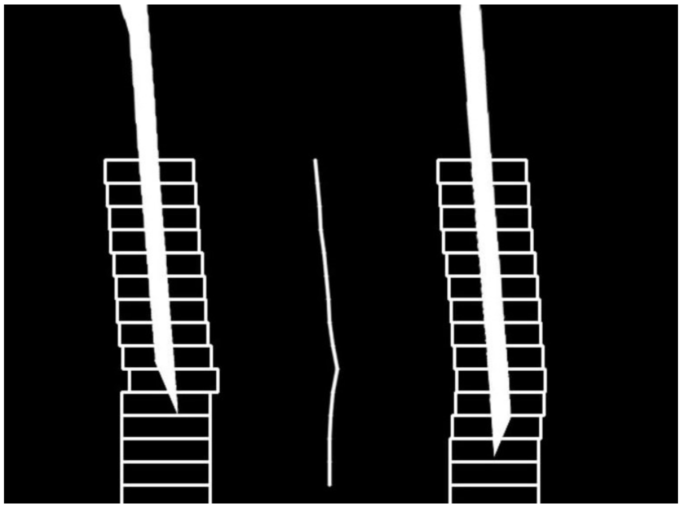


Figure 3.8: Result of the sliding window algorithm, performed by considering 15 windows.

Chapter 4

Trajectory Planning and Lane Keeping algorithms

Lane trajectory, like lane detection, influences the performance of a lane keeping system, as the correct reference line is obtained when the lane boundaries are located and identified correctly.
The method typically adopted in the industry to realize dynamically the trajectory is the pipeline planning, that includes two strategies:

1. **Global route planning** [30], that generates a path from the starting point to the final destination, knowing the environment in advance. Even though this method provides the global optimal solution, the path is not always directly executable because this algorithm considers the autonomous vehicle as a particle and it expands the grid map of a distance comparable to the radius of the car itself but, even supposing that it will not collide with obstacles, actually the way-points result to be very close to objects, producing a possible contact with them. However, there exist strategies, based on the minimization of the risk of collision, that make this algorithm work properly.

2. **Local trajectory planning** [31], that instead generates a short-term trajectory based on the information from the surroundings. In particular, a local occupancy grid is used to take into account the new position of the car and of the obstacles, these last mainly moving. At each iteration, the grid is updated with the new positions in the perception zone, such that the autonomous vehicle always knows about the presence of possible obstructions.

In this thesis, however, the trajectory planning algorithm simply consists in the definition of a continuous path based on a set of way-points coming from the coordinates of the left and right lane lines, since the pipeline planning method requires a huge computational effort and many heuristic functions. More precisely, after having collected at each iteration the positions of the points in both lines, the mean value between them is calculated and then it is added to the left x-coordinate or subtracted to the right x-coordinate to obtain exactly the center lane position along the horizontal axis, that is where the ego car has to be in time.
The code written in 4.1, 4.2 and 4.3 shows how the center lane coordinates are computed, according to what the camera sees.

```
center_lane.push_back(Point((leftx_current +
    rightx_current)/2, win_y_low));
```

Listing 4.1: Computation of the center lane position having both lines.

```
center_lane.push_back(Point((leftx_current +
    meanCenterLane), win_y_low));
```

Listing 4.2: Computation of the center lane position having only the left line.

```
center_lane.push_back(Point((rightx_current -
    meanCenterLane), win_y_low));
```

Listing 4.3: Computation of the center lane position having only the right line.

- *leftx_current* and *rightx_current* are two integer quantities in which we store the current positions, corresponding to the highest peaks, to be updated for each window.
- *meanCenterLane* is a double variable that stores the result of the mean value operation, as explained in the previous chapter immediately before the code 3.27.
- *win_y_low* corresponds to the low y-coordinate of the windows of the sliding window algorithm, and it is the same for both left and right.
- *center_lane* is the vector of points where we store the way-points used to implement the trajectory planning.

It is important to distinguish *trajectory planning* from *path planning*: the first generates a path as a function of time, while the second is a sequence of points that can be followed at any speed.
Given these definitions, the trajectory is described by:

$$(x_k, y_k, t_k), \qquad k = 0, 1, 2, ..., N$$

where (x_k, y_k) are the coordinates of the reference trajectory and t_k is the time.

4.1 Lane Keeping

The reference path and the vehicle position often suffer from inaccuracy due to disturbances on the road, e.g. the artificial light or light reflection, or to data availability, as it may sometimes happen that some information about the coordinates miss because of the camera inclination or other real physical limitations.
To solve these problems, a lane keeping system is needed to generate a reference path and to obtain an accurate vehicle position on the road [32]: by implementing this control, in fact, it is possible to keep the autonomous car in its lane safely, avoiding any sort of unexpected behavior.
In research, the most common strategies [33] adopted for this type of discipline are:

1. **Pure Pursuit Controller:** it is a path tracking algorithm that estimates the angular velocity command used to move the autonomous car from its initial position, while the linear velocity is assumed to be constant. With this control, the vehicle, whose reference point is on the rear axle, has to follow some look-ahead points in front of it, that are moved by the algorithm on the path based on the actual position of the car until the last point of the course is reached.
   The look-ahead point is at a fixed distance ahead of the vehicle and it has to proceed to that point using a certain steering angle that has to be computed.
   The pure pursuit controller [34] is based on the Kinematic Single Track Model theory (summarized in 4.1.1) and it ignores dynamic forces and slip at the wheels.
   An example of how the control works is illustrated in Figure 4.1.

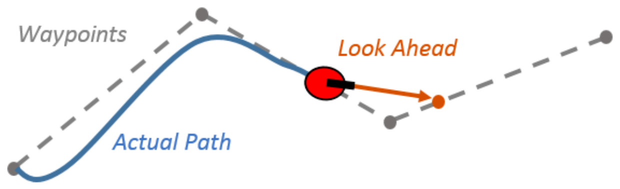


Figure 4.1: Pure Pursuit Controller.

2. **Stanley Controller:** it is similar to the Pure Pursuit Controller, but this time the vehicle reference point is set on the front axle. Moreover, its control law considers both a kinematic and dynamic model: the first is appropriate for path following at low speeds, as inertial effects are small, the second instead is good at high speeds, where inertia is no more negligible.

3. **Model Predictive Control:** it is used in autonomous driving to predict the future evolution of a system by generating a series of control actions that make the car follow the desired trajectory. It focuses on both a longitudinal (speed) and lateral (steering) control, that estimate the future states of the vehicle.
   The optimisation problem is based on the minimisation of a quadratic cost function over a finite prediction horizon under control effort and lateral error limitations, as written in equation 4.1:

$$\min_{\delta} \int_0^{\infty} (e^T Q e + u^T R u)\, dt \tag{4.1}$$

   s.t. $|\delta|<\delta_{max}$ and $|e_y| < l$, where:

   - e is the tracking error.
   - u is the control input.
   - e_y is the lateral error with respect to the center lane.
   - δ_{max} is the maximum steering angle.
   - l is the constraint on the maximum lateral deviation from path.

