Politecnico di Torino

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Validation of ADAS algorithm using virtual vehicle approach

Supervisor:
Prof. Massimo Violante

Author:
Leonardo Locardo

December 2019

"One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man."

Elbert Hubbard

Acknowledgment

It was a period of profound learning, not only scientifically, but also personally. Writing this thesis has had a strong impact on my personality. I would like to say a word of thanks to all the people who supported and helped me during this period.

Special thanks goes to the supervisor of my thesis, Prof. Massimo Violante for his valuable advice and teachings and above all, for having provided me with all the tools I needed to take the right path and complete my thesis.

Finally, I would like to thank my dearest people to whom I dedicate this work of mine: my friends who, in good times and bad times, have supported me in moments of despair and joy until I reached my goal, my parents and my uncle for their wise advice and their ability to listen to me, my brother who always encouraged me and spent part of his time re-reading and discussing the drafts of my work with me.

You have always been by my side.

Abstract

Nowadays autonomous driving represents a sector in continuous growth and expansion, to which the majority of car manufacturers are actively participating.
Among the many features provided from the earliest levels of automation, there is both the detection of objects and the detection of lane lines, which contribute not only to perceive the surrounding environment in a clearer and more precise way but also to increase road safety. This choice was dictated by the fact that these two systems allow to implement some functions typical of autonomous driving like the automatic lane keeping or the emergency braking.
In particular, this thesis project deals with two algorithms that use computer vision to identify lane lines on the road and an algorithm that uses a deep learning approach to identify objects placed in front of the vehicle.
In the first part of the thesis project, a simple and fast algorithm for identifying lane lines (SLLD), realized in Python, has been analyzed, focusing attention both on the detailed description of the various steps to be able to perform the entire procedure and on its advantages and disadvantages.
In the second part, instead, a different method of identifying lane lines has been introduced and examined (ALLD) which is slower and more complex than the previous one but at the same time more efficient. Furthermore, the algorithm also offers the ability to extract useful information from the images coming from the camera. Specifically, in addition to detecting the pixels of the lane lines in a robust way, it shows on the screen the curvature of the lane and the position of the vehicle with respect to the center of the lane (lateral offset). The latter algorithm was initially developed in python and then it was translated into C++ language.
In the third and final part, the object detection system, which represents the backbone of many practical applications of artificial vision especially for autonomous cars, has also been discussed.
After an initial and careful comparison between different algorithms for identifying objects with their relative problems, an algorithm has been studied, YOLOv3 (written in Python and later translated into C++ language).
It uses a deep learning approach able to recognize objects in urban environments with crossings, vehicles, pedestrians and traffic lights using the convolutional neural networks (CNN).
In these 3 parts, computer vision algorithms and deep learning algorithms have been extensively exploited in combination with different libraries, including OpenCV, to be able to process and test each image frame and/or video coming from the virtual environment of CARLA simulator. CARLA simulator is a free open-source simulator based on Unreal Engine, which realistically reflects different driving scenarios, designed to be able to validate and verify autonomous driving systems.

To understand the reliability and robustness of the algorithms, the validation process was performed within CARLA Simulator through a collection of tests carried out using certain datasets. These datasets were generated based on two main parameters: lighting and weather conditions. The advantages and disadvantages of the individual algorithms were listed and summarized according to the different operating circumstances, highlighting the application limits.

Keywords: autonomous driving vehicles, automotive virtual simulation systems, perception, computer vision, lane detection, object detection, convolutional neural networks, deep learning.

Contents

Acknowledgment ... 2

Abstract ... 3

Contents ... 5

List of Figures ... 7

List of Tables ... 10

Abbreviations ... 11

1. Introduction ... 13

1.1 Classification Levels of driving automation ... 14

1.2 ADAS – Advanced Driver Assistance Systems ... 15

1.3 ADAS Development Process ... 19

1.4 Passive vs Active Safety and Laws ... 23

1.5 The GOLD (Generic Obstacle and Lane Detection) System ... 25

2. CARLA Simulator ... 27

2.1 Configuring the Simulation ... 30

2.2 Loading and Running the Simulation ... 34

3. Computer Vision and Sensor Fusion ... 38

3.1 Kalman Filter Prediction Equation ... 45

3.2 Kalman Filter Update Equation ... 45

3.3 Features of a camera ... 47

4. Lane Lines Detection ... 50

5. Simple Lane Line Detection ... 53

6. Advanced Lane Line Detection ... 61

7. Validation of Lane Detection Algorithms and Comparison ... 80

7.1 Light Conditions ... 81

7.2 Weather conditions ... 87

7.3 Final Consideration ... 92

8. Object Detection ... 97

8.1 Concepts behind CNN ... 98

8.2 Deep Learning approach ... 100

8.3 YOLOv3 model ... 104

8.4 YOLOv3 Object Detection Algorithm 107
9. Validation of YOLOv3 object detection algorithm 109
9.1 Light Conditions 110
9.2 Weather conditions 116
9.3 Final Consideration 120
Conclusions and Future Improvements 121
Bibliography 123
Appendix CARLA Simulator 126
Appendix Structure of project files 129

List of Figures

Figure 1.1 - The Road Safety Atlas for each European country [2] 13
Figure 1.2 - SAE J3016 Levels of Automated Driving [4] 15
Figure 1.3 - SAE levels of vehicle automation in relation to ADAS functions [6] 17
Figure 1.4 - Position of the ADAS sensors on a vehicle - Side View [7] 18
Figure 1.5 - Position of the ADAS sensors on a vehicle - Top View [8] 19
Figure 1.6 - V diagram model 20
Figure 1.7 - Typical HIL testing process 21
Figure 1.8 - VeHIL Scheme [13] 22
Figure 1.9 - Active and Passive Safety systems cooperate in the time interval before, during, and after incident [9] 23
Figure 1.10 - On the left, the pedestrians would die, on the right instead the passengers would die 24
Figure 2.1 - CARLA Simulator [17] 27
Figure 2.2 - View of a street in the Town 2, which shows 4 different weather conditions. From 1 to 4: sunny day, day with rain in the morning, day in the morning immediately after the rain and day at sunset [17] 28
Figure 2.3 - CARLA Client Server Communication with CARLA modules representation [18] 28
Figure 2.4 - Block Diagram of the entire simulation test 29
Figure 2.5 - 2D map visualization of Town04 in Carla Simulator 37
Figure 2.6 - 3D view of Town04 in Carla Simulator 37
Figure 3.1 - Most common types of ADAS sensors and their ranges 39
Figure 3.2 - Example of different types of sensors positioning in a self-driving car [19] 39
Figure 3.3 - Example of Lidar sensor data application for autonomous driving car [22] 41
Figure 3.4 - Example of ultrasonic sensor positioning [13] 42
Figure 3.5 - Comparison between different types of sensors [23] 43
Figure 3.6 - Example of Sensor Fusion [24] 44
Figure 3.7 - Kalman Filter Flowchart 46
Figure 3.8 - Computer Vision Processing Pipeline 47
Figure 3.9 - Pinhole Camera Model and Reference Frame [26] 49
Figure 5.1 - A graphical representation of 2D Gaussian distribution with mean at origin and $\sigma = 1$ [32] 54
Figure 5.2 - Mapping of a line to the Hough space 57
Figure 5.3 - Mapping from Image Space to Hough Space 58
Figure 5.4 – SLLD Algorithm's pipeline 60
Figure 6.1 - Main comparison between C++ and Python [33] 62
Figure 6.2 - Input image (snapshot) taken from Carla Simualtor Town04 using ADAS_scenario.py 63

Figure 6.3 - Thresholded binary gradients images 65
Figure 6.4 - Combined Binary Gradient 66
Figure 6.5 - Example of different color spaces [35] 67
Figure 6.6 - Thresholded binary color channel images 68
Figure 6.7 - Combined binary color 69
Figure 6.8 - Summary scheme to create a combined binary image 69
Figure 6.9 - Combined binary image 70
Figure 6.10 - Image ROI (Region of Interest) 71
Figure 6.11 - Warped image 72
Figure 6.12 - Binary warped image 72
Figure 6.13 - Histogram Peaks 73
Figure 6.14 - Sliding Window method 75
Figure 6.15 - Skipping sliding window 76
Figure 6.16 - Graphic representation of the radius of curvature R of a curve K and of its osculating circle which is tangent in the point P 77
Figure 6.17 - Graphical representation of the parameters used to calculate the distance from the center of the lane 78
Figure 6.18 - Final output image 79
Figure 7.1 - Input image during the morning 82
Figure 7.2 – Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning 82
Figure 7.3 - Input image during the sunset 84
Figure 7.4 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the sunset 84
Figure 7.5 - Input image during the night 86
Figure 7.6 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the night 86
Figure 7.7 - Input image during the morning with fog 88
Figure 7.8 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning with fog 88
Figure 7.9 - Input image during the morning with soft rain 90
Figure 7.10 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning with soft rain 90
Figure 7.11 - Input image during the morning with heavy rain 91
Figure 7.12 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning with heavy rain 91
Figure 7.13 - Input image during the morning with curved lines 92
Figure 7.14 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning with curved lines 92
Figure 8.1 - CNN architecture with layers [40] 100
Figure 8.2 - YOLO model [45] 103

Figure 8.3 - Full YOLO network...104
Figure 8.4 - YOLOv3 architecture [47]..105
Figure 8.5 - Performance comparison between different object detection algorithms using COCO dataset [46]..106
Figure 9.1 - Input image during the morning...111
Figure 9.2 - Output image applying YOLOv3 during the morning................................111
Figure 9.3 - Input image during the sunset...113
Figure 9.4 - Output image applying YOLOv3 during the sunset....................................113
Figure 9.5 - Input image during the night..115
Figure 9.6 - Output image applying YOLOv3 during the night......................................115
Figure 9.7 - Input image during the morning with fog...117
Figure 9.8 - Output image applying YOLOv3 during the morning with fog..................117
Figure 9.9 - Input image during the morning with rain..119
Figure 9.10 - Output image applying YOLOv3 during the morning with rain................119
Figure 9.11 - Graphs that summarize the results obtained during the tests...................120

List of Tables

Table 1.1 - Overview of safety related ADAS applications [5] 17
Table 7.1 - Pros and Cons of SLLD (Simple Lane Line Detection) algorithm 93
Table 7.2 - Advantages and Disadvantages of ALLD (Advanced Lane Line Detection) algorithm 94
Table 7.3 - Example of SLLD algorithm performance testing 95
Table 7.4 - Example of ALLD algorithm performance testing 95

Abbreviations

ABS	Antilock Braking System
ACC	Adaptive Cruise Control
ADAS	Advanced Driver Assistance System
AEBS	Autonomous Emergency Breaking Systems
ALC	Adaptive Light Control
ALLD	Advanced Lane Line Detection
API	Application Programming Interface
ASIRT	Association for Safe International Road Travel
BEV	Birds-Eye-View
CARLA	Car Learning to Act
CNN	Convolutional Neural Network
DIL	Driver In the Loop
DSP	Digital Signal Processor
ECU	Electronic Control Unit
ESP	Electronic Stability Program
EuroNCAP	European New Car Assessment Programme
FCA	Forward Collision Avoidance
FMECA	Failure Modes, Effects and Analysis
GNSS	Global Navigation Satellite System
GOLD system	Generic Obstacle and Lane Detection system
GPS	Global Positioning System
HIL	Hardware In the Loop
HLS	Hue, Lightness, Saturation
HMIs	Human Machine Interface
HOG	Histogram of Oriented Gradients
HSV	Hue, Saturation, Value
IC	Integrated Circuits

ICA	**Intersection Collision Avoidance**
IMU	**Inertial Measurement Unit**
IPM	**Inverse Perspective Mapping**
KF	**Kalman Filter**
LCA	**Lane Change Assistant**
LIDAR	**Light Detection and Ranging**
LKA	**Lane Keeping Assistant**
LV	**Level**
MCU	**MicroController Unit**
MIL	**Model In the Loop**
MPU	**Microprocessing Unit**
NPCs	**Non-Playable Characters**
RADAR	**Radio Detection and Ranging**
RCP	**Rapid Control Prototyping**
ReLU	**Rectified Linear Unit**
RGB	**Red, Green, Blue**
ROI	**Region Of Interest**
SAE	**Society of Automotive Engineers**
SIL	**Software In the Loop**
SLLD	**Simple Lane Line Detection**
SSD	**Single Shot Detector**
SVM	**Support Vector Machine**
VEHIL	**Vehicle Hardware In the Loop**
YOLO	**You-Only-Look-Once**

1. Introduction

A reality that is becoming increasingly popular in the last few years is undoubtedly self-driving cars, which represent one of the most demanding challenges that man is facing in the automotive sector. When we talk about self-driving cars, we refer to all those vehicles that, through various sensors used to perceive the surrounding world, are able to manage, independently, different situations and tasks replacing human intervention during the transport phase. In fact, creating cars that are able to signal and implement decisions on their own using artificial intelligence is not easy at all.
Among the various advantages, which derive from the use of these types of vehicles, the most important is certainly to improve road safety. Indeed, evaluating the data, according to **ASIRT** “every 365 days approximately 1.25 million people lose their lives (Figure 1.1) due to road accidents, with an average of around 3,287 deaths per day, considering that more than half of these deaths involve relatively young human among the 15 years and 44 years” [1].