   The limitations have to be respected during the optimisation and they can refer to state variables, e.g. the speed, or to control variables, e.g.

the acceleration.
The implementation of this control algorithm consists in the definition of a prediction horizon N, that specifies the number of future control intervals the MPC controller must consider by prediction, and of a control horizon T, that is the number of future steps to optimize. Typically, $T \leq N$.
In this way, it is possible to forecast the state evolution of the system according to the chosen prediction horizon and minimise the cost function by satisfying at the same time the constraints.

4. **PID Control:** the Proportional Integral Derivative controller [35] is a control loop feedback technique that computes the difference between a desired set-point and the actual output from a process on which the correction is applied. This controller block diagram is depicted in Figure 4.2 and it consists of three parts:

   (a) **Proportional term:** it contains a proportion K_p of the current error value. This part computes the corrective response to the process, but the response exists only if the error is present.

   (b) **Integral term:** it considers all past error values and accumulates them over time. In this way, the integral term K_i increases until the error reaches zero. At this point, the error does not grow anymore, but if a new error is present after having applied the proportional control, the integral term again attempts to eliminate it and the proportional effect decreases consequently.

   (c) **Derivative term:** denoted as K_d, it estimates the future trend of the error, based on its current rate of change and it adds a damping effect to the system. In fact, the faster the change, the greater the damping effect.

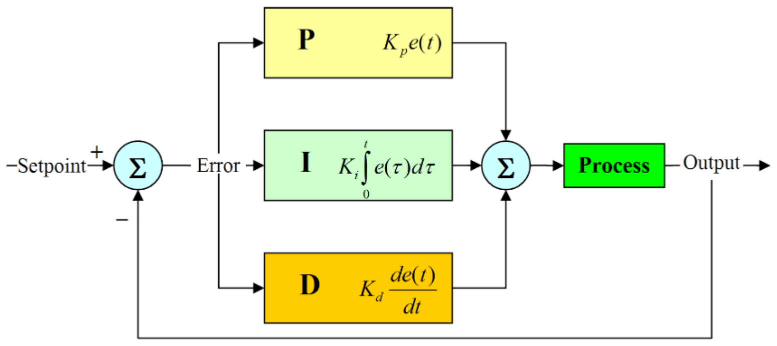


Figure 4.2: PID Controller scheme.

4.1.1 Kinematic Single Track Model

The model [36] shown in Figure 4.3 is used to implement lane keeping algorithms and it works by considering a look-ahead distance and by ignoring unknown parameters. However, it is limited to low speed applications, as the kinematic model does not really represent the movement of the vehicle in all situations, where some contributions become no more negligible.
The continuous time equations in the inertial frame are:

$$\dot{x} = v \cdot cos(\psi + \beta)$$

$$\dot{y} = v \cdot sin(\psi + \beta)$$

$$\dot{\psi} = \frac{v}{l_r} \cdot sin(\beta)$$

$$\dot{v} = a$$

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

where:

- (x, y) are the coordinates of the center of mass in the (X,Y) frame.
- ψ is the car heading.
- v is the speed.

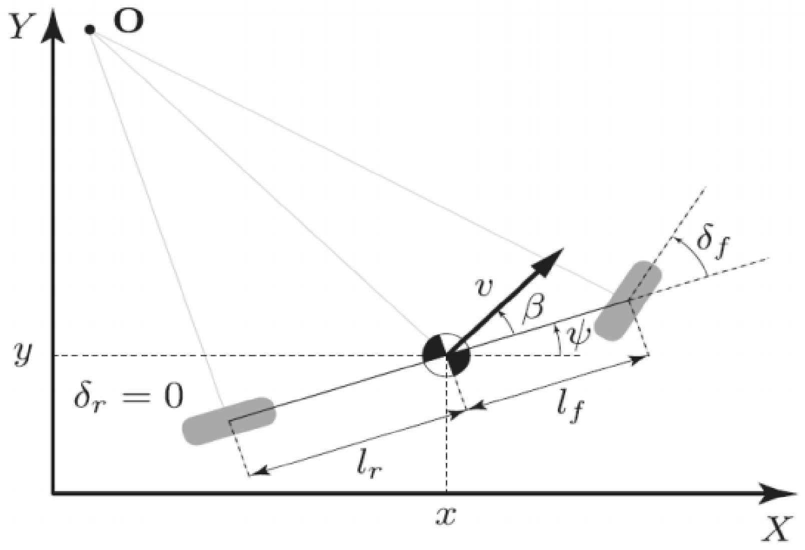


Figure 4.3: Kinematic bicycle model

- l_r and l_f are the distance from the center of mass to the rear and to the front axles respectively and they are the only parameters to determine.
- β is the angle between the center of mass velocity vector and the longitudinal axis of the car.
- a is the acceleration of the center of mass and it is consistent with the velocity.

The input variables are the front steering angle δ_f and a.

4.2 Lane Keeping in our application

Even though the Model Predictive Control is the most common technique applied to implement a lane keeping strategy, for this thesis we decided to realize a quite different algorithm, as the MPC requires a huge computational effort that the Raspberry Pi is not able to handle within the required time-frame.
Since the goal is to make the car run on its lane without leaving it, the states that describe the behavior of the vehicle in time are:

1. **Yaw angle:** treating the car as a rigid body, it is the angle between its x-axis and the path along which it is travelling.
   An example of how this angle is computed is shown in Figure 4.4.
2. **Lateral offset:** it is the difference between the center of the frame, giving the position of the car, and the lateral lane line.

The outputs instead are the velocity and the steering angle, that are increased or decreased according to where the car is placed on the road.

4.2.1 Gain scheduling control

In order to implement a gain scheduling control based on the yaw angle of the car with respect to the lane, it is first necessary to calculate the linear regression using the points that resulted to be the reference of the trajectory [37], [38] from which we can then derive its slope with respect to the car, and therefore to the vertical of the acquired frame. This is equivalent to finding the slope of the car with respect to the lane.
The equation of a straight line is written in equation 4.2:

$$y = q + mx \tag{4.2}$$

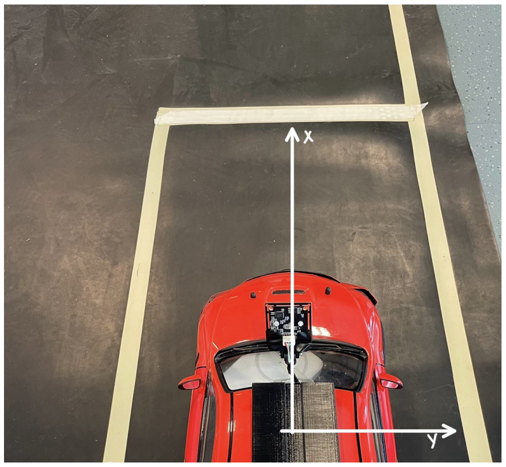
where the intersection with the y-axis, in a linear regression, is given by 4.3

$$q = \frac{[(\sum y)(\sum x^2) - (\sum x)(\sum xy)]}{[n(\sum x^2) - (\sum x)^2]} \tag{4.3}$$

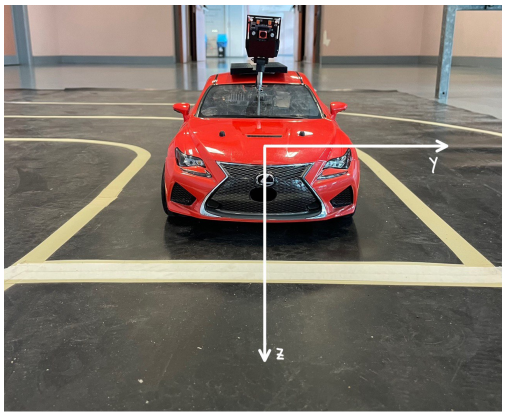
and the slope is found with the formula shown in 4.4

$$m = \frac{[n(\sum xy) - (\sum x)(\sum y)]}{[n(\sum x^2) - (\sum x)^2]} \tag{4.4}$$

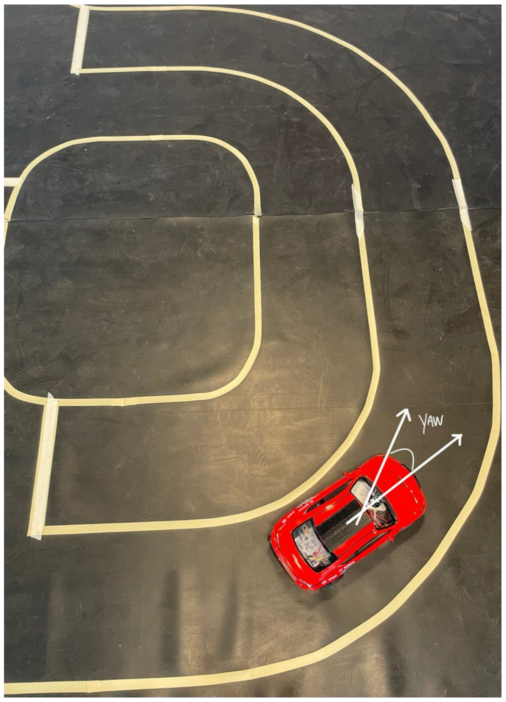
Our variable of interest is the second one, i.e. the slope, as it is fundamental for the computation of the correct vehicle positioning.
After some algebra calculations, they correspond to the lines of code shown in listing 4.4: the x and y coordinates refer to the position of the center lane.



(a)Position of the car in the x-y plane.



(b)Position of the car in the y-z plane.



(c)Representation of the yaw angle.

Figure 4.4: Yaw angle measurement.

```
for (size_t i=0; i<center_lane.size();i++){
  sum_x = sum_x + center_lane[i].x;
  sum_y = sum_y + center_lane[i].y;
  sum_x2 = sum_x2 + center_lane[i].x*center_lane[i].x;
  sum_xy = sum_xy + center_lane[i].x*center_lane[i].y;
  }

  med_x = sum_x/n;
  med_y = sum_y/n;
  med_x2 = sum_x2/n;
  med_xy = sum_xy/n;

  var = med_x2 - med_x*med_x;

  cov = med_xy - (med_x*med_y);
  m = -cov/var;
}
```

Listing 4.4: Computation of the slope of the regression line.

According to the result of m, it is then possible to define how much the steering angle has to be increased: typically, if the absolute value of the slope is large, the correction on the angle is very small and it increases as the slope decreases, as shown in listing 4.5. This occurs because, if the value of the slope is small, it means that it is more crushed on the x-axis of the frame, meaning that the wheels are not straight, but they tend to be crooked with respect to the desired position. On the contrary, when the slope is large, the wheels of the car are almost parallel to the y-axis of the frame, and, consequently, the vehicle moves along the correct direction.
The maximum adjustment stops at 20 degrees, as the limitation on the steering angle for the car is fixed at 25 degrees: in this way, by adding the contribution of the lateral offset, we never reach values greater than the threshold itself.

```
if  (abs(m)>16){
    steeringvalue = 1;
      }else if(abs(m)>14){
    steeringvalue = 2;
      }
      else if(abs(m)>12){
    steeringvalue = 3;
      }else if(abs(m)>10){
    steeringvalue = 4;
            }else if{

            ...

            }else if(abs(m)>3.15){
    steeringvalue = 17;
      }
      else if(abs(m)>2){
    steeringvalue = 20;
      }
```

Listing 4.5: Correction on the steering angle given the slope.

4.2.2 Control based on the lateral offset

The lateral offset is defined for both left and right sides. The algorithm is computed taking into account the position of the lane center line, that is subtracted to the half of the frame width. In this way, in fact, it is possible to know exactly where the car is placed on the road: if it is centered, if it is on the right or on the left.
The result is then compared to specific thresholds and, on the basis of this, the steering value previously obtained is corrected with an increase or decrease in the angle.
Listing 4.6 shows how the correction is implemented to make the car turn left when it is moving away from the center line. A similar control is also applied to the right.

```
if(steeringvalue >= -10) {
        if((center_lane[3].x-birdsEyeView.cols/2) < -0.05*(
            rightx_current-leftx_current))
      {
          steeringvalue = steeringvalue - 1;
      }

        if((center_lane[3].x-birdsEyeView.cols/2) < -0.10*(
            rightx_current-leftx_current))
      {
          steeringvalue = steeringvalue - 1;
      }

        ...

        if((center_lane[3].x-birdsEyeView.cols/2) < -0.50*(
            rightx_current-leftx_current))
      {
          steeringvalue = steeringvalue - 3;
      }
 }
```

Listing 4.6: Correction on the steering angle given the lateral offset. The car turns left.

We set $steeringvalue >= -10$ because we add the contribution of the lateral offset on the correction only if the error due to the yaw angle is small, otherwise there is a too large impulse in the steering command that the car swerves, being the maximum allowed steering angle equal to +/-25 degrees. Between the two sources of error, the most significant is due to the heading, as the car has to be able to stay within its lane, independently of the fact that it follows the center line. The capability of performing this action, in fact, depends on the yaw rate, that is on the slope of the lines that the camera sees according to where the car is placed on the track. The lateral offset also influences the behavior of the vehicle, but of a small quantity, since it depends on its heading: when the center of mass of the car is perfectly centered with respect to the lane center line, the lateral offset is zero.