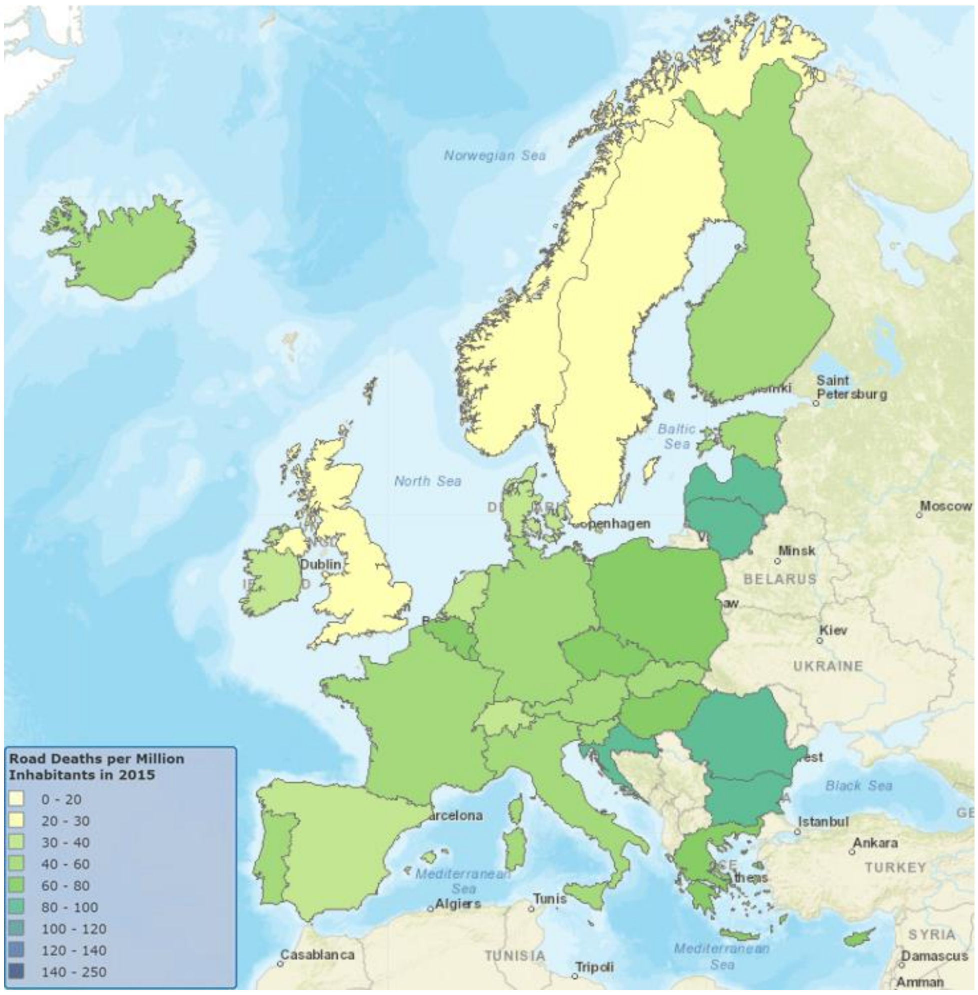


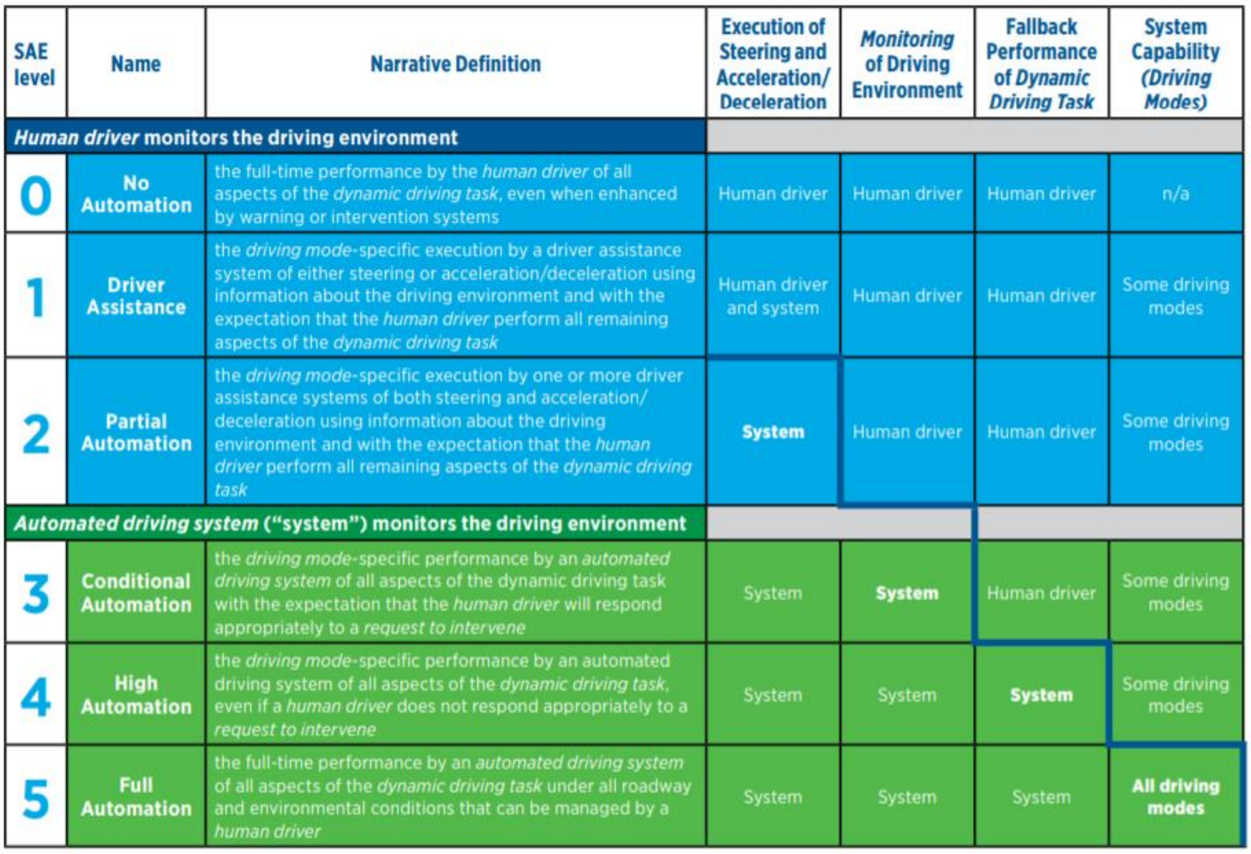
Figure 1.1 - The Road Safety Atlas for each European country [2]

Most of these road accidents are caused mainly by the carelessness of the driver or by the use and abuse of alcohol with consequent strokes of sleep, long duration driving, or even by very adverse weather conditions.
In order to prevent and reduce considerably both the risk of road accidents and pollution with the consequent improvement in road safety, countermeasures are needed that must be implemented and used to avoid all those behaviors that lead to accidents.
For this reason, autonomous driving cars have been introduced not only to optimize road safety and transport efficiency, but also to improve environmental quality.
This can be done by installing sensors such as radar, lidar and GPS that interact with the on-board computer system (ECU) on the car allowing it to have a general snapshot of the surrounding environment to make the best decision without human intervention.
This process can therefore be defined as a sort of real "driving revolution", and although autonomous driving cars are advancing inexorably, revolutionizing the world of mobility, progress is gradual and not without risks and difficulties, both from a practical, administrative, cultural and economic point of view. It is as plain as technologies related to self-driving vehicles are evolving more and more together with the development of closely interconnected sectors such as electrification, infrastructures and artificial intelligence.
According to some optimistic estimates [3], in the near future, the adoption of self-driving cars could not only reduce road accidents by 90% with the direct effect of saving thousands of lives, but also reduce carbon dioxide emissions up to 60% improving and optimizing the overall traffic flow, decreasing travel times.
However, the fact that the human factor always plays a crucial role in the adoption of new technologies should be considered. With the introduction of a system that provides a high density for the exchange of data such as that of connected cars, privacy will also be put to the test. Governments will have to introduce new laws concerning autonomous vehicles for the protection of people.
Furthermore, the ethics underlying the management and decision-making software for autonomous driving must be taken into account: in potentially critical or fatal conditions, it may be necessary to make the choice, as complex as it is fundamental, between preserving the life of the passengers inside the vehicle and minimizing the number of victims outside the vehicle.

1.1 Classification Levels of driving automation

The SAE, Society of Automotive Engineers (a worldwide association of engineers, technicians and experts in the aerospace and automotive sectors) has established and defined six levels of autonomous driving from level zero (no automation) to level five (full automation).

Each level lists certain characteristics on the greater or lesser degree of automation of the car in relation to the importance of driver attention and intervention which spans from manually controlled vehicle (LV.0) to vehicles that do not require human intervention (LV.5) (see Figure 1.2). This subdivision allows learning not only how the automation levels for autonomous driving cars work, but also the taxonomy and the definitions that distinguish the different levels.
Note that towards the end of 2019, vehicles belonging to level three (such as the new audi A8 or new Tesla model 3) of automated driving will be produced and marketed.



SAE level	Name	Narrative Definition	Execution of Steering and Acceleration/ Deceleration	*Monitoring* of Driving Environment	Fallback Performance of *Dynamic Driving Task*	System Capability *(Driving Modes)*
***Human driver* monitors the driving environment**						
0	No Automation	the full-time performance by the *human driver* of all aspects of the *dynamic driving task*, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a
1	Driver Assistance	the *driving mode*-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	Human driver and system	Human driver	Human driver	Some driving modes
2	Partial Automation	the *driving mode*-specific execution by one or more driver assistance systems of both steering and acceleration/ deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	**System**	Human driver	Human driver	Some driving modes
***Automated driving system* ("system") monitors the driving environment**						
3	Conditional Automation	the *driving mode*-specific performance by an *automated driving system* of all aspects of the dynamic driving task with the expectation that the *human driver* will respond appropriately to a *request to intervene*	System	**System**	Human driver	Some driving modes
4	High Automation	the *driving mode*-specific performance by an automated driving system of all aspects of the *dynamic driving task*, even if a *human driver* does not respond appropriately to a *request to intervene*	System	System	**System**	Some driving modes
5	Full Automation	the full-time performance by an *automated driving system* of all aspects of the *dynamic driving task* under all roadway and environmental conditions that can be managed by a *human driver*	System	System	System	**All driving modes**

Figure 1.2 - SAE J3016 Levels of Automated Driving [4]

1.2 ADAS – Advanced Driver Assistance Systems

The Advanced Driver Assistance Systems (ADAS) are adjuncts to autonomous driving systems, and represent all those intelligent safety systems integrated on board vehicles that contribute to not only increasing driving comfort but also increase safety levels leaving the driver free from the execution of specific tasks thanks to automated systems that provide warnings that may be visual or auditory.

In fact, these particular safety systems have been designed with the aim of reducing collisions and above all fatal accidents, providing some advanced technological systems that provide to warn the driver of possible problems surrounding the vehicle.
The main goal of ADAS is to help make car passengers and the surrounding environment safer for the driver while ensuring traffic efficiency. Of great importance is the fact that some of these systems are extensively and daily used in today's cars, such as ESP (Electronic Stability Control), ABS (Antiblockiersystem), ACC (adaptive cruise control), turning on and off automated lights, incorporate GPS / traffic alerts.
Nevertheless some of these systems (the most complex) need sensors and actuators to be able to analyze each situation in its entirety and fully satisfy the purpose dedicated to them, performing the required actions, such as automated parking, automate braking in the event of an accident, blind spot detection, adaptive cruise control, lane keeping. The classification with relative descriptions of some ADAS is reported in Table 1.1.

ADAS function	Definition and/or description	Level	Impact
navigation system	provision of vehicle positioning, route calculation and route guidance	I+S	long
adaptive cruise control (ACC)	System that adaptively adjusts the speed and distance automatically in relation to the vehicle traveling in the same lane, braking or accelerating depending on the behavior of the vehicle in front	C	long
adaptive light control (ALC)	dynamic aiming headlamps and situation adaptive lighting	S	long
vision enhancement	assist driver vision capability in adverse lighting and weather conditions by providing enhanced visual information	S	long
legal speed limit assistance	assist driver in keeping within (static or dynamic) legal speed limits	I/W/C	long
curve speed assistance	assist driver in keeping within an appropriate and safe speed in curve manoeuvres	W/C	long
dangerous spots warning	assist driver by providing information or warning on a dangerous location (based on accident statistics) at inappropriate speed	I/W	long
stop and go (S&G)	assist driver by taking over full vehicle control in congested stop-and-go traffic at low speeds (automated lane keeping and platooning)	C	long
forward collision avoidance (FCA)	warn driver in case of an imminent forward collision, and/or provide automatic control of the vehicle in such situations	W/C	long
lane keeping assistant (LKA) (= lane departure avoidance)	assist driver to stay in lane (on unintentional lane departure or road departure) by warning (e.g. by rumble strip sound) and/or semi-control of the vehicle (by force feedback on the steering wheel) and/or full control	W/C	lat

ADAS function	Definition and/or description	Level	Impact
lane change assistant (LCA) (= lateral collision avoidance)	for change-of-lane manoeuvres, provide information about vehicles in adjacent lanes, and/or warning for potential collision, and/or vehicle control in case of imminent collision	I/W/C	lat
intersection collision avoidance (ICA)	avoid collisions at intersections by warning or control based on: - radar and/or vision - vehicle positioning and short-range communication (requires all participating vehicles to be equipped)	W/C	long
intersection negotiation	regulate vehicle traffic at intersections based on vehicle positioning and short-range communication in all participating vehicles	C	long
autonomous driving	fully automated driving in controlled motorway situations at all speeds by full lateral and longitudinal control	C	lat+long

Source: Partly based on NextMAP Consortium 2000.
Level: I = information, W = warning, C = control, S = support
Impact: long = longitudinal, lat = lateral

Table 1.1 - Overview of safety related ADAS applications [5]

The relationship, which describes how ADAS system functions adapt to various SAE levels of autonomous driving, is brought back in Figure 1.3.

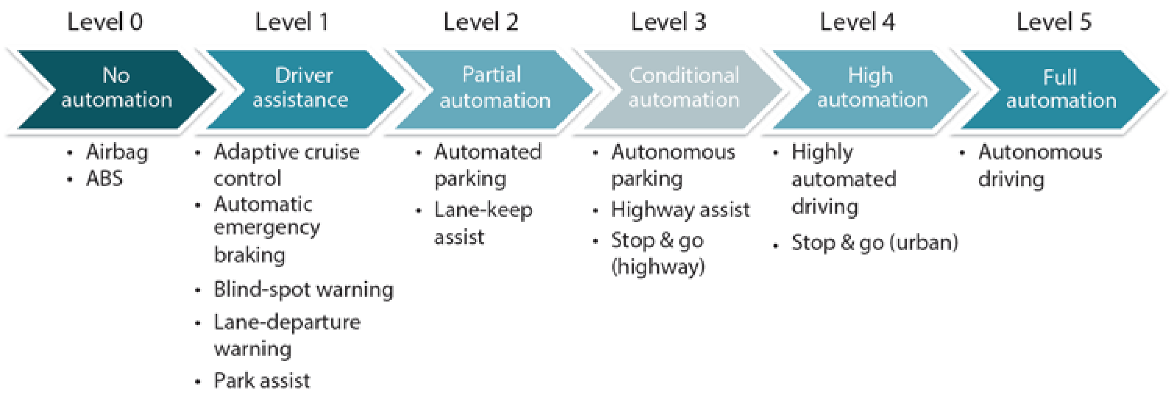


Figure 1.3 - SAE levels of vehicle automation in relation to ADAS functions [6]

An automated guidance system also involves the use of functional blocks that are able to perform particular and important tasks in order to obtain a reliable peripheral vision system as a whole. Among these, surely the following are of fundamental importance:

- sensors that capture and collect various information on the surrounding environment, sending signals to the driver in case of risks

- high-performance integrated circuits (ICs) for communication, which are able to cope with increasing complexity in the acquisition and processing of information, especially for predictive algorithms
- high-performance microprocessors (MPUs) or digital processors (DSPs) for the analysis of the data coming from the sensors in such a way as to ensure the main functions of the car with autonomous driving, that is the perception of the environment around the vehicle, the accurate analysis of information in real time and the consequent decision-making aimed at activating certain functions
- microcontrollers (MCUs) that are able to activate and control functions

Surely a key role is played by different types of sensors located in the vehicle system capable of eliminating "blind spots", as they must not only detect objects such as mobile or fixed infrastructures, but also vehicles and human beings such as pedestrians.
Moreover, they must operate at a speed very close to the real-time with a high efficiency to prevent any type of accident especially on the motorway/freeway where cars travel at high speed. Cameras, broadband radar sensors, ultrasounds and LIDARS are technologies that contribute to the important task of making the driving scenario safer, guaranteeing accuracy and satisfactory responsiveness.
As already listed above, modern cars incorporate a series of ADAS systems that take advantage of the capabilities of different sensors, which according to their task, are located at strategic points of the vehicle in order to obtain a high fidelity in the measurements.
Figure 1.4 and 1.5 show the list of sensors used to perform ADAS functions with their range and their relative position with respect to the vehicle.

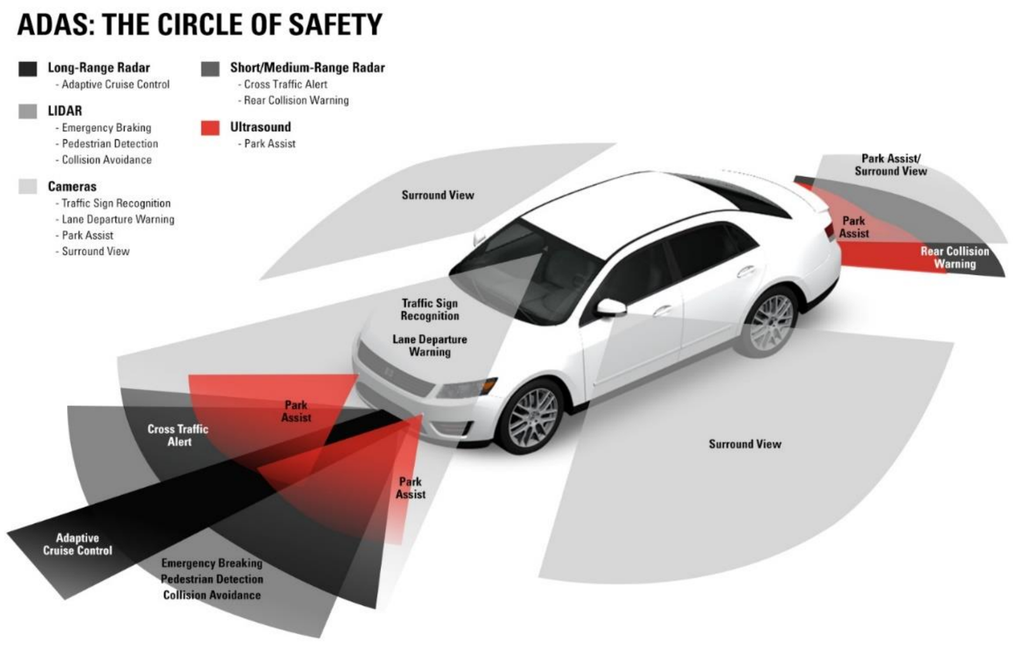


Figure 1.4 - Position of the ADAS sensors on a vehicle - Side View [7]

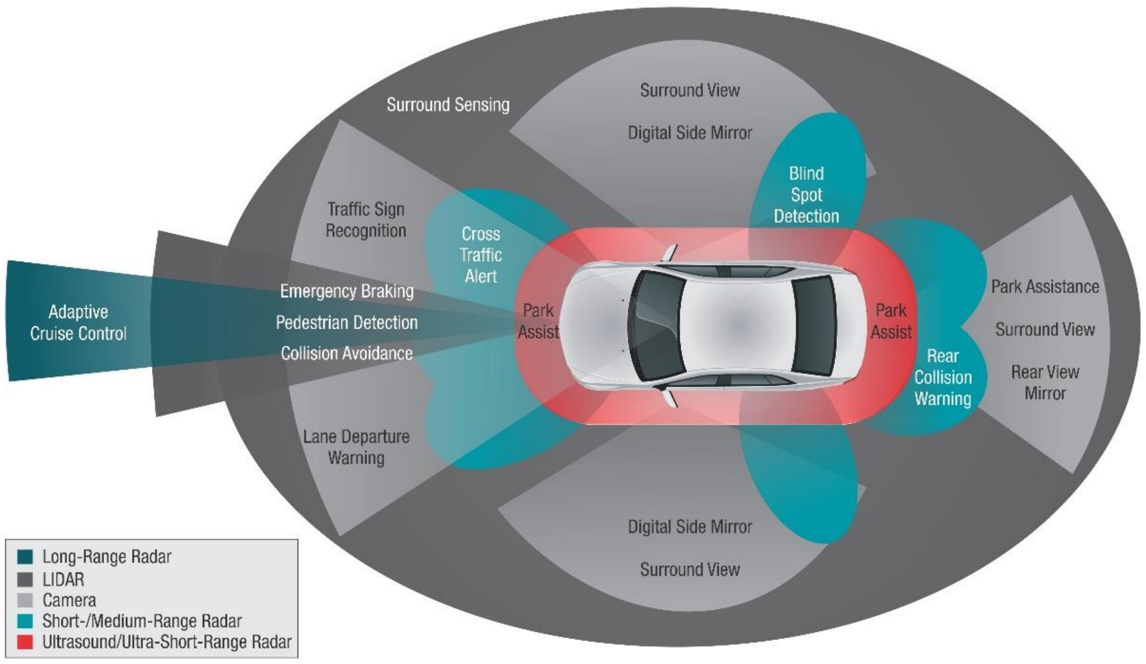


Figure 1.5 - Position of the ADAS sensors on a vehicle - Top View [8]

Data coming from a single sensor type is not enough to correctly understand the environment, a very advantageous option lies in the synchronization, aggregation and merging of various data coming from multiple sensors that will be used for centralized processing.
In fact, the sensors are able to share information with each other and, using intelligent algorithms, they are able to create an independent and autonomous system capable of storing all the data received from each of them. Therefore, the purpose of the fusion of the sensors is the union of more information coming from different types of sensors in order to provide a clear and 360° view of the surrounding environment, providing a high degree of reliability and safety in autonomous driving systems.