Figure 4.5 depicts how the yaw angle and the lateral offset are measured.

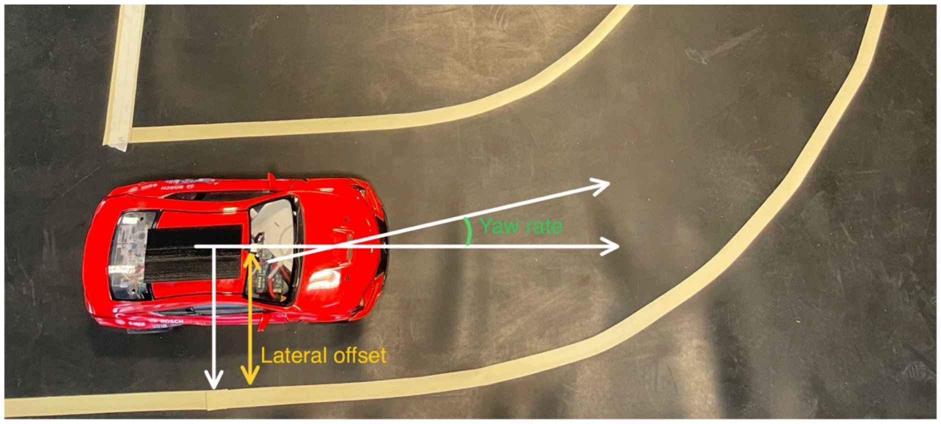


Figure 4.5: Representation of the lateral offset and of the yaw angle.

4.2.3 Speed evaluation

If the yaw angle and the lateral offset are two quantities that influence the computation of the steering angle, this last is used to define instead the velocity at which the vehicle has to run.
When the steering angle is small, in fact, the car follows its trajectory with more or less the same initial speed, because it is supposed to go almost straight, but when the angle starts to increase, the speed is decreased by a certain amount to still ensure that the car stays within the lane and to have more time to make up for the error that the car has with respect to the reference.
For values higher than 6 degrees, instead, the velocity is set to the minimum, as reported in listing 4.7.

```
// refSpeed is the initial speed.

if(abs(steeringvalue)<2)
  {
// we check that we do not fall below the minimum speed
    if(refSpeed*0.9>=0.068)

    {
      speedvalue = refSpeed*0.9;
    }else{ speedvalue = 0.068;}
  } else if(abs(steeringvalue)<3)
  {
    if(refSpeed*0.8>=0.068)
    {
      speedvalue = refSpeed*0.85;
    }else{ speedvalue = 0.068;}
  } else if(abs(steeringvalue)<4.5)
  {
    if(refSpeed*0.75>=0.068)
    {
      speedvalue = refSpeed*0.75;
    }else{ speedvalue = 0.068;}
  }else if(abs(steeringvalue)<6)
  {
    speedvalue = 0.07;
  }
            else
            speedvalue = 0.07;
```

Listing 4.7: Computation of the velocity of the car.

4.2.4 Stop sign management

The last part belonging to the lane keeping section is the stop sign recognition, as the car has to be able to identify, on its own, situations for which it must stop.
By referring to the `crosswalk_stop` flag defined in chapter 3, it is possible to know if the vehicle is going to meet an horizontal line or not, simply by reading the logical value of that variable. If it is true, we set a new boolean called `flag` that, when it is 1, it makes the car advance backwards before the stop line if it stops a bit after. The vehicle waits for k-counter ticks, during which it stays still. When they expire, the velocity is set to a fixed value,

0.075 in our case, and all the flags and counters are set to zero.
The code associated to this part is shown in listing 4.8.

```
if(crosswalk_stop==true)
    {
        flag = true;
    }
    if(flag == true){
        speedvalue = -speedvalue-0.05;
        k = k+1;
        if (k>13 && k<100){
        speedvalue = 0.0;

    }

    if(k==100)
    {
        speedvalue = 0.075;
        crosswalk_stop=false;
        k=0;
        flag = false;
    }
```

Listing 4.8: Speed control in the presence of a stop line.

4.3 Speed and steering threads

As for the frame acquisition part, it is necessary to implement a thread function also for the communication between the Raspberry Pi and the Nucleo board that manages the transmission of the commands to the steering and traction motors, as reported in listing 4.9.

```
void communicateWithSTM32(double &steeringvalue, double &
    speedvalue) {
     while (!stopCapture) {

      traction_fnc(speedvalue);

      turn_fnc(steeringvalue);

         }
     this_thread::sleep_for(chrono::milliseconds(1));
}
```

Listing 4.9: Thread function for the speed and steering commands.

`traction_fnc` and `turn_fnc` are the functions where we set the velocity and the steering angle of the car respectively and they are described in listing 4.10.
The command is a variable of type array char with a predefined size, that is written to the serial port by means of the `write()` function of the `unistd.h` library.
It is important to remark that, after having written a command to the serial port, this last has to be flushed in order to reset the serial buffer and to avoid an accumulation of commands that inhibit the correct communication between the two boards, resulting otherwise in wrong actions performed by the vehicle.

```
 void turn_fnc(double steeringvalue){
    // command #2 sets the steering angle
    char com[320];
    sprintf(com, "#2:%lf;;", steeringvalue);
    write(serial_fd, com, strlen(com));
    usleep(0.4e4);

    tcflush(serial_fd,TCIOFLUSH);
}

 void traction_fnc(double speedvalue){
    // command #1 sets the speed
    char com[320];
    sprintf(com, "#1:%lf;;", speedvalue);
    write(serial_fd, com, strlen(com));
    usleep(0.4e4);

    tcflush(serial_fd,TCIOFLUSH);
}
```

Listing 4.10: turn_fnc and traction_fnc for sending the commands.

Chapter 5

Discussion

In this chapter we are going to analyze the results obtained during the tests and to discuss some parameters that influence the performance of the car.
In particular, a series of tests were carried out to understand the current drawn at a certain speed, the actual values of the speed once a certain input command was given and also the trend of the output as a function of the input values.
These studies were useful to understand the non-idealities of the vehicle, giving the possibility to choose the most suitable steering and traction control ranges for our application.
The analysis performed on the current required by the motor led to the decision to adopt a different battery compared to the one initially supplied with the kit, because it could not handle the current demands from both the traction and steering motors at the same time.

5.1 Lane keeping on a straight road

The tests were carried out on a straight 6-metre long road to maintain the lane at different speeds in absence of intersections and stop signs. The results were very good at low speeds, i.e. less than 1 km/h, but the non-negligible problem was the torque transmitted by the traction motor that was not sufficient to overcome all the mechanical friction inside the car and between the car and the road, avoiding it to move.
At higher speeds, i.e. around 3km/h, instead, the car was not always able

to maintain the lane because of the limited computational power of the Raspberry Pi board to generate a real-time reference sufficiently correct for the vehicle to follow, leading to some skidding.

5.2 Stop at stop signs

Tests were also carried out to calibrate the control on the stop signs: these involved testing, both in a bend and on a straight stretch, how the car reacted when approaching this kind of scenario.
On the straight stretch the car resulted to be quite accurate in stopping at stop lines. Stop lines immediately after the curves, instead, were not always recognised because of some delays between the movement of the car and the frames continuously acquired by the camera and processed by the brain board. Moreover, under a certain threshold of angular coefficient, when the car was going out from a curve, the camera could not always read the stop lines, but, unfortunately, changing the angle threshold below which stop lines were recognised could lead to excessive errors in the recognition of them, even when there were not any.

5.3 Approach to intersections

The control approach in absence of lines consists in letting the car continue to go straight.
As far as this type of test is concerned, the car always reacted rather well at junctions coming after a straight road, but a little bad at junctions placed after a roundabout.
All in all, however, this type of control tended to give good responses.

5.4 Current request

As the data-sheet says, the motor requires a current of maximum 3 A at no load condition and equal to 15 A maximum in case of a load connected to it. However, the inrush current is a bit higher when it is started because of the

switching on of the motor itself to overcome the idle condition.
This current is sometimes also called “locked rotor current” because, like Stan Turkel states in his blog [39], “the current necessary at startup to begin the rotation of a non-rotating, de-energized motor shaft, is very similar to the extreme current draw experienced for the moments when a motor is overloaded to the point of seizing.”
To demonstrate what has been just stated, tests were performed to assess the initial current and the results are reported in tables 5.1, 5.2 and 5.3. During these experiments, the correct trajectory is still followed, even at high velocities.
The same test was repeated three times, but with different speed values, as the main objective is to demonstrate that, by changing the speed, the initial current is very large: the motor, in fact, draws at start-up a current that is approximately 20 A, regardless of the command set, while it diminishes a bit when the car moves along the track. As a consequence, the battery has to be capable of powering continuously the motor, otherwise malfunctions occur due to the lack of required energy.

Table 5.1: Current measurements at start-up. First test.

Speed command	**Current [A] (test 1)**
0.07	16.024
0.08	19.440
0.09	14.051
0.1	10.505
0.11	8.985
0.12	11.511

Table 5.2: Current measurements at start-up. Second test.

Speed command	**Current [A] (test 2)**
0.07	19.371
0.08	8.349
0.09	13.754
0.1	13.469
0.11	15.616
0.12	10.860

Table 5.3: Current measurements at start-up. Third test.

Speed command	**Current [A] (test 3)**
0.07	16.686
0.08	12.217
0.09	10.911
0.1	11.435
0.11	7.859
0.12	11.641

5.5 Speed

The speed of the car is measured in cm/s, but the value sent to the Nucleo board by means of the Raspberry Pi has not that unit of measurement.
To obtain it, in fact, it is needed to convert the command to the corresponding speed by simply applying the formula

$$v = \frac{s}{t},$$

where the distance s was fixed to 6 meters and the time t was kept with a chronometer. Making reference to the previous tests, the corresponding values related to the time are now shown in tables 5.4, 5.5 and 5.6: the initial speed of the car is zero and, as the car is started, it immediately reaches the given set value.
With these results, we can state that, the smaller the speed command, the higher the time needed to travel the path, and consequently the smaller the

physical speed too.
In the third measurement, in fact, the vehicle requires 72.2 s before it reaches the end of the straight road, and this parameter is strictly influenced by the given command, because, if at the beginning the inrush current is high even at low speeds and the battery is not completely charged, the car requires more time to start-up and run, since it has also to deal with the presence of friction due to the rubber track.
Another incisive factor, in fact, is the charge of the battery at which the vehicle is, and hence the current delivered to the motor, which is why it was decided to perform several tests even at different charge levels.
On the contrary, when the command is increased, the time is reduced and good results are obtained for values greater than 0.1 for which the different times are comparable, while speeds increase approximately by 10.
The plots corresponding to the measured quantities are shown in Figures 5.1 and 5.2. Figure 5.1 recalls a hyperbola, being the formula of the speed a hyperbolic function.

Table 5.4: Speed measurements. First test.

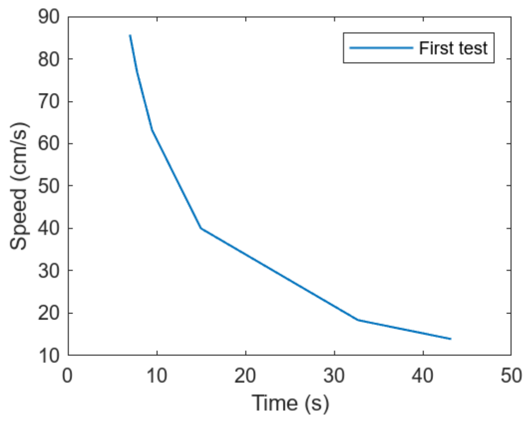
Speed command	**Time [s] (test 1)**	**Distance [m]**	**Speed [cm/s]**
0.07	43.2	6	13.88
0.08	32.7	6	18.35
0.09	15	6	40.00
0.1	9.5	6	63.16
0.11	7.8	6	76.92
0.12	7	6	85.71

Table 5.5: Speed measurements. Second test.

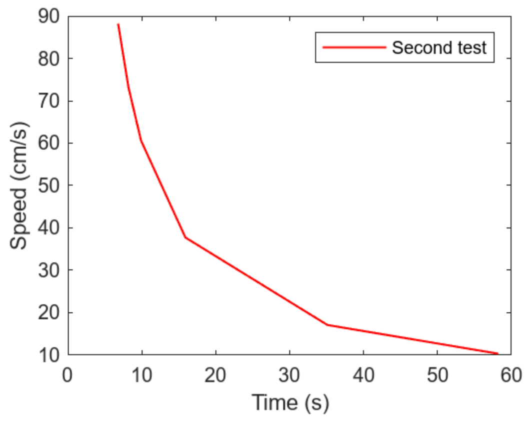
Speed command	**Time [s] (test 2)**	**Distance [m]**	**Speed [cm/s]**
0.07	58.2	6	10.31
0.08	35.1	6	17.09
0.09	15.9	6	37.73
0.1	9.9	6	60.60
0.11	8.2	6	73.17
0.12	6.8	6	88.23

Table 5.6: Speed measurements. Third test.