1.3 ADAS Development Process

In order to perform ADAS tests, it is necessary to recreate a driving scenario by generating a virtual environment in which to model each actor (pilot, sensors, climate, traffic, etc.)
This simulation, in addition to being safer than the tests performed in the real world, is able to analyze several driving parameters with great speed, including various aspects and situations that include different scenarios.
For this reason, the "V" cycle is used to manage the different production levels of the mechatronic and automotive systems. This diagram is based entirely on top-down (design) and bottom-up (validation) not necessarily in order; to obtain a product that works properly and above all meets the basic requirements imposed by users.

The model, implemented through a structured method, highlights how there is a close correlation between each phase of the software development life cycle and its testing phase. The V model is named after the two main phases present within it: Verification and Validation phases (Figure 1.6). Of great importance is the verification phase, i.e. the procedure aimed at ensuring that the output of each test phase meets the previously imposed requirements and specifications [9]. However, this is a time-consuming process, in order to detect even possible errors present in the system specifications.
In the validation phase, instead, each module is tested individually, verifying that they respect the needs of consumers by using data coming from real world.

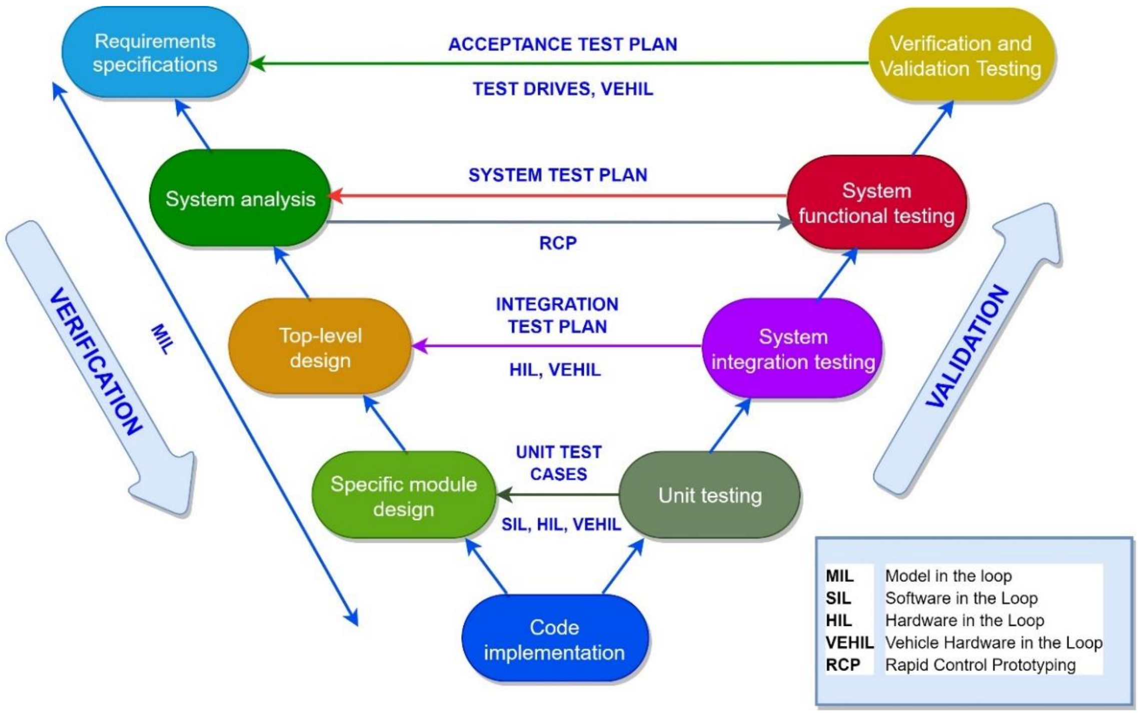


Figure 1.6 - V diagram model

Nowadays the constant and sudden increase in complexity both in terms of hardware and software has given the test phases and verifies an increasingly significant role in order to achieve not only a reduction in costs, but above all to test faster and faster in a repeatable and flexible way. To this end, various simulation methods have been introduced for the design and validation of the ADAS control system [10].
The initial set-up and verification of the ADAS controller subsystem is performed through a dynamic simulation of the behavior within a control loop MIL (model in the loop), simulating both the input of the vehicle dynamics, and sensors and actuators checking the controller algorithm. These verification techniques (testing) of electronic control units (ECU) are carried out on particular models and on specific HW.
After executing the MIL simulations, the software code of the controller is obtained in the final programming language and the SIL (Software-in-the Loop) tests are performed [11].

SIL is a method that allows to test some components of the ADAS software and evaluate the reactivity and performance of the source code based on the input sent in a simulated environment by connecting the relative algorithms that include the hardware of a vehicle.
The advantage deriving from the use of this particular technique lies in the possibility of performing tests, using a virtual environment, on a specific hardware avoiding the use of expensive physical simulation systems.
As a consequence of the introduction of these cutting-edge systems equipped with autonomous functionalities, there is a growing need to perform in succession at the SIL the HIL (hardware-in-the-loop) tests that integrate the controller software code to runs it on the controller hardware, used for validation and verification of advanced driver assistance systems.
Furthermore, it is necessary to take into account that a lot of energy is consumed in executing and developing some tools that can reduce real-world tests. That is the so-called virtual ADAS test, which uses a simulation environment to highlight any problems present within the simulation system. The simulation tests are carried out through HIL, which is the set of testing procedures of an ECU (electronic control unit). In that way, it is possible to reproduce all the possible scenarios involving the ADAS within a driving simulation system, which allows running faster, and less expensive tests, highlighting the possible flaws in the system with the hardware limitations.

The hardware-in-the-loop approach involves the development of a model that uses real-time simulation to control the vehicle hardware organized with other simulated or artificial elements (see Figure 1.7). This method is flexible and excellent for prototyping [12].

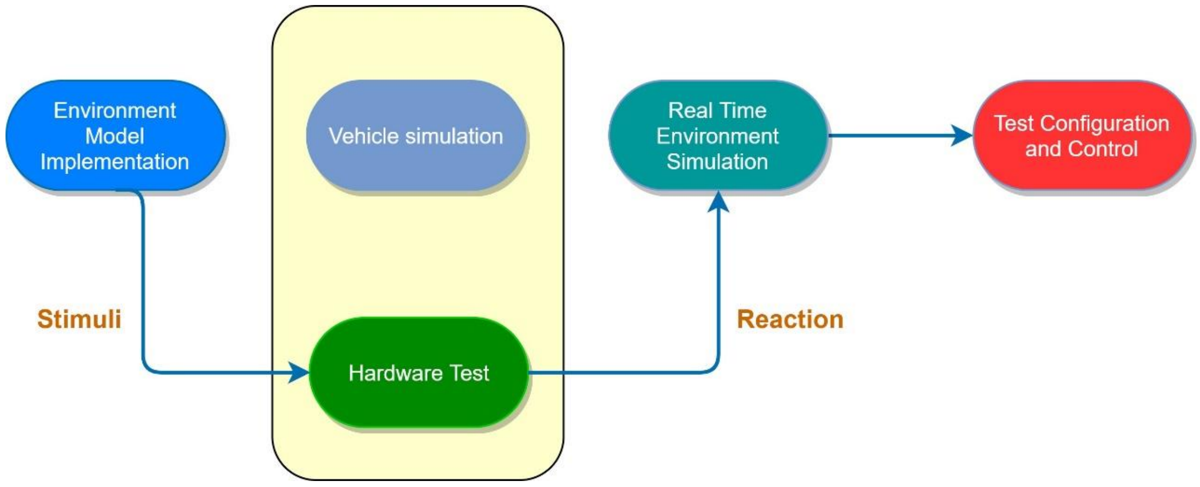

Environment Model Implementation
Vehicle simulation
Real Time Environment Simulation
Test Configuration and Control
Stimuli
Hardware Test
Reaction

Figure 1.7 - Typical HIL testing process

The DIL (Driver-in-the-loop) simulation can also be performed in addition to the previous method [12]. The advantages of using the DIL simulation laboratories are related to tests carried out on prototypes. For this reason, operators use a simulated vehicle in a virtual environment carefully, testing the driving of the virtual car that has controls very similar

to the real ones. The method that is able to unify both the use of virtual simulations and the tests in the real world is represented by the VEHIL (vehicle hardware in the loop).
VEHIL is a simulation with multiple agents that is used inside a well-equipped laboratory, in addition to the vehicle with autonomous guide, also other particular actors (usually artificial robots) able to interact with each other through an appropriate management of the situations.
In these terms, the self-driving vehicle equipped with ADAS in the situation in which another additional vehicle (mobile robot) is positioned in front of it, is put under observation and testing (Figure 1.8). The vehicle equipped with ADAS has all the sensors needed to collect all the information relating to the instantiated scenario, especially when there is movement between the two cars. This information is combined and integrated with each other to create a complete real-time environmental scenario that is usually projected on a screen positioned exactly in front of the vehicle under examination.
Finally, the output provided by the resulting system is shown through different HMIs (Human-Machine Interface) that allow to monitor and analyze the reality of the facts by shedding light on what is happening.

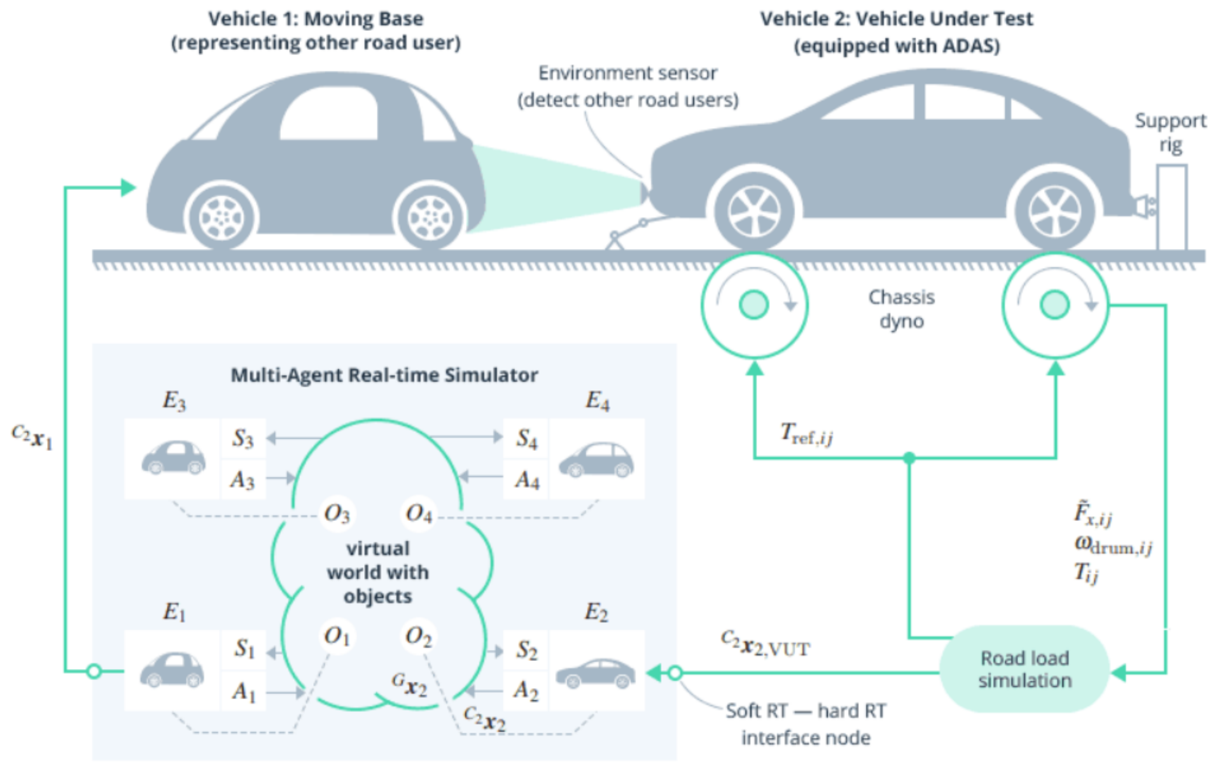


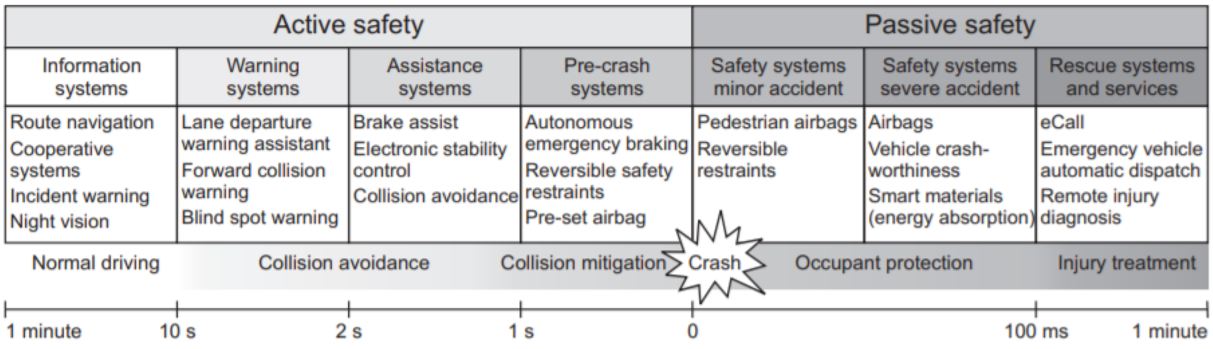
Figure 1.8 - VeHIL Scheme [13]

After having performed and passed a series of tests, the system is in the end verified and validated and the characterizing subsystems are regulated and improved and are subsequently inserted in a complete ADAS, on a virtual basis. In this final phase, it is

verified that ADAS meets all the basic requirements previously specified based on the logical and technical specifications provided.
In this thesis project, the requirements analysis and software development phases have been mainly addressed, subsequently executing a validation process that follows a series of tests aimed at measuring the robustness of the system, highlighting its limits and defects.

1.4 Passive vs Active Safety and Laws

Passive safety devices have the task of reducing (possibly to a minimum) the consequences caused by an accident. Their purpose is to protect not only the driver, but also the passengers of the vehicle by absorbing their kinetic energy.
This category includes both seat belts and airbags as well as the internal structure of the vehicle itself. The Euro-NCAP program tests the performance of these devices, through crash tests.
On the contrary, the active safety systems include the set of all those devices connected to a constantly activated control unit that try to prevent the occurrence of an accident, carrying out a preventive action controlling the dynamics of the vehicle. This category includes both traction and stability control and braking systems as well as ABS and acoustic sensors that promptly alert the driver in the event of imminent dangers such as in the event of collisions or lane abandonment. Figure 1.9 shows the differences between Active and Passive Safety.



Active safety				Passive safety		
Information systems	Warning systems	Assistance systems	Pre-crash systems	Safety systems minor accident	Safety systems severe accident	Rescue systems and services
Route navigation Cooperative systems Incident warning Night vision	Lane departure warning assistant Forward collision warning Blind spot warning	Brake assist Electronic stability control Collision avoidance	Autonomous emergency braking Reversible safety restraints Pre-set airbag	Pedestrian airbags Reversible restraints	Airbags Vehicle crash-worthiness Smart materials (energy absorption)	eCall Emergency vehicle automatic dispatch Remote injury diagnosis

Figure 1.9 - Active and Passive Safety systems cooperate in the time interval before, during, and after incident [9]

The European Parliament has approved and established a series of laws and rules that are aimed at the mandatory use of some of the assisted driving systems on all modern cars by the 2021 deadline, thus contributing to reducing the risk of road accidents and save the lives of hundreds of people.
In particular, at least 11 of the most important ADAS devices available including Automatic Emergency Braking (AEB a collision avoidance system), pedestrian detection, autonomous

cruise control (ACC), Lane Departure Awareness System & Maintenance and Intelligent Speed Adaptation (ISA) will become standard integrated devices.
Without any doubt, the introduction of this new type of vehicle will introduce new risk factors. An example is the moral machine [14] that is nothing more than a study carried out on the possible and different moral decisions taken by an autonomous vehicle. How to answer the question: in the event of an unavoidable accident, what should the self-driving car do once it is given the responsibility to decide who should live or die? (See figure 1.10).
Even though the Highway Code, both in America and in the rest of the world, has not yet set precise and clear rules on how and to whom to assign responsibilities in the event of accidents with self-driving cars. It will therefore be necessary to carry out meticulous operations aimed at minimizing and protecting the vehicle from sensor failures, design defects, hijacking, identity theft, traffic control system malfunctions.
In fact, even the EuroNCAP [15], an entity that provides an assessment (based on the number of stars) of passive safety of vehicles through a series of ad hoc tests (accidents that result in injury or death to individuals), has started to give more and more importance to active safety systems in its assessments.