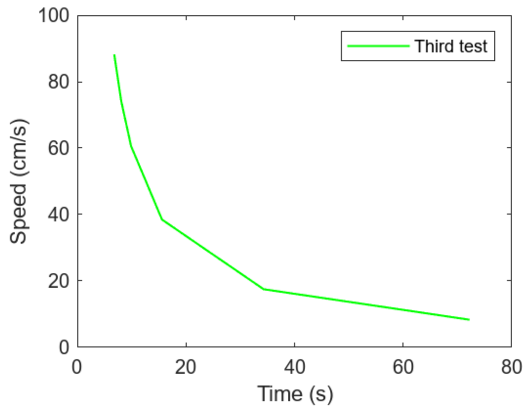
Speed command	**Time [s] (test 3)**	**Distance [m]**	**Speed [cm/s]**
0.07	72.2	6	8.31
0.08	34.3	6	17.50
0.09	15.6	6	38.46
0.1	9.9	6	60.60
0.11	8.1	6	74.07
0.12	6.8	6	88.23



(a)Speed test 1.

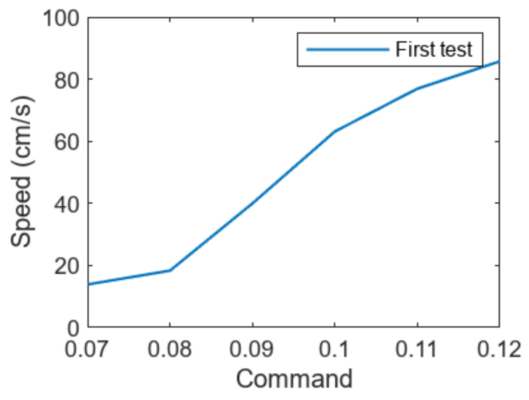


(b)Speed test 2.

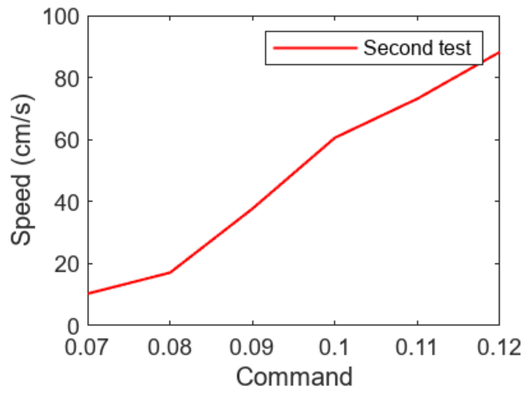


(c)Speed test 3.

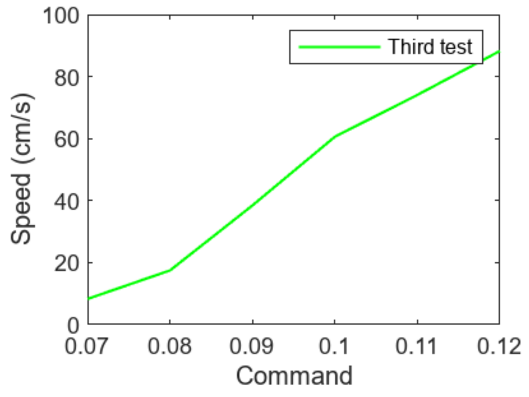
Figure 5.1: Plots speed vs time.



(a)Speed test 1.



(b)Speed test 2.



(c)Speed test 3.

Figure 5.2: Plots speed vs command.

From Figure 5.2, it is possible to see that the speed is a piece-wise function. Its domain, in fact, is divided into three sub-intervals on which it is defined differently:

1. from command 0.07 to command 0.08: for varying inputs, the outputs change of a small quantity.
2. from command 0.08 to command 0.1: for varying inputs, the outputs change more.
3. from command 0.1 to command 0.12: the characteristic operates as in the first sub-interval.

From one interval to the other, there is a change of the slope in the output velocity with respect to the command and the reason of these distinct behaviours may lay in the fact that, at low speeds, the frictional resistance and inertia may affect more the traction motor. Furthermore, the control of a PWM motor may be less precise and thus it will be less sensitive to input variations. For increasing speeds, instead, the mechanical friction in the gears and bearings increase, as well as the aerodynamic drag, that increments quadratically with respect to the speed, even though our car is a scale model and therefore it does not have all that much incident surface area.
All the reasons mentioned above refer to non-ideal behaviours of the real model that have to be taken into account however in order to achieve an even more precise control of the car.

5.6 Steering angle

The steering angle, as already mentioned in chapter 2, is constrained in the window [-25°,25°], because the servo is not able to turn more than those angles in both directions resulting in a completely stuck car.
The tests were performed always by sending the speed command 0.1 and measuring the distance, i.e the diameter of the semicircle, associated with different angles and then, on the basis of it, obtain the curvature radius and the conversion to the corresponding real steering value.
A representative image is depicted in Figure 5.3 that shows how the diameter is calculated.

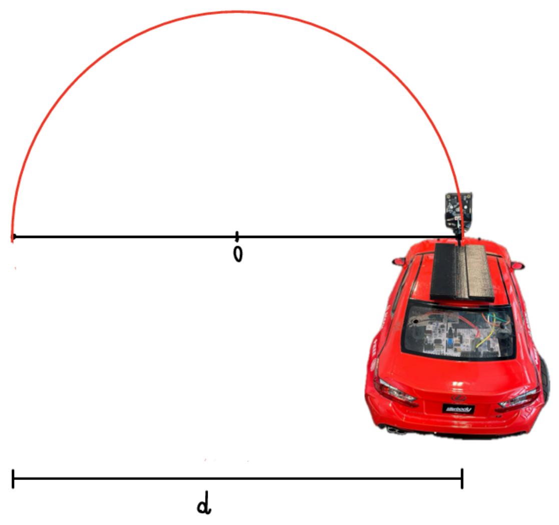


Figure 5.3: Representation of how the steering angle is computed.

Starting from the highest steering value command and decreasing it by 5 at every run, the following results reported in table 5.7 have been obtained:

Table 5.7: Measurement of the distance.

Steering command [°]	**Distance(1) [m]**	**Distance(2) [m]**	**Mean distance [m]**
25	1.170	1.170	1.170
20	1.460	1.430	1.445
15	2.190	2.250	2.220
10	3.380	3.130	3.255

Two tests were performed with the same set of controls to be sure to be repeated by the car.
The results of both tests differed slightly in terms of the distance travelled by the vehicle, and this was mainly due to the non-ideality of the steering control.

The curvature radius is easily got by $\frac{mean\ distance}{2}$ and this quantity is useful for the computation of the real steering angle, always expressed in degrees, and whose formula is

$$\arctan(\frac{axial\ distance}{curvature\ radius})$$

where the axial distance is defined as the distance from the front axle to the rear axle and it is equal to 26 cm in our case.
The results are shown in table 5.8.

Table 5.8: Measurement of the real steering angle.