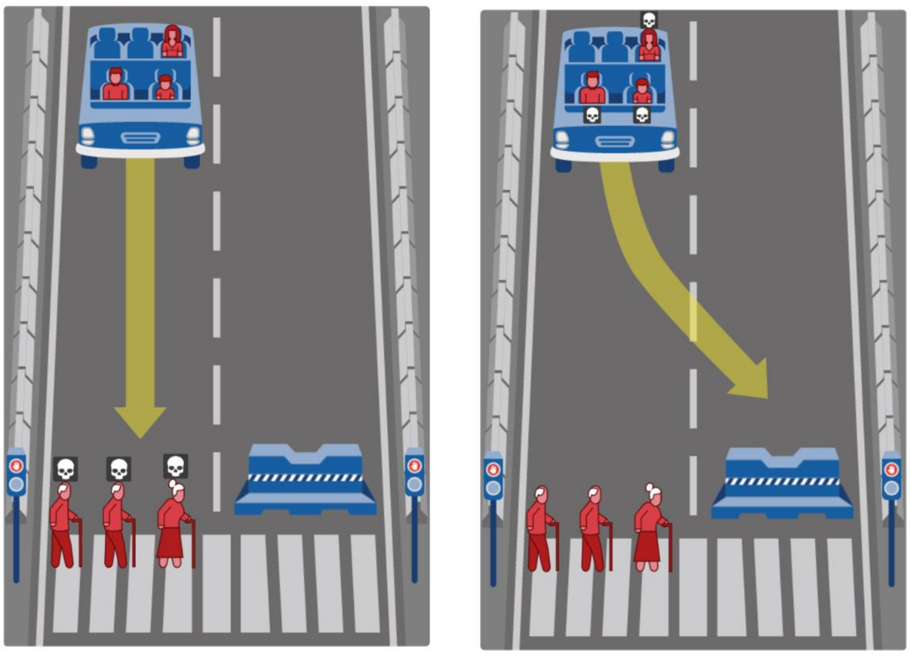


Figure 1.10 - On the left, the pedestrians would die, on the right instead the passengers would die[1]

[1] https://dblue.it/blog/the-moral-machine-una-morale-per-le-auto-a-guida-autonoma/

1.5 The GOLD (Generic Obstacle and Lane Detection) System

The GOLD system (Generic Obstacle and Lane Detection), uses a particular hardware and software structure focused on stereo vision, thoroughly developed with the aim of increasing road safety [16]. This type of algorithm allows both to detect generic obstacles without any constraint of symmetry or shape and to analyze the position of the lane "observing" the painted road signs, in various types of articulated circumstances.
The GOLD system is based on the PAPRICA system (which stands for Parallel Processor for Image Checking and Analysis), which uses a particular parallel architecture designed for special purposes at low cost, that operates at a frequency equal to 10 Hz (0.1 s).
The GOLD system analyzes both the detection of the lanes, which is based on the horizontal signs and the detection of obstacles, which location is notified by processing images present in front of the vehicle. All of this is done without any 3D reconstruction in order to make the process faster and more robust with respect to camera calibration and vehicle movements.
The GOLD system is executed on a specific hardware that uses a SIMD (Single Instruction Multiple Data) architecture in order to obtain not only real-time performance but also consume not too much energy [16]. Under the hypothesis of adopting the algorithm on a road with no slopes (i.e. flat) with present and visible lines, it is possible to obtain good performances, in the event in which these hypotheses are no longer valid, the algorithm's performance drops drastically.

In this thesis, both features (Lane Detection and Obstacle Detection) are developed using only the visual data acquired from a camera installed on the front of the vehicle (in a virtual environment) in which the image processing is involved.
For lane detection, some principles of operation of the GOLD system were used, such as the IPM (Inverse Perspective Mapping) approach, which eliminates the perspective effect.
Considering also the following assumptions: the position of the camera and its orientation are known, and a flat route can represent the type of road. Through this system, it is possible to overcome the limits imposed by the perspective effects produced by the images acquired by the camera mapping each pixel of the input image (2D) onto a new image (2D) that represents the top view.
After removing the effects of perspective, in the top-view image the lane lines should appear approximately parallel to each other. In this way, it is possible to process the image more simply by executing the algorithm for lane detection, which is based on identification of the pixel brightness of the lane lines, which is greater than the road surface.
Subsequently, the image is binarized (enhanced) through an adaptive threshold in order to detect the characteristics of the road and specifically discard all those pixels with a value of zero (completely black) and so consider only those that have a value other than zero.
The adaptive threshold is computed by the following expression:

$$t(x,y) = \begin{cases} 1 & if\ e(x,y) \geq \dfrac{m(x,y)}{k} \\ 0 & otherwise \end{cases} \tag{1.1}$$

Where e(x,y) represents the enhanced image, m(x,y) represent the maximum value computed in a given neighborhood and k is a constant. In this phase, the polar histogram is calculated and its peaks are isolated, which are used as a starting point to highlight the lane lines on the image.
For the obstacle detection, instead, a greater emphasis has been given to the preprocessing procedure without any three-dimensional (3-D) world reconstruction, which is able to detect not only objects with almost vertical edges, but also with different types of shapes.

2. CARLA Simulator

In order to establish the feasibility of the algorithms used with results that can be used even in the real world, it is necessary to test them in a particular simulated environment.
Different types of 3D simulators are available on the market also as open source projects; among these, for example there is CARLA Simulator (the simulator used in this thesis, Figure 2.1).
The use of a simulator is of fundamental importance as it is possible to highlight and analyze the decisions made by a self-driving car that uses a specific algorithm. The higher the accuracy of the simulation, the better the understanding of the scenario and future use.
The main objective of this thesis is to apply the algorithms to images and/or videos derived from the simulator and use the results of the virtual simulation to evaluate the performance of the algorithm, to improve it, making its application safer for autonomous vehicles. CARLA (wich stands for Car Learning to Act) is an open source simulator developed to study and test the mechanisms and algorithms concerning autonomous driving [17].

Figure 2.1 - CARLA Simulator [17]

It is based on the C++ language and on the Unreal Engine (version 4 is available from 2015) which is used to manage the different types of scenarios. CARLA also allows setting a third-person or first-person view with the vehicle itself as a reference point to have a general and complete view of the surrounding environment.

CARLA is able to provide open digital resources and infrastructures such as buildings and vehicles managed together with a series of open source protocols. CARLA is also able to simulate and manage realistic weather conditions (see Figure 2.2), advanced urban scenarios and NPCs (non-playable characters) in order to achieve greater randomness in different driving scenarios.

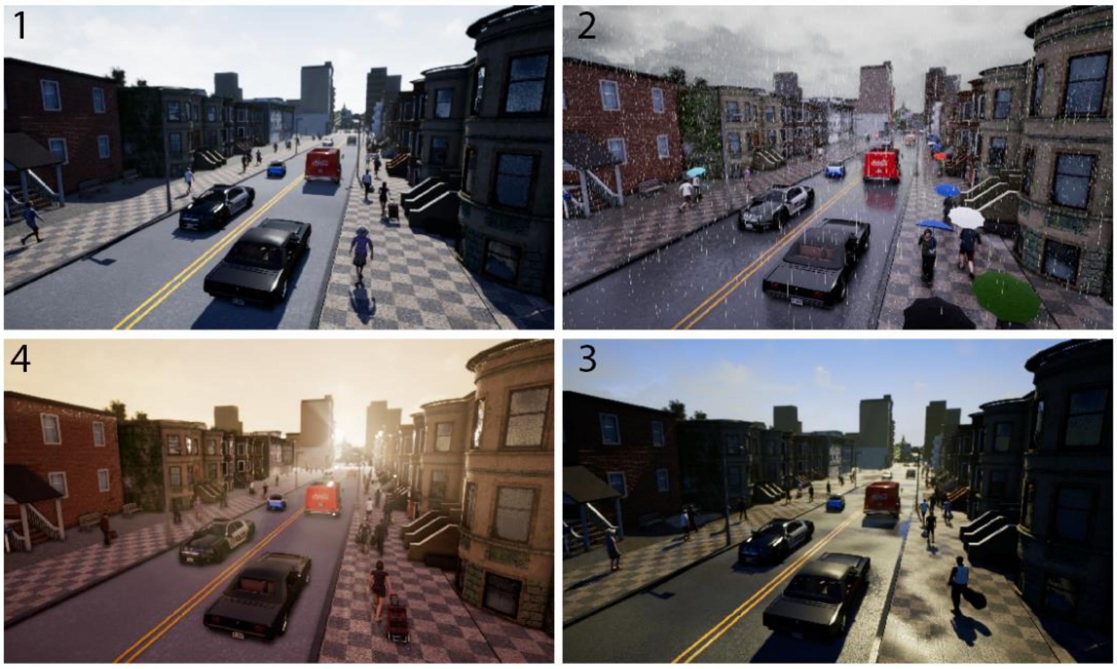


Figure 2.2 - View of a street in the Town 2, which shows 4 different weather conditions. From 1 to 4: sunny day, day with rain in the morning, day in the morning immediately after the rain and day at sunset [17]

CARLA has seven different "Towns" (up to now), i.e. different urban traffic scenarios in which the functions performed by the vehicles can be tested according to the tasks required. In the development of this thesis, in order to test the algorithm in the best possible scenario, the "Town04" has been used as it provides both an urban environment with intersections and with different types of objects and a toll-free motorway, whose lanes have different radii of curvature.

Furthermore, the platform is able to support not only the control of the map and the related actors on the map, but also camera and a suite of sensors such as lidar and a flexible python API which is able to perform a client-server communication (Figure 2.3).

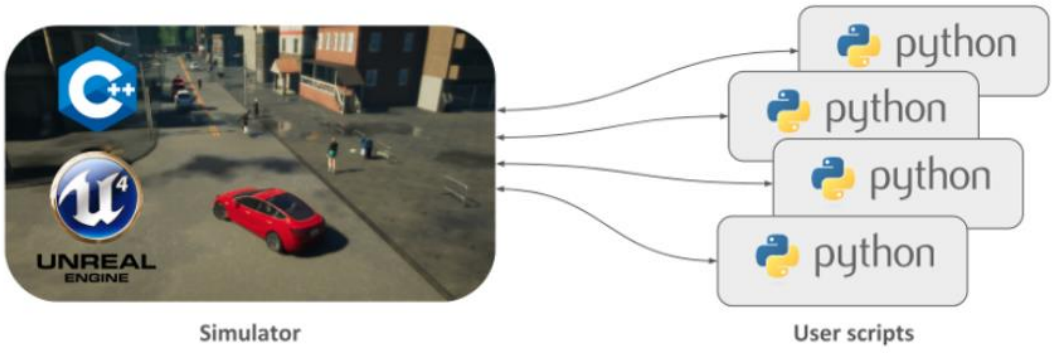


Figure 2.3 - CARLA Client Server Communication with CARLA modules representation [18]

CARLA simulator is made up of two modules that allow communication between client and server:

- CARLA Simulator (server)
- CARLA Python API modules (clients)

The simulator plays a role of fundamental importance as it performs most of the work: it controls the logic and artificial intelligence of pedestrians and autopilot vehicles, manages the physics of objects especially in the event of collisions and renders all the actors and sensors on the map. The Python CARLA API is a module that provides an interface for controlling the simulator with the possibility of collecting a series of different types of data. Hence, through the Python API, it is also possible to characterize the dynamics of any vehicle and control it within the simulation, insert different types of cameras and sensors and connect them to it in order to extrapolate and read the data produced by these sensors. In this thesis project, in order to collect the data in a precise and secure way, it has been decided to modify the example file made available by the CARLA development team (manual_control.py, present in the CARLA python API library[2]).
This script has been adapted to the need to configure the simulation by creating a suitable scenario to perform and test lane detection and object detection algorithms.
Through the communication between python API client and the CARLA simulator server it is therefore possible to capture single images or multiple images (which will later be put together to form a video) by a camera placed in the central front part of the ego-vehicle equipped with autopilot. These images will be the input for the lane detection and object detection algorithms (implemented in both python and C++ languages), which, through a series of methods and procedures, will produce the desired output. Figure 2.4 shows the block diagram of the simulation.

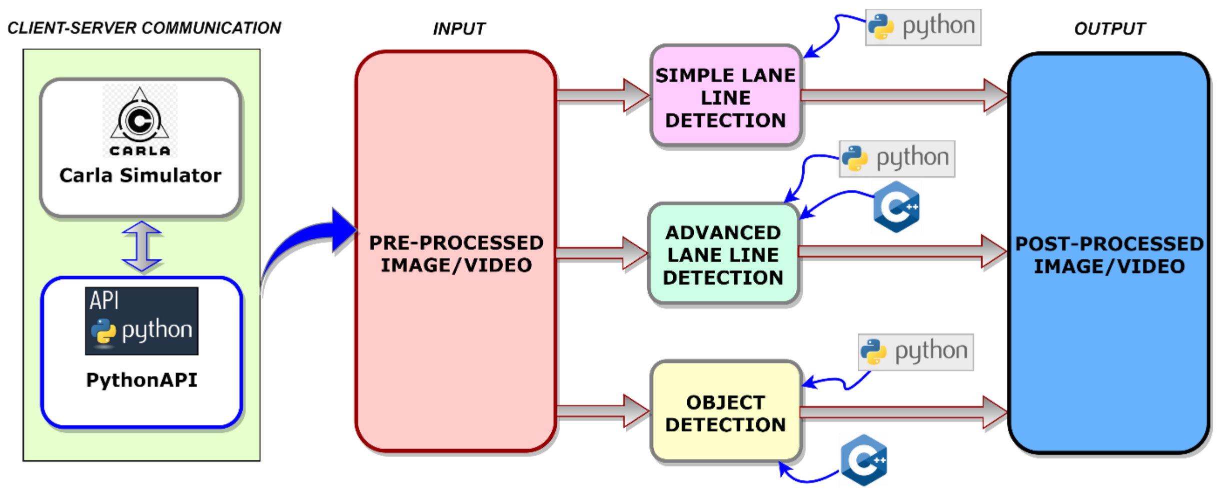

CLIENT-SERVER COMMUNICATION
CARLA
Carla Simulator
API
python
PythonAPI
INPUT
PRE-PROCESSED IMAGE/VIDEO
python
SIMPLE LANE LINE DETECTION
python
ADVANCED LANE LINE DETECTION
python
OBJECT DETECTION
OUTPUT
POST-PROCESSED IMAGE/VIDEO

Figure 2.4 - Block Diagram of the entire simulation test

[2] https://github.com/carla-simulator/carla/blob/master/PythonAPI/examples/manual_control.py

2.1 Configuring the Simulation

The first operation to perform before starting the tests, concerns the configuration of the simulation taking into account some important details in order to obtain the correct scenario. The steps required for configuring the simulation and the driving scenario are summarized in the horizontal diagram in Figure 2.5.
The basic idea of CARLA is that the simulator acts as a server, that renders the scene, and waits for a Client to connect, which is responsible for the interaction between the stand-alone agent and the server via socket [17]. The client is able to send commands to the server in order to verify and manage parameters such as the car, the pedestrians or the weather. The server can send data and images acquired from the sensors to the Client. As per default, communications between the client and the server take place on TCP ports 2000, 2001 and 2002.
Taking advantage of the Client class [18] it is possible to use the client side, which returns an object that can be used to obtain certain information from the simulation. After initializing pygame, it is possible to connect to the simulator creating a "Client" object, by providing both the IP address of the localhost and the port (default = 2000) on which to run the simulator. The server connection timeout can be set and later the town (Town04) can be loaded (**load_world()**).
After setting the python display window and HUD, in order to establish a connection between the client and the server simulator, the world is retrieved through the client object using the **get_world()** method. The World class, containing all the elements of the simulation, is thus instantiated. Moreover, in this part it is possible to adequately configure, according to the needs, the actors of the simulation, which include pedestrians, sensors, cars and even the view of the spectator.
After setting the initial simulation weather, which can be changed with the world object (**world.set_weather()**), actors can be generated. The class that contains all the information needed to generate an actor is represented by the ActorBlueprint class [18].
The actor can be considered as anything that plays a particular role in the simulation.
The carla.ActorBlueprint class contains the identifier (ID) of each actor that uniquely identifies it including also all the instances of the actors created with it (i.e. the list of all the tags of each actor present on the CARLA simulator). Generally, IDs are represented by three words separated by a dot.
Through the **get_blueprint_library()** method the entire list of available Blueprint can be retrieved. This blueprint library allows to find specific patterns by ID, filter them with wildcards or simply choose one at random. A specific blueprints can be easily selected using the **blueprint_library.find()** method specifying the ID of Blueprint. The ego-vehicle used in the simulation is Tesla Model 3 (with blue color).

It is possible to select any actor present in the blueprint library, in particular, the vehicles present in the simulation that share a certain attribute will be selected using the command: **world.get_actors().filter('vehicle.*. *')**, which returns the complete list of all actors of the ActorList type. This object is repeatable. It is important to highlight how carla.Actor, the base class for all actors, represents the actors present in the simulation; instead, the carla.ActorBlueprint class hosts all the information about the actor that can be produced, but not necessarily present within the simulation.
In addition, blueprints can be set and modified using the **set_attribute()** method specifying the ID and value or on the contrary, information can be extracted from an attribute of an actor using its ID with the **get_attribute()** or check that the actor has a certain attribute with **has_attribute()**. It is possible to establish the desired position of the vehicle using **get_transform()** method.
Among the available actors in the list, the spectator actor to change the position of the view on the simulator side can also be used. Its features are made visible through the **world.get_spectator()** method which allows to freely determine the position and orientation of the spectator in order to get a clear view of the scenario as a whole.
An actor can be generated within the simulation world using the **world.spawn_actor()** method, which requires the following arguments:

- The blueprint
- Information on the specified 3D pose (based on *carla.Transform* objects)
- The attachment options

The advantage of this method is that it can be used to produce any actor, including pedestrians or sensors connected to a vehicle. Of great importance is the notification by means of a tick message every time an actor is generated or modified, in such a way to give the world time to learn the change and avoid any kind of error in the subsequent act of code execution. Every type of change in the world within the simulator is only performed after the tick, although in the most recent version of CARLA (0.9.6) they automatically synchronize with the server. CARLA Simulator also allows to determine and modify the physical parameters of the wheels and the entire vehicle.
The classes that carry out this task are mainly two: the carla.WheelPhysicsControl class, which is used to control vehicle wheel attributes and the carla.VehiclePhysicsControl class, which is used to manage vehicle parameters (including wheels).
For example, within the carla.WheelPhysicsControl class, it is possible to modify the friction value to which a wheel is subjected by setting the **tire_friction()** variable. Instead, within the carla.VehiclePhysicsControl class, for example, it is possible to modify the maximum RPM of the vehicle's engine through the variable **max_rpm().**
Finally, to apply and adapt these parameters to the ego-vehicle within the simulation, the **apply_physics_control()** method is used, which applies physical control to the vehicle.

The carla.Waypoint class was also used, which allows to obtain the position of strategic points placed on the map.
Through this, it is possible to obtain some important information and compare it with the position of the ego-vehicle to obtain measurements on the lane, from which the width of the lane and the lateral error of the vehicle can be obtained.
Using the blueprint library, it is also possible to spawn pedestrians at random points on the map where the pedestrian can pass using a batch of commands, through which to create the controller that will automatically manage the path of pedestrians.
The sensors in CARLA represent a special type of actor, which, based on the type, are capable of measuring and transmitting different flow of data. In this sense, the Carla library is able to offer different types of sensors (see CARLA documentation [18]).
These can be easily connected to a vehicle, with which it is possible to save the images generated by the sensors in the simulation on disk. Each type of sensor has a **listen()** method that registers the callback function that will be called each time the sensor generates a new image. In this case, the class used to set up the sensors used in the simulation (Camera RGB and Camera Semantic Segmentation) with their characteristics, also establishing a method to store the sensor information (images or snapshot) from the screen on the disk was called CameraManager.
Immediately after collecting the images from Carla simulator, you can launch the python script **images_to_video.py,** passing as argument the folder where the images are, to put together a multitude of images to create a video, and the desired fps using OpenCV library.

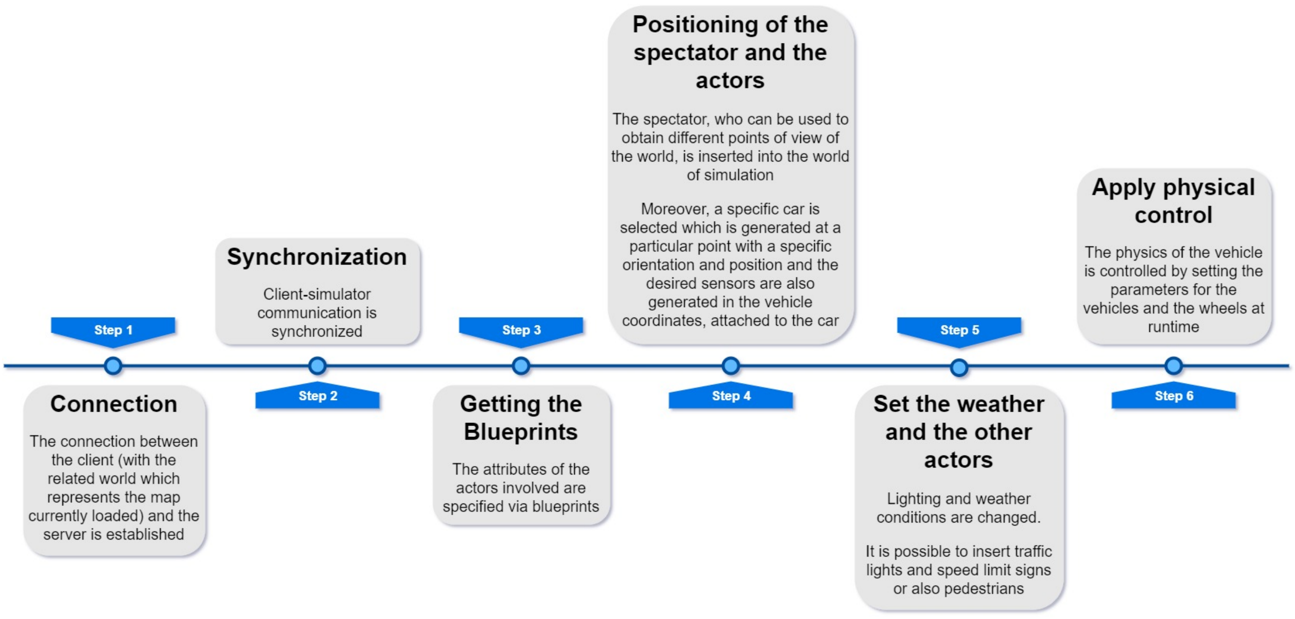
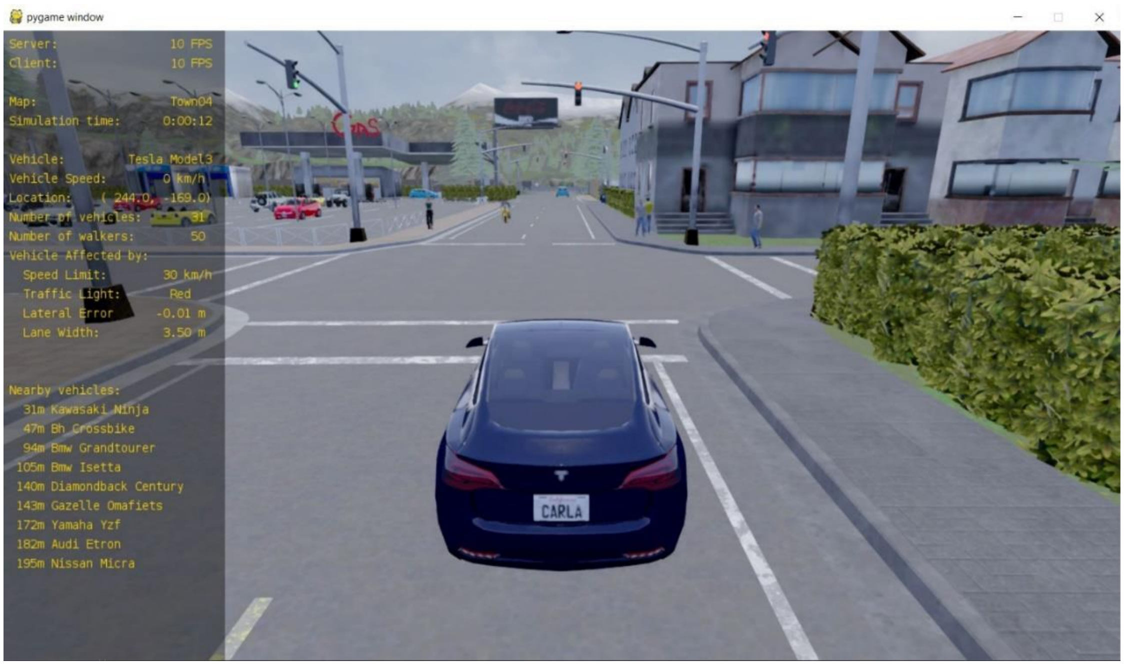

Step 1
Connection
The connection between the client (with the related world which represents the map currently loaded) and the server is established
Step 2
Synchronization
Client-simulator communication is synchronized
Step 3
Getting the Blueprints
The attributes of the actors involved are specified via blueprints
Step 4
Positioning of the spectator and the actors
The spectator, who can be used to obtain different points of view of the world, is inserted into the world of simulation
Moreover, a specific car is selected which is generated at a particular point with a specific orientation and position and the desired sensors are also generated in the vehicle coordinates, attached to the car
Step 5
Set the weather and the other actors
Lighting and weather conditions are changed.
It is possible to insert traffic lights and speed limit signs or also pedestrians
Step 6
Apply physical control
The physics of the vehicle is controlled by setting the parameters for the vehicles and the wheels at runtime

Figure 2.5 - Simulation Configuration Scheme

Furthermore, after correctly configuring the simulation it is possible to set a class (KeyboardControl) that is able to manage the different requests of the user through a command of the keyboard (controller). All the possible functions that can be activated by keyboard command are the following:

- P: Activate/Deactivate autopilot mode to the ego-vehicle
- TAB: Change the position of the sensor (from a third person view to first person view and vice versa)
- H: Activate/Deactivate helper on the screen
- ESC/Q: Quit the simulation
- F1: Activate/Deactivate the Head-up display (HUD) used to monitor simulation information
- C: Change the type of weather
- R: Activate/Deactivate recording images to disk (using camera sensor)
- N: Change the type of sensor
- I: Activate/Deactivate a snapshot of the scene of the simulation saving data on disk (using camera sensor)

Finally the HUD, FadingText and HelpText classes are used respectively to display some relevant information about the current simulation (such as ego-vehicle speed and location or simulation time) on the HUD screen located on the left part of the window, to show at the bottom of the screen the notification of events (keyboard command) and to display the helper text (Keyboard command function) on a surface in the center of the screen.
Once the simulation is finished, it is advisable to destroy any generated actor on the map by their ID. For more details, see **ADAS_scenario.py** in Appendix. Figure 2.6 shows the start screen of the simulation scenario with the active HUD (on the left of the screen).



```
Server:                10 FPS
Client:                10 FPS

Map:                   Town04
Simulation time:      0:00:12

Vehicle:          Tesla Model3
Vehicle Speed:          0 km/h
Location:       ( 244.0, -169.0)
Number of vehicles:         31
Number of walkers:          50
Vehicle Affected by:
  Speed Limit:         30 km/h
  Traffic Light:           Red
  Lateral Error        -0.01 m
  Lane Width:           3.50 m

Nearby vehicles:
  31m Kawasaki Ninja
  47m Bh Crossbike
  94m Bmw Grandtourer
 105m Bmw Isetta
 140m Diamondback Century
 143m Gazelle Omafiets
 172m Yamaha Yzf
 182m Audi Etron
 195m Nissan Micra
```

Figure 2.6 - Third-person view of the simulation in Carla Simulator (Town04)

2.2 Loading and Running the Simulation

After configuring the simulation, in order to be able to launch and successfully run the simulation, the following points are needed:

- Linux x64 operating system with Ubuntu 16.04 (or later) or Windows 7 64-bit operating system (or later) in order to run the CARLA loader.
- Python 3.7.x (x means any number) installed and added to the system environment variables in order to run the CARLA python client.
- The following python packages: pip package manager (to install modules and packages from the Python Package Index (PyPI)), pygame, pillow, moviepy, OpenCV, Numpy, Matplotlib, and PIL.
- Install Git, Make and CMake adding them to the environment path
- Allow the firewall access to ports 2000, 2001 and 2002 (TCP and UDP)
- Unreal Engine v4.22.x
- Visual Studio 2017 with Visual Studio x64 Native Tools Command Prompt
- CARLA is computationally expensive, recommended hardware requirements available on the Unreal Engine 4 page[3]:
    - Quad-core Intel or AMD processor, 2.5 GHz or faster
    - NVIDIA GeForce 470 GTX or AMD Radeon 6870 HD series card or higher
    - 8 GB RAM
    - About 20GB of hard drive space for the simulator setup
- CARLA 0.9.6 (last version) precompiled version (binary) or compiled version. The binary version is ready to be used by the user and contains all the components of Unreal Engine to load world, sensors, maps, actors, blueprints, etc.
  It has the advantage of taking up less space on the hard disk but on the other hand, it does not allow to use maps other than those already existing and does not allow to modify either the Python API or the C++ simulator source code.
  For this reason, the compiled version has been used, which not only allows the use of pre-existing maps, but also the creation of new maps using a simple and powerful software: RoadRunner

After following all the steps listed above, it is possible to start running the Carla server simulator (CarlaUE4 executable: **CarlaUE4.exe** for Windows or **./CarlaUE4.sh** for Linux):

[3] https://wiki.unrealengine.com/Recommended_Hardware

1. Run Carla simulator server:
   In the folder where CARLA was extracted a command window is opened and the Carla server simulator is started with the following command:

```
C:\carla-master\carla\Unreal\CarlaUE4\Content\Carla> CarlaUE4.exe -carla-
server –windowed -ResX=1280 -ResY=720 -quality-level=Medium –benchmark –
fps=10
```

   After executing this command, a new window will open with the width (-ResX) and the height (-ResY) of the rendered image at a resolution of 1280 x 720 pixels (common choise for rendering) with a view of the city initialized by default to the Town03.
   Setting a fixed time interval for the simulator equal to 10 frames per second (which means that every 0.1 seconds the image is updated) allows the simulation to be performed as quickly as possible, simulating the same time increment on each step (acceptable minimum = 10 fps, otherwise the simulation is no longer synchronized with the physical engine). The simulator's rendering quality level is set to the default value.
   The simulation scenario thus obtained represents the so-called "spectator" view, in which it is possible to move freely in any direction observing the world in its entirety but without having the opportunity to interact with it.
   At this point, the Carla simulator is running as a server, waiting for a client app to connect and interact with the newly created world from the server. In this way, the Carla simulator server will have to access the network bypassing the firewall.

2. Run Python API client:
   In the folder where the script file named "ADAS_scenario.py" is located, open the command prompt and run it without passing any argument as below:

```
C:\carla-master\carla\Unreal\CarlaUE4\Content\Carla\PythonAPI\examples>
python ADAS_scenario.py
```

   In this way the simulation runs using pygame 1.9.6 and by default:
   - Open a new window with a resolution of 1280x720 pixels ('--res');
   - Set the map to the "Town04" ('--map')
   - Set the simulation at 10 frames-per-second ('--fps');
   - Use a Gamma correction of the camera of default value of 2.2 ('--gamma');
   - Set no autopilot mode activated at the beginning of the simulation ('--autopilot');
   - Add 30 vehicles e 50 pedestrians to the world driving in "autopilot" mode, i.e. controlled by artificial intelligence (respectively '--number-of-vehicles' and '--number-of-walkers'). Cars will continue to drive randomly and pedestrians will continue to move to random positions until we stop the simulation, alternatively,

you can use the -n or -w flag to choose how many cars or pedestrians you want to generate within the simulation.

Performing all the steps mentioned above allows you to fully analyze the scenario just created and by pressing the F1 key, you can show on the screen different information contained in the simulation such as the server and client fps number, the number of vehicles and pedestrians present, the type and position of the ego vehicle, its speed, etc. This makes it possible to comprehensively analyze the scenario just created by pressing the F1 key, showing information contained within the simulation on screen as the fps number of the server and the client, the number of vehicles and pedestrians present, the type and position of the ego-vehicle, its speed, etc.

To observe the situation from a more general point of view it is necessary to switch to CARLAUE4 simulator server window and observe the scenario just created.
In the DOS terminal of the Python Client, it is possibile to see all the information about the world, and all the actors created in the simulation. If you want to voluntarily end the CARLA simulation session, you can do so either by pressing Q/ESC on the client side or by pressing Alt-F4 buttons in the Carla simulator server (or Ctrl-Alt-Delete buttons), ending up the CARLA UE4 application from the Task Manager.