Curvature radius [m]	**Steering angle [°]**
0.585	23.962
0.723	19.792
1.110	13.183
1.628	4.567

As a result, the real steering angle values are approximately the same as the ones sent to the car; the only discrepancy of some degrees is visible for commands smaller than 10 degrees, for which the corresponding angle is almost the half.
A possible explanation always lies in the fact that the low torque delivered, this time by the steering motor, is not sufficient to overcome friction.
Furthermore, from plots 5.4 and 5.5, it is possible to state that, for angles ranging from 10 to 20 degrees, the characteristic is almost linear, and this is more accentuated in the real case, while for values in between 20 and 25 degrees, the slope changes and the relationship between input and output can not be treated as in the ideal case.
However, as one can imagine, too small steering angles are associated with large travelled distances, as both graphs show.

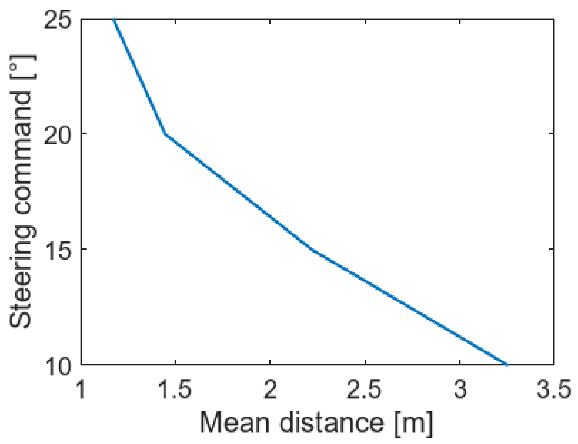


Figure 5.4: Steering command vs mean distance.

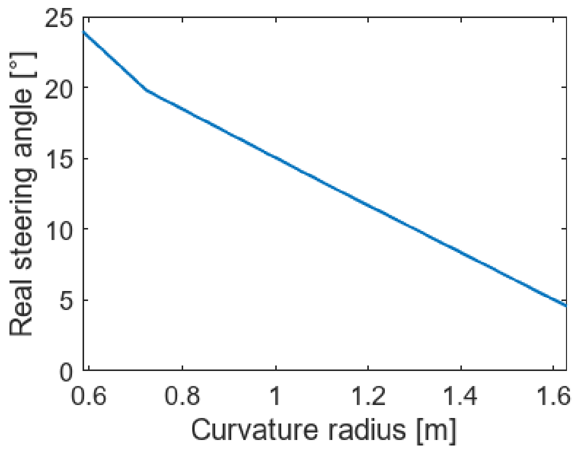


Figure 5.5: Real steering angle vs curvature radius.

Chapter 6

Conclusion

This thesis work has led to several conclusions that will be discussed in the next lines.

To start, the predetermined goal, i.e. the lane keeping, was successfully achieved despite the several challenges and problems encountered during the project.

A first critical issue was encountered with the compatibility between systems that are usually used for prototypes or models: as mentioned in chapter 2, the update of the Raspberry operating system with substantial improvements regarding the management of the image acquisition led to the incompatibility with the camera through the flat cable, forcing us to change that hardware. From this fact, it can be affirmed that, when working with systems, usually used for prototypes, having their own operating system, it may sometimes be necessary to change or adapt the hardware components with which they normally work because they can become obsolete. A future improvement related to this aspect can be of course considered, in particular when the desire or request is to obtain a much more performing system, but in this case, new upgrades are needed and, consequently, new strategies have to be adopted in the case incompatibility with other new components rise.

Another critical issue was found with the development of the control and image processing algorithms, since the more robust and accurate they are, the more intensive from a computational point of view these actions are for a microprocessor. As a consequence, depending on the degree of robustness and accuracy one wants to reach, it is essential to have a controller capable of processing the calculations in a time comparable or even shorter than the time required for the task to be handled.

The final aim of this thesis was the development of the lane keeping for a model in which the maximum reachable speed was experienced to be about 4 km/h. To achieve this goal, a board with a good computational power was required, with quite a lot of work to optimize the algorithm in many ways to effectively achieve the intended goal, also thanks to the adoption of the C++ programming language, that resulted to be a good choice in terms of computations, memory allocations and code management.

Finally, the study and tests carried out on this 1-to-10 scale model car have shown that such control is difficult to perform on a model car because there are more pronounced non-idealities in proportion to a real car. For example, the active steering of a car will have a much more accurate actuator, perhaps even through a feedback control, and it is much more powerful, so that it is not affected a lot by friction in the steering action, that instead happens in the case of the model car.
Another big difference concerns that a real car has different types of sensors that help to control better what are the states of the system, such as the speed and acceleration, and therefore one is able to act much more accurately by having more real data available, and this is a fact that in the model, at present, is not possible to do though.
Not surprisingly, in order to measure the actual steering angles and speeds, many tests had to be carried out, and the results were very often in a range of not too small values, or they even varied quite a bit depending on the conditions in which the model car was placed, not only environmental, namely the kind of floor on which the car was running, but also internal within the car, such as the traction motor overheating, which also made the performance worse, or the loosening of some screws after only a few hours of operation that compromised the kinematics of the car.
The experimentation part, in fact, resulted to be more complex than expected because the attitude of the car was sometimes quite detached from the theoretical reasoning.
From this it can be concluded that, as much as the model may be an approximation of the real behavior of an automobile on the road, and the control may be in some ways more simplified, on a real car it seems to be

easier to perform such controls that are for sure accurate and effective, but obviously the degrees of difficulty increase, as we talk about vehicles that have to travel in real scenarios with real road users and every mechanical, electrical and environmental contribution has to be considered, because any phenomena can not be neglected to safeguard people safety.

Chapter 7

Future Developments

In this thesis project some improvements have already been made and they are related to the three main areas characterizing an autonomous car: electronics, computer science and mechanics, but others are however possible.
From an electronic point of view, the upgrade consists in the adoption of the brain board Raspberry Pi 5 instead of the previous model provided by Bosch Future Mobility Challenge, but, to make the code running better and to let the system be more performing, it is possible to evaluate new electronic boards capable of handling real-time applications and a huge computational effort for all the computer vision algorithms that need to be implemented in order to realize an autonomous driving. In the case of a possible change, however, it is a good practice to check before the compatibility between operating systems, in particular when dealing with objects coming from different suppliers.
From a computer science point of view, the original code running on Raspberry was completely written in Python, while the one on the Nucleo was and still is in C++.
The choice to switch from Python to C++ also for the brain is justified by the several advantages reported in table 2.2 and by the easiness in the memory allocation and de-allocation when dealing with vectors of objects. Furthermore, it does not need to be interpreted, as it is already understood by the micro-controller thanks to the low-level access to the hardware. Finally, the several features and libraries existing in C++ allow the use of this programming language for many applications in embedded systems.
On this side, a future development can exist in terms of written algorithms, as for sure the implementation of both the image vision and of the lane keeping