The Town used in this thesis project is *"Town04"* which is quite large and includes both an urban and an extra-urban scenario: it contains a freeway, connecting ramps with different heights and a small town (Figure 2.5 and Figure 2.6).
Moreover, the simulation used in this thesis were carried out using a Notebook computer with the following features:

- Processor: Intel (R) Core^{TM} i7-7500U
- Base frequency of the processor: 2.70 GHz CPU
- RAM: 16 GB
- Graphics card: AMD Radeon^{TM} R7 M465

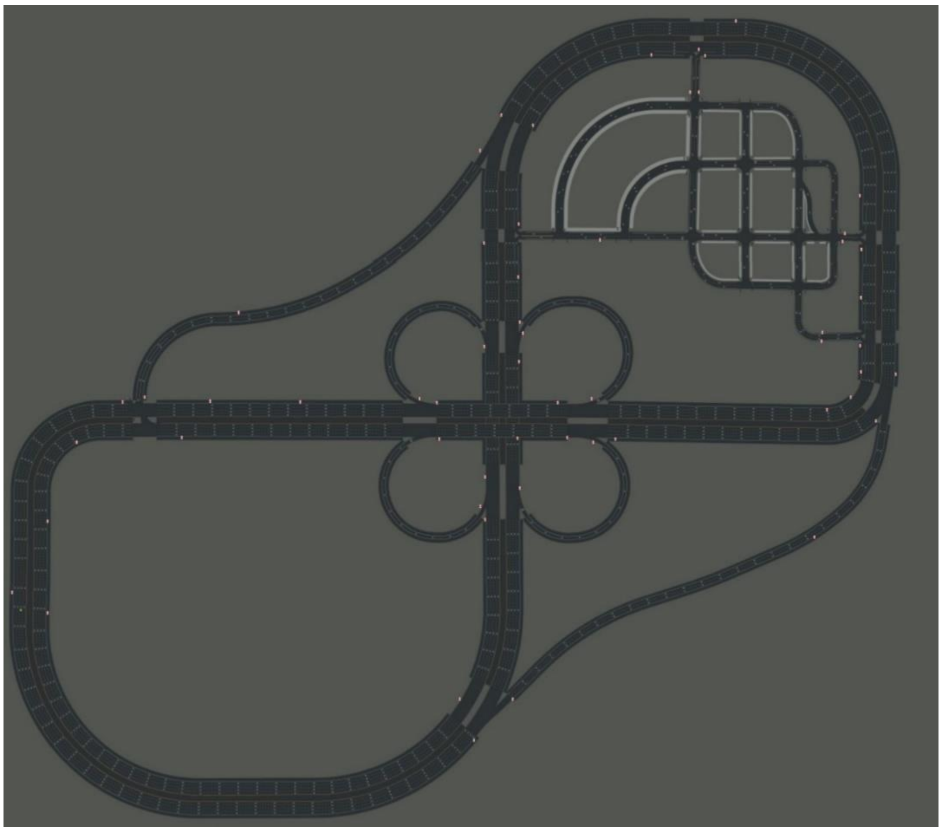


Figure 2.5 - 2D map visualization of Town04 in Carla Simulator

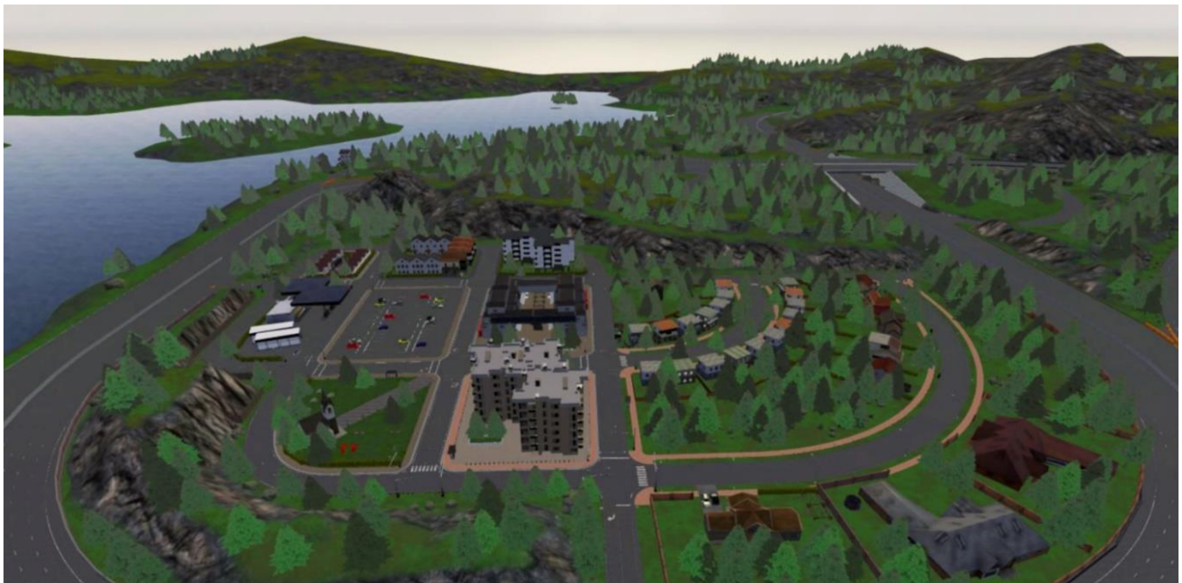


Figure 2.6 - 3D view of Town04 in Carla Simulator

3. Computer Vision and Sensor Fusion

The Computer vision represents the set of all those processes that exploit advanced technologies that have the main purpose of creating a concrete and reliable model of reality, thus reconstructing a sort of 360-degree human view. In particular, in the field of autonomous driving vehicles, computer vision represents the way in which machines visually perceive the world and respond in relation to it, collecting and putting together sensitive data through cameras and other sensors in order to obtain greater precision.
Then they use the input received to generate an output that will be exploited by artificial vision algorithms to be able to travel safely around the world. Computer vision is used in a myriad of applications, for example, the recognition of traffic light's state (red, green, yellow), the detection of lanes and the avoidance of obstacles on the roadway and so on. A fundamental aspect about computer vision is that all the techniques used are useful for any data that has spatial coherency, which is all those data that can predictably vary over space, such as the sound spatial information: the closer you get to the sound source, the stronger the sound will be. The key words of this complex system are mainly two: the diversity of the types of sensors used and the redundancy of the same sensors combined which provide an accurate final result as output.
Self-driving cars are located in a strategic position (Figure 3.2) and are able to take advantage, in addition to the cameras, also four types of sensors, which use sound waves, radio waves and lasers to be able to collect data on everything that surrounds the vehicle (see Figure 3.1). Although they perform different tasks, are all connected to the car computer's control unit (ECU/AI):

1. Radar Sensors, used to detect obstacles in the blind spots or furthest
2. LiDAR Sensors, used to correctly identify obstacles
3. Ultrasonic Sensors, used to detect nearby obstacles
4. GPS and Odometry sensors, used to identify the longitude, latitude and course speed of an object (vehicle)

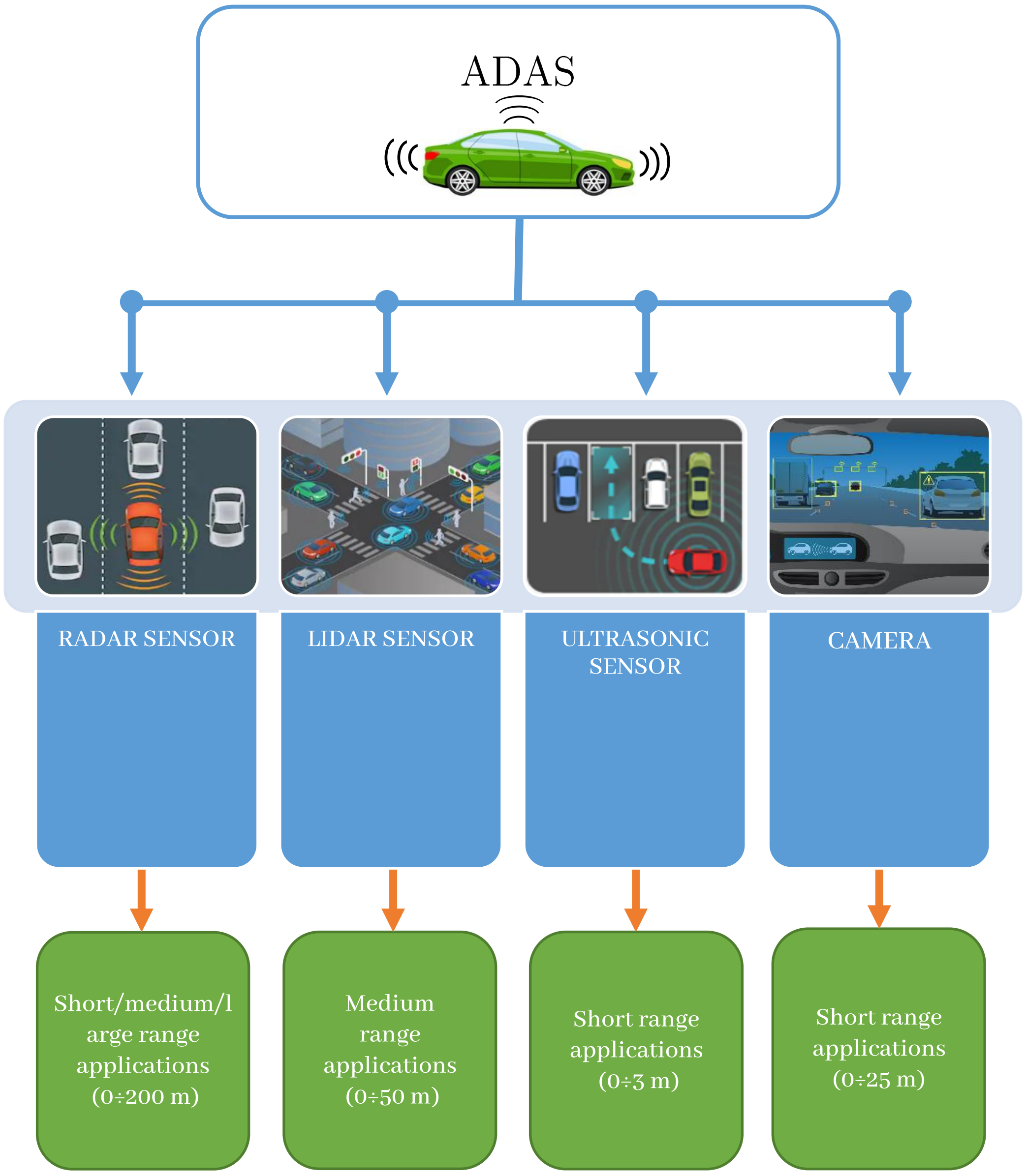

ADAS
RADAR SENSOR
LIDAR SENSOR
ULTRASONIC SENSOR
CAMERA
Short/medium/large range applications (0÷200 m)
Medium range applications (0÷50 m)
Short range applications (0÷3 m)
Short range applications (0÷25 m)

Figure 3.1 - Most common types of ADAS sensors and their ranges

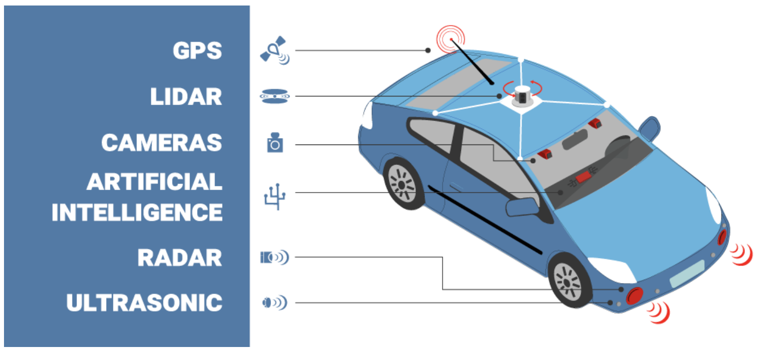


Figure 3.2 - Example of different types of sensors positioning in a self-driving car [19]

In general, RADARs are used for long-range surveys. They emit radio waves to detect objects close together in a range that goes from a few meters (from 1.5 to 30 meters) to a few hundred meters (up to 200 meters), using the reflected waves. This mechanism contributes greatly not only to detect objects in blind spots of a car (useful for parking assistance) but also to avoid collisions maintaining automatic cruise control.
They offer better results on objects that are generally not static but in motion where Doppler Effect occurs, i.e. a change in the radio wave frequency. The further an identified object moves away from the source, the more the frequency of radio waves will decrease.
Radar sensors can be used to provide support for the images provided by the cameras in all those scenarios where visibility is not the best, such as night driving or when there is fog, thus improving detection [20].

The LIDAR sensor is a type of sensor that works much like radar, but instead of sending radio waves, it uses light (a laser) to measure the distance between itself and the objects that reflect light. In particular, it emits a series of laser pulses and measuring the time taken by an object to reflect the light back to the sensor creating a point cloud in the observed directions. The more time the reflected light takes to return to the sensor, the more the object will be away from the sensor.
Through the rotation, the sensor is able to map 3D objects placed in the environment. It is possible to calculate the distance knowing the time (t) taken by the reflected light and the speed of light [21]:

$$Distance = \frac{c * t}{2} \tag{3.1}$$

Where c is the speed of the light ($3 \cdot 10^8 \ m/s$) and t is the Time of Flight, i.e. the time the pulse takes for return. This time is divided by two because the light must travel to the object and go back to the sensor. With this mechanism, the Lidar sensors are able to generate 3D images at 360° of the surrounding environment producing precise information on the depth of the objects immersed in it. They can also operate at distances greater than cameras, up to 50 meters. In this sense, LiDAR is a spatially coherent datum and in fact it can be used to generate a visual representation of the world (see Figure 3.3). They are mainly used for purposes such as obstacle detection and collision prevention. On the other hand, in addition to having a considerable cost, the Lidar are neither light nor bulky.
The LiDAR and RADAR sensors are part of the family of **active sensors**, which means they are able to perceive the surrounding environment based on the transmission of energy. On the other hand, the cameras are **passive sensors**, which means that they can only perceive the environment based on the energy already present (the photons).

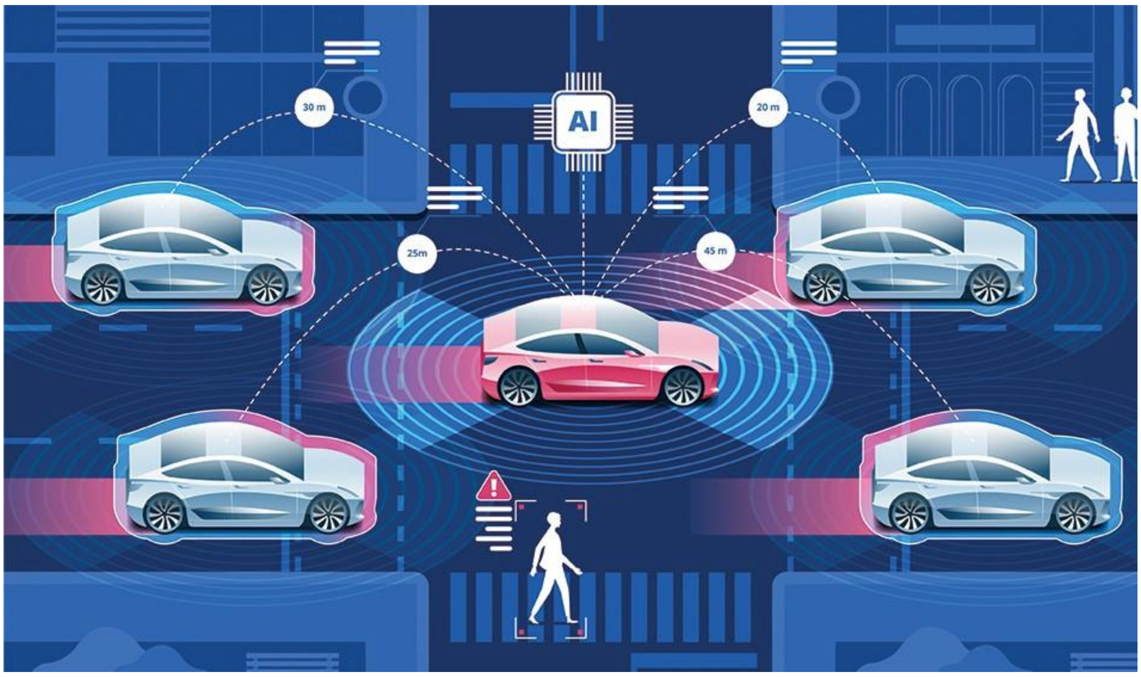


Figure 3.3 - Example of Lidar sensor data application for autonomous driving car [22]

The cameras, as opposed to Lidars, radars and ultrasonic sensors, are passive sensors, and represent the sensors that comes closest to sight for human beings. It can be compared to a third eye (in addition to the two of the driver) as it can accurately capture images in real time, even at high framerates, and process them using different types of algorithms to extract particular features and generate the desired output. In fact, these cameras frequently have a wide field of view and are usually placed on the center, on the left and right, front and rear of the vehicle to get a 360° view of the surrounding environment to monitor the situation.
The cameras are mainly used to perform various identification tasks, ranging from the detection of vehicles and pedestrians to obstacles present on the roadway or even the recognition of lanes and road signs (both vertical and horizontal) including traffic lights.
Despite providing a more than good resolution, the cameras have limitations. Regarding the images that are generated by monocular cameras (camera system that have only one objective), these have low processing requirements and provide poor information about the depth of the image. Thus, they are unsuitable for measuring the distance correctly. For self-driving cars [20], it is extremely necessary to know how far the near objects are positioned, to identify them in time and act accordingly.
The stereoscopic vision cameras instead, are characterized by two or, in most cases, more parallel lenses placed at the same distance, in such a way as to simulate the binocular vision of man and therefore create three-dimensional images from which to extract particular information. They can also be used to estimate the distance (depth) of an object accurately, as long as it is a short distance (maximum 25 meters).

In favorable climatic conditions, they are able to capture the details of both fixed and moving obstacles in the surrounding environment but they appear to be almost inefficient in detecting details in conditions of poor visibility (and therefore when the image becomes too dark, like at night or too disturbed and faded when there is fog).

Ultrasonic sensors work on the principle of reflected sound waves (similar to echolocation used by bats) and they are usually used to estimate the position of stationary vehicles (not moving objects), since the sensors can determine how far the objects are and notify the ECU of the vehicle when it approaches (range of about 3 meters).
The Ultrasonic sensors exploit sound waves with a frequency that are imperceptible to the human ear and are particularly suitable for performing functions involving low speed, short and medium range, as for example while parking (Figure 3.4).

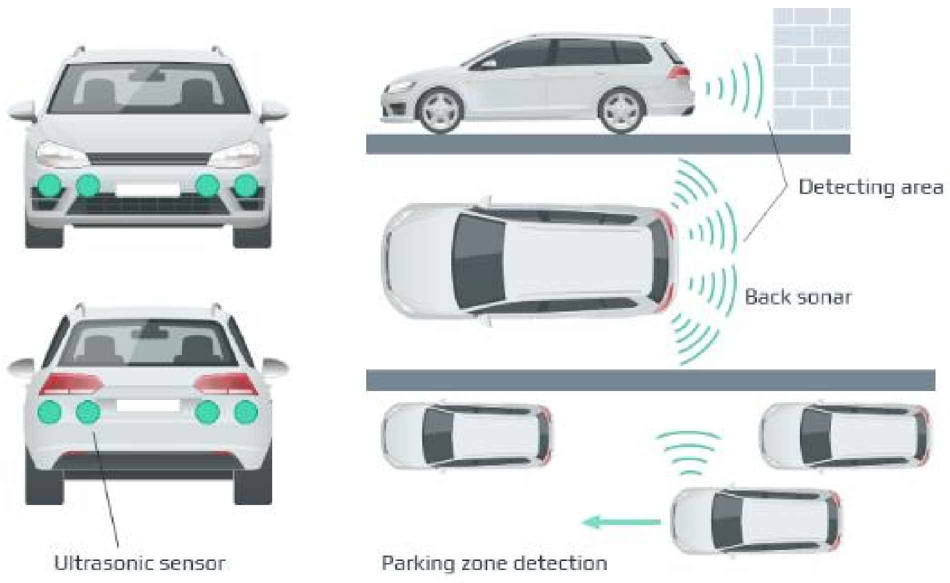


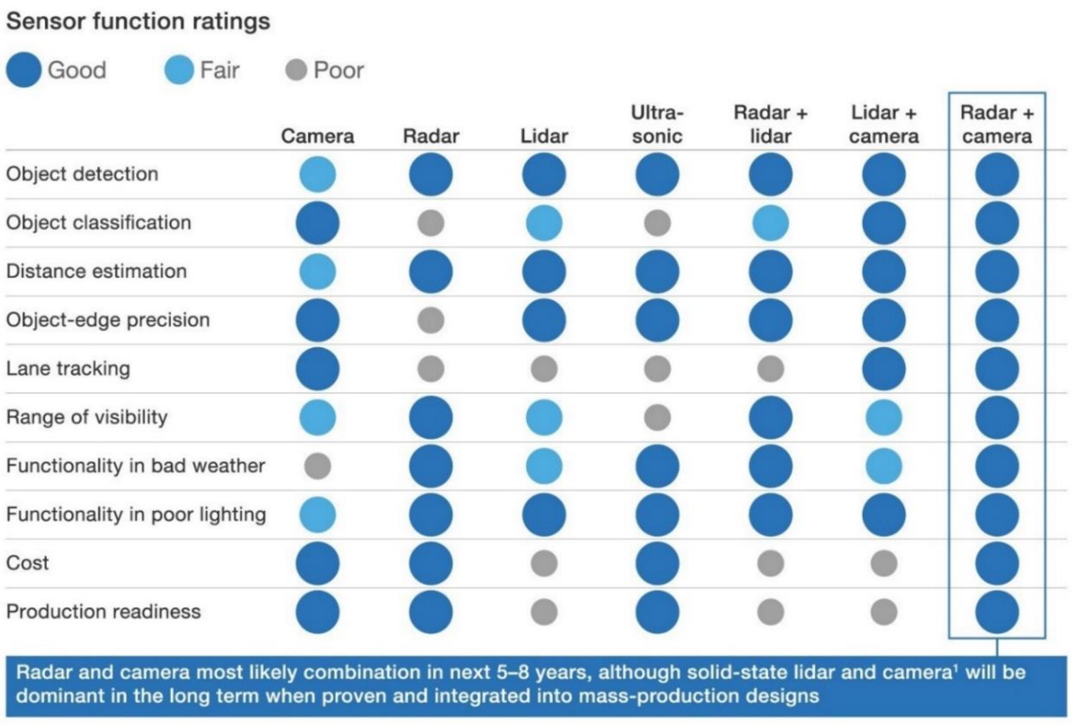
Figure 3.4 - Example of ultrasonic sensor positioning [13]

Nowadays, GPS is the most widely used positioning system in the world (but not the only one), which performs its duty very effectively to detect the position of an object (the receiver). It is based on the principle of trilateration: it exploits the radio signals generated by four or more satellites (with their relative travel time) and the distance of the Earth from each of them [21].
In fact, it is widely used in car navigation algorithms to plan the best route to follow on a digital map. For the self-driving cars, GPS is of fundamental importance as it serves to accurately determine the position of the vehicle also based on the position of the lane lines, to correctly follow them. In order to obtain the position in (x, y, z coordinates), the receiver needs to solve the following equations [21]:

$$\begin{aligned}
c\left(t_{T,1} - t_{R,1} + t_s\right) &= \sqrt{(x_1 - x)^2 + (y_1 - y)^2 + (z_1 - z)^2} \\
c\left(t_{T,2} - t_{R,2} + t_s\right) &= \sqrt{(x_2 - x)^2 + (y_2 - y)^2 + (z_2 - z)^2} \\
c\left(t_{T,3} - t_{R,3} + t_s\right) &= \sqrt{(x_3 - x)^2 + (y_3 - y)^2 + (z_3 - z)^2} \\
c\left(t_{T,4} - t_{R,4} + t_s\right) &= \sqrt{(x_4 - x)^2 + (y_4 - y)^2 + (z_4 - z)^2}
\end{aligned} \tag{3.2}$$

Where c represent the speed of the light ($3 \cdot 10^8\ m/s$), $t_{T,i}$ is the transmission time of the receiver's position from the satellite i, $t_{R,i}$ represent the time to receive position information from the satellite i and finally t_s is the skew time between transmitter clock and receiver clock. GPS and IMU (inertial measurement unit) data are usually combined together to generate more accurate odometry data. In fact, the combination of the two also makes it possible to estimate the speed of the vehicle accurately by studying and analyzing the displacement of its wheels.
Although the GPS also uses several satellites, it is not free from measurement errors.
The accuracy of the GPS depends strongly on several factors: the position of the satellites, the climatic conditions that influence the travel time of the light and multipath effect that affects the actual distance traveled by the signal because the signal can bounce between buildings before reach the receiver, which may receive a distorted distance.

Each of these sensors has several advantages and disadvantages, which allow a limited use for each specific application. For a concise and complete comparison of the most common types of sensors for self-driving cars, see Figure 3.5.


Sensor function ratings

● Good ◍ Fair ○ Poor

	Camera	Radar	Lidar	Ultra-sonic	Radar + lidar	Lidar + camera	Radar + camera
Object detection	◍	●	●	●	●	●	●
Object classification	●	○	◍	○	◍	●	●
Distance estimation	◍	●	●	●	●	●	●
Object-edge precision	●	○	●	●	●	●	●
Lane tracking	●	○	○	○	○	●	●
Range of visibility	◍	●	◍	○	●	◍	●
Functionality in bad weather	○	●	◍	●	●	◍	●
Functionality in poor lighting	◍	●	●	●	●	●	●
Cost	●	●	○	●	○	○	●
Production readiness	●	●	○	●	○	○	●

Radar and camera most likely combination in next 5–8 years, although solid-state lidar and camera[1] will be dominant in the long term when proven and integrated into mass-production designs

Figure 3.5 - Comparison between different types of sensors [23]

Taken individually, no single sensor can fulfill all the safety requirements required for autonomous driving, above all in any type of weather condition. For example, the cameras provide high quality images, but their performance is considerably reduced in low light conditions or in the case of very bad weather conditions. On the other hand, radar and lidar sensors work well even in adverse weather conditions, but their resolution is not suitable to classify adequately any objects.
Therefore, to overcome these limits it is necessary to use data coming from multiple sensors and merge them intelligently to ensure reliable driver assistance in such a way as to create overlapping data models (redundancy) so that the processed data are as precise as possible, thus obtaining a detailed 3D perception of the surrounding environment (Figure 3.6).
The purpose of the sensor fusion is to rely on the different advantages offered by each sensor as if it were a single entity.

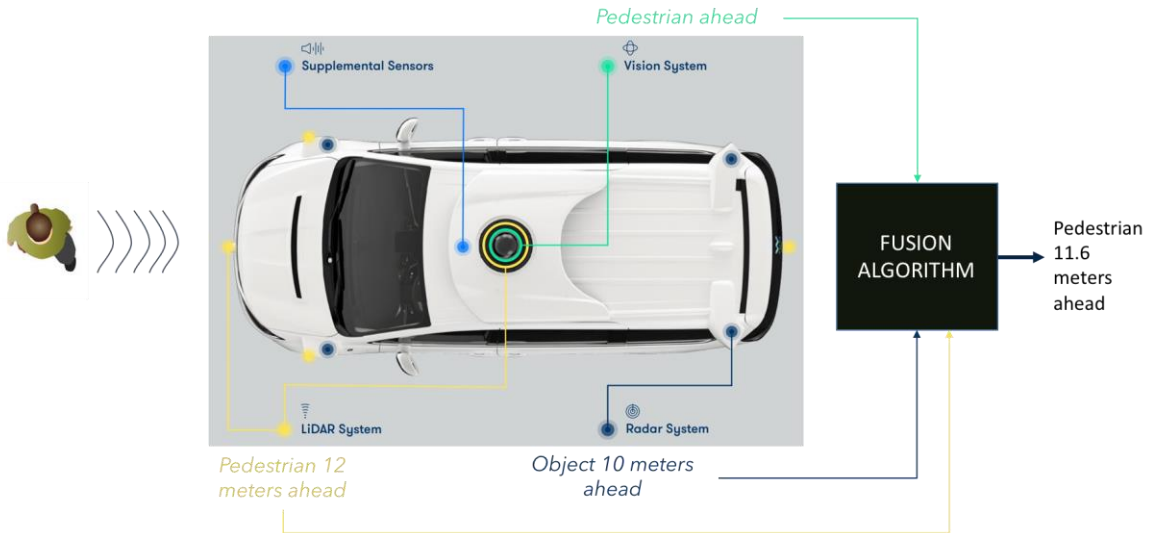


Figure 3.6 - Example of Sensor Fusion [24]

There are different types of algorithms that can be used to perform this data fusion, but surely, the most popular one that is also able to handle the imprecision of sensor data (noise) is certainly the Kalman filter. The power of the Kalman filter can also be exploited to predict certain types of future actions that the car will undertake based on information derived from the sensors. The Kalman filter is used to estimate the state of a system x based on its previous values, obtaining the measurements of that state using the new data coming from the sensors. The forecast is updated. Then the filter, based on the probability, updates the sensor measurements and repeats the infinite cycle.

3.1 Kalman Filter Prediction Equation

The state of the system x is characterized by a vector containing a position p and a velocity v.

$$x = \begin{bmatrix} p \\ v \end{bmatrix} \tag{3.3}$$

From the mathematical point of view, the prediction consists in estimating the state of x' (the predicted state) and the uncertainty of P' (the predicted covariance matrix) at the time t based on the previous states of x and P at the instant t-1 [25].

$$\begin{gathered} x' = Fx + u \\ P' = FPF^T + Q \end{gathered} \tag{3.4}$$

Where F is the state transition matrix from t-1 to t needed to convert the matrix from one form to another, u, instead, inserted in order to make the prediction more correct, represents instead the noise in the process and finally Q is the process covariance matrix including the uncertainty that depends on the ΔT interval.

3.2 Kalman Filter Update Equation

The update consists of taking into consideration the z measurement from a sensor, calculating the difference between the predicted value and the measured value and thus adequately correcting the forecast (x and P) based on the Kalman Gain. The difference between the actual measured value and the prediction, that represent the error, can be expressed as:

$$y = z - Hx' \tag{3.5}$$

Where z is the actual measurement value coming from the sensor, H represent the state transition matrix (H performs the same function as F in the Prediction Step). The estimated total system error S (prediction error plus measurement error) and the Kalman gain K can be computed as follow:

$$\begin{gathered} S = HP'H^T + R \\ K = P'H^T S^{-1} \end{gathered} \tag{3.6}$$

Where R represents the covariance matrix associated to sensor noise in the measurement.

The Kalman Gain K is perhaps the most important parameter of the whole algorithm, since it represents the weight to be attributed both to the expected value and to the measured one. Based on its output value K, which can vary from zero to one, it is possible to determine whether the actual, real value is in accordance with the measured value or the expected value. The closer the value of K is to zero, the more the predicted value is approximate to the real one (high measurement error), while if the value of K is close to one, the measured value will be approximate to the real value (high prediction error).

In the final step, based on the value of the Kalman Gain K, the values of x and P are updated and the endless cycle starts again (see Figure 3.7):

$$\begin{gathered} x = x' + Ky \\ P = (I - KH)P' \end{gathered} \tag{3.7}$$

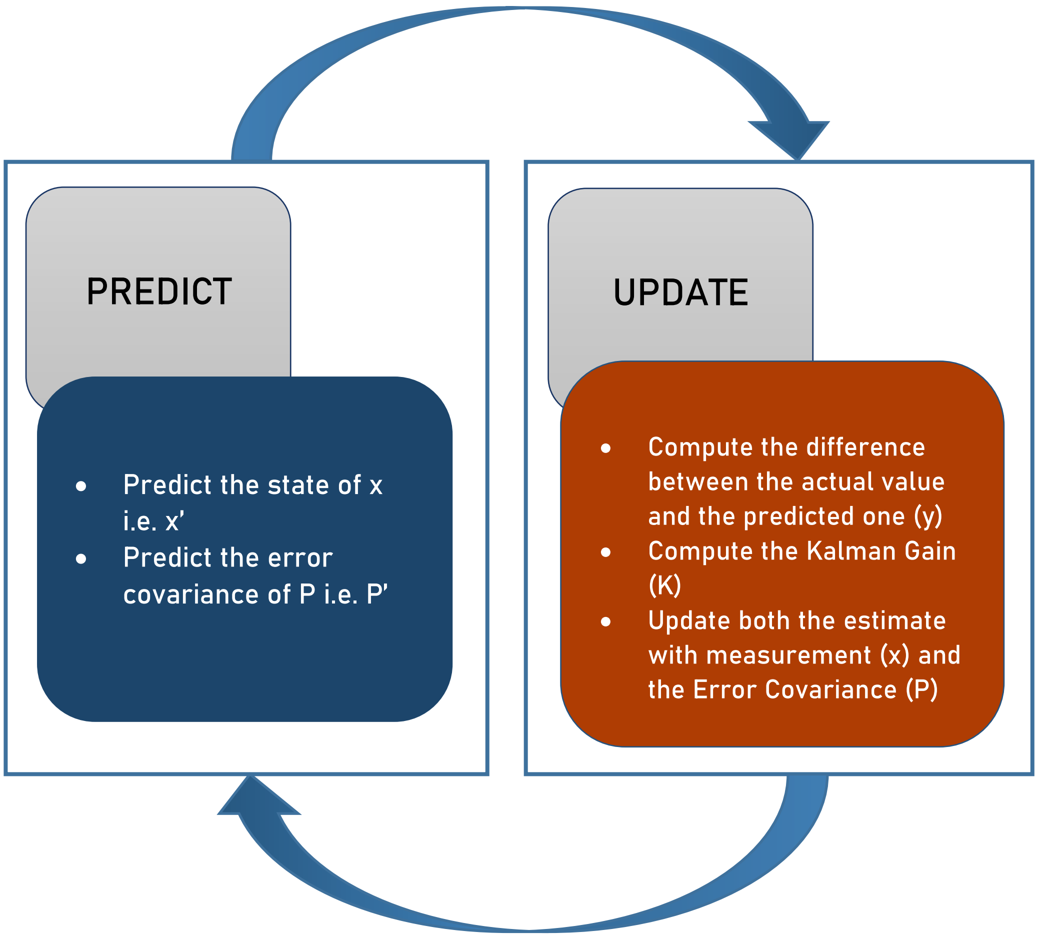

PREDICT
Predict the state of x i.e. x'
Predict the error covariance of P i.e. P'
UPDATE
Compute the difference between the actual value and the predicted one (y)
Compute the Kalman Gain (K)
Update both the estimate with measurement (x) and the Error Covariance (P)

Figure 3.7 - Kalman Filter Flowchart

3.3 Features of a camera

The images can be seen, from all point of view, as a 2D representation and projection of the 3D reality from which it is possible, through various operations, to extract mainly information of a geometric nature, such as points or lines, and of photometric type, as color or intensity values of the pixels. In general, most color images can be represented by the combination of only three colors: Red, Green and Blue, also known as RGB images to which you can add an additional channel called alpha, which contains extra information.
In this thesis project the RGB color model was widely used for the representation of an image (with High Definition: 1280x720 pixels of resolution and 16:9 aspect ratio), which provides graphic information in the form of a pixel matrix, one for each channel (red-green-blue).
In particular, color images can be defined as 3D cubes in which there are values with width, height and depth. The depth of an image represents the number of color channels and in RGB images, this parameter is equal to 3, which also represents the number of stacked 2D color layers, one for each color. Each digital image is formed by pixels and each pixel has a determined value. A pixel is black when its value is as low as possible, i.e. zero.
When the pixel value increases, its intensity also increases to a maximum value that depends on the number of bits allocated for each pixel, which in the case of 8 bits (1 byte) is 255.
An RGB image with 8 bits allocated for each pixel has therefore 256 possible levels for each channel, i.e. over 16 million color values. Color information is very useful in classification and recognition processes, such as lane identification or vehicle or pedestrian recognition, as it is possible to choose which type of algorithm to adopt based on the purpose. For this reason, images based on the RGB model are widely used for image processing applications, such as vision-based ADAS.
In fact, many computer vision applications exploit images and raw input data re-elaborating them through specific algorithms and multiple techniques intended for other types of analysis and recognition where, based on the results obtained, it is possible to generate an output action. Figure 3.8 summarizes all these steps in a horizontal chain scheme.