parts need to be improved: for the first, the elaboration of the acquired frame and how it is treated can be ameliorated and made even faster. Also some new vision controls and steps can be included, depending on the requests. For the second, instead, it is possible to think about the implementation of the Model Predictive Control, as it is the most common technique in autonomous driving, but there should be the guarantee that the electronic board on which this part is developed has sufficient computational power such that is able to handle concurrently both the lane keeping and the lane detection steps.
Another possible improvement could be to change the camera back to a wide-angle camera, which captures a larger space in front of the camera: on the one hand, this might seem pejorative, as the more things you see, the more noise there is, but on the other hand, by going for the action of selecting only certain parts of the frame, and thus with specific regions of interest, this problem is solved. Acquiring an image with more elements can also be useful for developing algorithms for recognising traffic signs or other things outside the road, so as to have an even closer approximation to real situations.
Finally, as a last remark, improvements are also possible from a mechanical point of view, but they only consist in fixing better some screws and mechanical pieces, as well as in changing old components that compromise the actions of the car, as it is the case of the servo and traction motors: the first because its pinion is not exactly fixed, and consequently it is never really set to zero in the case of straight wheels, but it has a bit of offset, the second because it overheats after some long and fast runs, forcing the motor to enter in protection zones for short periods.

Bibliography

[1] L. Theodorakopoulos C. Karras P. Kranias N. Schizas G. Kalogeratos D. Tsolis A. Giannaros, A. Karras. Autonomous vehicles: Sophisticated attacks, safety issues, challenges, open topics, blockchain, and future directions. *Journal of Cybersecurity and Privacy (JCP)*, vol. 3, 2023.

[2] A. Rajpurkar M. Chahal N. Kumar G. Prasad Joshi W. Cho D. Parekh, N. Poddar. A review on autonomous vehicles: Progress, methods and challenges. *Electronics 2022*, 2022.

[3] Frits Klaver. *The economic and social impacts of fully autonomous vehicles*. Compact Magazine — KPMG, 2020.

[4] Jan Wienrich. Autonomous driving: The steps to self-driving vehicles. *ZF Magazine*, 2022.

[5] SAE Blog. Sae j3016 levels of driving automation. Technical report, SAE International, 2021.

[6] I Barabás et al. Current challenges in autonomous driving. *Conf. Ser.: Mater. Sci. Eng.*, 2017.

[7] American Control Conference Self-Driving Car Student Competition. `https://www.quanser.com/community/student-competition/2024-student-self-driving-car-competition/`.

[8] VDI Autonomous Driving Challenge. `https://www.vdi-adc.de/`.

[9] NXP Cup EMEA's Intelligent Car Racing. `https://nxpcup.nxp.com/`.

[10] Autonomous Driving Challenge CARNET Barcelona. `https://carnetbarcelona.com/autonomous-driving-challenge/`.

[11] Bosch Future Mobility Challenge (BFMC). https://boschfuturemobility.com/.

[12] Self Driving Challenge. https://www.selfdrivingchallenge.nl/.

[13] AutoDrive Challenge II by SAE. https://www.sae.org/attend/student-events/autodrive-challenge.

[14] Indy Autonomous Challenge. https://www.indyautonomouschallenge.com/.

[15] Formula Student Driverless. https://www.imeche.org/events/formula-student/team-information/fs-ai.

[16] https://www.raspberrypi.com/products/raspberry-pi-5/.

[17] https://www.geeksforgeeks.org/difference-between-c-and-c/.

[18] https://www.geeksforgeeks.org/difference-between-c-and-python/.

[19] https://www.geeksforgeeks.org/difference-between-python-and-c/.

[20] https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/.

[21] https://docs.opencv.org/3.4/index.html.

[22] https://www.sameskydevices.com/product/resource/amt10-v.pdf.

[23] https://eu.mouser.com/datasheet/2/389/vnh5019a_e-1852574.pdf.

[24] https://asset.conrad.com/media10/add/160267/c1/-/en/002141322DS00/datasheet-2141322-reely-standard-servo-rs-610wp-mg-analogue-servo-gear-box-material-metal-connector-system-jr.pdf.

[25] R. Muthalagu S. Fernandes, D. Duseja. Application of image processing techniques for autonomous cars. *Proceedings of Engineering and Technology Innovation*, vol. 17:pp. 01–12, 2021.

[26] S.M. Yang C.Y. Kuo, Y.R. Lu. On the image sensor processing for lane detection and control in vehicle lane keeping systems. *Sensors 2019*, vol. 19, 2019.

[27] `https://docs.rs-online.com/88e0/A700000008916437.pdf`.

[28] `https://www.geeksforgeeks.org/multithreading-in-cpp/`.

[29] Surya Teja Karri. Hough transform. 2019. `https://medium.com/@st1739/hough-transform-287b2dac0c70`.

[30] D. Zhao Y. Shu Z. Zhang S. Wang Y. Sun, J. Yang. A global trajectory planning framework based on minimizing the risk index. *Actuators 2023*, 2023.

[31] C. Francis H. Shraim A. Said, R. Talj. Local trajectory planning for autonomous vehicle with static and dynamic obstacles avoidance. *IEEE International Conference on Intelligent Transportation Systems*, pages pp. 410–416, 2021.

[32] S. Wang M. Foo M. Silverio Fernandez N. Horri, O. Haas. Mode switching control using lane keeping assist and waypoints tracking for autonomous driving in a city environment. *Transportation Research Record: Journal of the Transportation Research Board*, pages pp. 712–727, 2022.

[33] Yan Ding. Three methods of vehicle lateral control: Pure pursuit, stanley and mpc. 2020. `https://dingyan89.medium.com/three-methods-of-vehicle-lateral-control-pure-pursuit-stanley-and-mpc-db8cc1d32081`.

[34] `https://it.mathworks.com/help/nav/ug/pure-pursuit-controller.html`.

[35] Madhu Dev. Tuning pid controller for self-driving cars. 2021. `https://medium.com/@madhusudhan.d/tuning-pid-controller-for-self-driving-cars-3813f7f18eb0`.

[36] G. Schildbach F. Borrelli J. Kong, M. Pfeiffer. Kinematic and dynamic vehicle models for autonomous driving control design. *2015 IEEE Intelligent Vehicles Symposium (IV)*, 2015.

[37] https://www.geeksforgeeks.org/linear-regression-formula/.

[38] Adriano Gilardone. La regressione lineare. https://adrianogilardone.com/retta-di-regressione-lineare/.

[39] Stan Turkel. Understanding motor starting (inrush) currents. 2017. https://www.jadelearning.com/blog/understanding-motor-starting-inrush-currents-nec-article-430-52/.