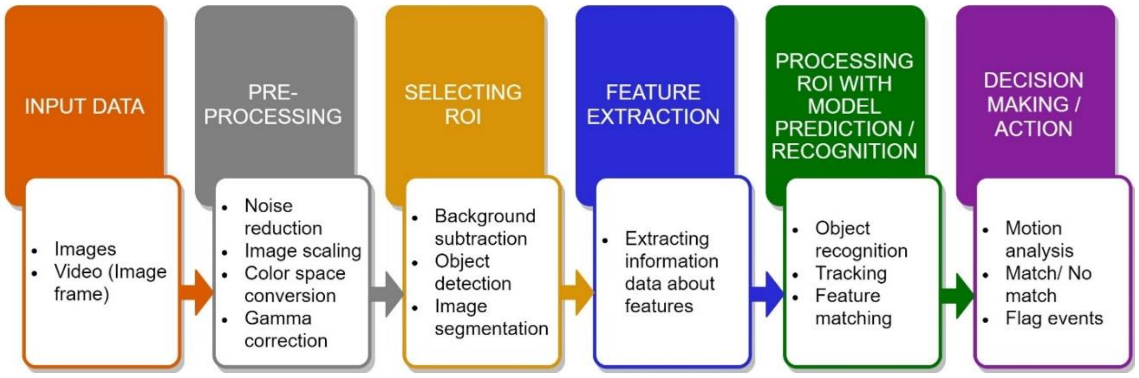


Figure 3.8 - Computer Vision Processing Pipeline

In order to be effective, the camera data need to be interpreted by expensive and advanced artificial vision algorithms or deep learning and need to be executed on powerful hardware so that they provide useful information to the driver in real time, constituting an excellent system for ADAS and autonomous driving.
This approach, however, can be time consuming for calculation and may not work properly on devices that do not have adequate hardware. Hence, if there is a lack of high power hardware, the system cannot be used for real-time applications because the execution of the entire algorithm and then the rendering of a single image frame may take a few seconds to be processed entering into conflict with the initial security requirements.

Another aspect that should not be underestimated and which could affect the performance of computer vision algorithms is certainly the distortion of the image, which occurs when a camera takes 3D objects in the real world and transforms them into a 2D image.
This transformation is not perfect and causes the distortion of the shapes and sizes of 3D objects within the image. Therefore, the first step in analyzing images from a camera is to eliminate this distortion so that we can get correct and useful information from the original image.

The simplest model of a camera is the pinhole camera model [26]. When a camera takes an image, it observes the world in a similar way as our eyes do, in this particular case, through a small pinhole, the camera focuses the light that is reflected by 3D objects in the world forming an image 2D on the back of the camera. The image that will be formed will be upside down and inverted because the light rays that hit an object from above continue on that path angled through the hole and will end up in the inferior part of the formed image. From the mathematical point of view, this transformation from 3D object points, P (X, Y, Z) to 2D image points, p (x, y) is performed by a transformative matrix called the camera matrix (see Figure 3.9).

$$x = f\frac{X}{Z}; \quad y = f\frac{Y}{Z} \tag{3.8}$$

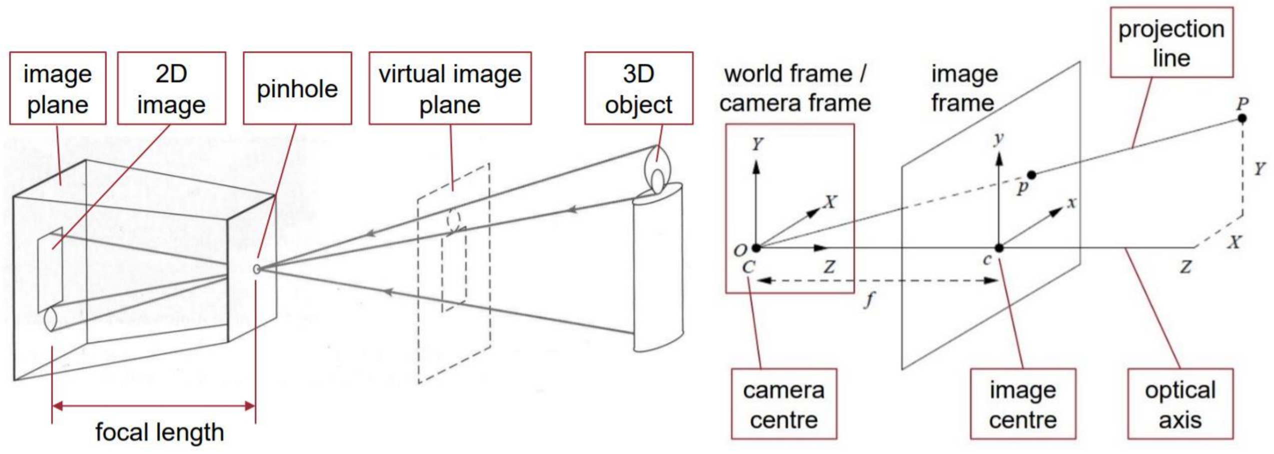


Figure 3.9 - Pinhole Camera Model and Reference Frame [26]

However, real cameras do not use small holes like the pinhole model; instead, they use lenses to focus more light beams at a time to allow the rapid formation of images. Nevertheless, even lenses can introduce distortions especially at the edges of images (as for the fish-eye lenses), so that the lines or objects appear, more or less, curved of what they really are.
The radial distortion of the image represent the most common one, due to the optics caused by imperfect lenses that produce more evident deviations. CARLA Simulator uses virtual 2D cameras and for this reason the images are distortion-free and therefore both distortions and the calibration procedure will not be taken into consideration.

In autonomous driving cars, the vision-based ADAS use the images of the cameras to perform mainly tasks related to machine vision to be able to extrapolate relevant information. These tasks mainly include detecting lane lines on the road and the road surface and identifying objects on it [27].
In particular, the thesis project carried out is dedicated to the fulfillment of the above-mentioned tasks, tested in CARLA simulator, all carried out through the use of a single camera positioned externally to the vehicle in the central front part, slightly inclined towards the surface of the road.

4. Lane Lines Detection

In the field of self-driving cars, identifying lane lines plays a key role in driving assistance. The identification of the lanes is not only useful to understand which is the radius of curvature to which the vehicle is subjected, but also to calculate the position of the vehicle with respect to the lane lines and possibly inform the driver if the vehicle is leaving the lane. All this in compliance with the rules of the road that also include maintaining speed and safety distance. This is a very critical task to perform for an autonomous vehicle, since the GOLD system has been developed and applied on a particular hardware architecture that cannot be executed on the hardware of a normal PC. For this purpose 2 different algorithms have been implemented which are based on the GOLD guidelines. The 2 types of algorithms proposed here, based on computer vision, use different procedures which, with their advantages and disadvantages, reach different conclusions. The first use a simple method, fast but very inaccurate and sensitive to noise, the second proposed later, is more advanced and much more precise but less fast.

The first algorithm uses the Hough transformation to detect lane lines (found by the Canny Edge Detection method) and the least squares for a polynomial first-order adaptation. In order to work properly, it is necessary that both the right and left lane lines are present, both possibly visible, continuous and not broken. The algorithm is also not efficient in detecting curved lines.

The second algorithm, instead, uses various color and shading transformations, which, together with the Sobel operator for edge detection, combine to determine a binary image with threshold. Subsequently, after accurately selecting the ROI (Region of Interest), the perspective transformation is applied to correct the binary image (the so-called “bird's-eye view”) and then the pixels of the lane are detected and adapted through a polynomial adaptation of order 2 to find the lane limit. Finally, both the offset of the vehicle with respect to the center of the lane and the curvature to which the vehicle is subjected, are calculated. This algorithm is able to work even in the presence of non-continuous, curved and barely visible lines. The Kalman filter can possibly be used to improve the final result for lane estimation. This operation was not implemented in this work because it is not part of the objectives of the thesis, but the project presented here can be considered as the starting point for future developments.
A clarification is necessary: the lane lines may vary depending on the type of road and the regulations of the country where the road is located. In some cases, these lines may be yellow or white, with dotted or continuous lines separated at a fixed distance. For this reason the second algorithm proposed allows, through a series of computer vision techniques, to detect the lane lines robustly and adequately.

Generally, the artificial vision algorithms that provide for the recognition and detection of the lanes are characterized by different steps [28] [29]:

1. **Extraction of lane line characteristics:** Extracting the characteristics of lane lines makes it easier to identify the pixels that belong to lane lines and eliminate pixels that do not represent lane lines. The most common approach is based on the idea that there is a high contrast between the road surface and the lane lines. Other types of approaches seek to detect lane line signs from a different perspective, searching for low-high-low intensity patterns along the image rows. The most common is the box filter (mainly known as a top-hat filter) other types of algorithms concern the use of the steerable filter, which is able to obtain the response of the gradient directions.

2. **Perspective Transformation:** In an image, perspective is that particular phenomenon which occurs when an object appears smaller the farther away it is from a viewpoint (like a camera), and parallel lines appear to converge to a point. Thus, the lane looks smaller the farther the observer moves away from the camera. Mathematically, we can characterize perspective by saying that, in real world coordinates x, y, z the greater the magnitude of an objects z coordinate, or distance from the camera, the smaller it appear in a 2D image.
   A perspective transform uses this information to transform an image. In particular, it allows mapping certain points of a given image to different points of the image through a new perspective. The most important and useful perspective transformation is the "bird's eye" one, which provides the necessary means to visualize an image as if we saw it from above, enlarging the most distant objects and eliminating perspective distortion due to the original image. Thus, a perspective transformation allows changing perspective to view the same scene (image) from different viewpoints and angles to be able to process more easily.

3. **Model fitting:** Represents the process of extracting a high-level representation of the lane based on the correct derived results of lane line detection. Depending on the model used, the vehicle can be derived from the fitted model. There are different types of road patterns used mainly: parametrics that include straight lines, parabolas and hyperboles, and the semi-parametric ones that refer to the spline group. There is no single model that can cover all types of road shapes, it is necessary to take into consideration the one that best suits the situation.
   Regarding the fitting methods, the most commonly used are the Hough transformation for straight line fitting and the second polynomial fitting for the curved lines, while RANSAC for other types of model fitting.

4. **Time integration:** The temporal integration includes the use of the previous information in order to guide the search in the current image. It is used to imposes both fluency and continuity between consecutive images. There are several approach, for example using the Kalman filter (on flat road surfaces) or particle filter which is more reliable especially under sudden changes in between consecutive images induced by vehicle vibrations or non-flat road surfaces. By taking advantage of this step, it is possible to improve the accuracy of the vehicle's location and prevent incorrect detection errors.

5. **Lane-level localization:** This last step consists in estimating the lateral position of the vehicle and the orientation in movement based on the lane model. In order to perform these calculations and conversions from a 2D image to the real 3D world, it is necessary to use the image depth, which can commonly be derived from the viewing angle of the camera or its angle of inclination (such as for IPM), bearing in mind to assume a constant height of the video camera and a flat road surface. In conclusion, the accuracy of the location is sensitive to the angle estimation noise and it is highly dependent on the angle of inclination.

5. Simple Lane Line Detection

In this part, the first algorithm to be able to detect lane lines from images of a camera is presented. This type of algorithm uses methods and functions presented in the web course for autonomous driving cars [30] suitably modified, in conjunction with Python and the OpenCV [31](Open Source Computer Vision) library. OpenCV is a library for computer vision and machine learning software, which has powerful means to analyze and process images in their entirety. Hence, this project provides the tools necessary to be able to detect lane lines in an image, also providing a visual representation of these.
Furthermore, since the images can be considered as dense matrix data, the numpy library will also be used, which allow to perform image transformation and rendering.

The operations to be performed in order to process the data and extract the characteristics of the lanes by identifying them are the following:

1. Capture the image as input
2. Resize and digitize scanned image
3. Grayscale conversion
4. Noise Reduction (using Gaussian Blur) and image attenuation
5. Detect edges (using Canny method)
6. Select the region to consider (applying that on image mask, using bitwise operation)
7. Hough Transform (in order to identify the lane lines)
8. Optimization (Line Filtering & Averaging) producing the output image

The first two steps can be easily met using the methods provided by the OpenCV library: **cv2.imread()** and **cv2.resize()** which are able to read and resize the image respectively. After that, the image is converted to grayscale (using **cv2.cvtColor()** method), which is a black and white image composed of a single channel, which has no color information and where the pixel value (ranging from 0 to 255) represents its brightness: the higher the pixel value, the brighter it will be and the other way around. This operation is performed so that the processing is not only faster, but also less complex from the computational point of view compared to a 3-channel RGB image.

Whenever an image is captured by a camera, it may be damaged by lines or dots and disturbed by random noise. The noise represents a fundamental component of the image since if it is high can alter the pixels inside it, influencing the identification of the lane edges when they are subsequently analyzed producing a non-optimal detection.
One of the image processing techniques to remove these disturbances and improve the image is the blur operation, performed using the Gaussian filter (**cv2.GaussianBlur()**), which is

a low pass filter capable of convolving the image and eliminate the high frequency components of the image, specifying the width and height of the Gaussian kernel which must be positive and odd. It is possible to decrease or increase the height and width parameters of the kernel to display the intensity variation of the blur. The kernel, used for the convolutions of the image, is represented as a matrix of numbers; moreover, kernels of different dimensions that have models of different numbers produce different results in convolution. It is essential to keep in mind that large kernels involve a high processing time. The distribution of the kernel value is calculated using the 2-D Gaussian function.
In 2-D, an isotropic Gaussian (circularly symmetrical) can be written in the following form:

$$G_{2D}(x,y) = \frac{1}{2\pi\sigma^2} e^{\left(-\frac{x^2+y^2}{2\sigma^2}\right)} \tag{5.1}$$

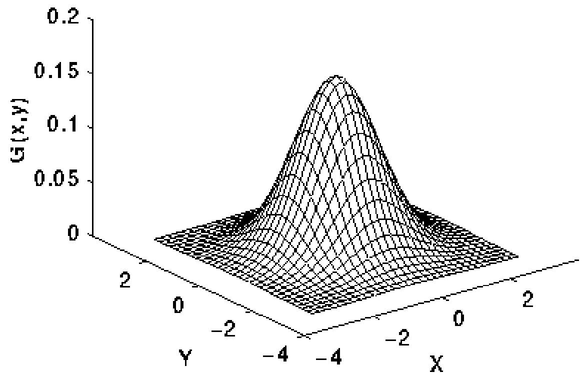


Figure 5.1 - A graphical representation of 2D Gaussian distribution with mean at origin and σ = 1 [32]

Where x, y represent the coordinates of the kernel in which the origin (i.e. x = 0 and y = 0) is placed at the center and **σ** is the standard deviation of the Gaussian distribution (hence, **σ²** represent the variance). Larger kernels need larger **σ** to get a precise Gaussian smoothing result. It is therefore necessary to determine the right kernel size to specify the neighborhood of each pixel. If this is too large, most probably some of the characteristics of the image may be lost and the image will appear very blurred, but if it is too small, the image noise reduction operation will not be successful. Accordingly, it is necessary to discretize continuous Gaussian functions in order to store them as discrete pixels. The relative representation of the convolution kernel approximating a Gaussian using a size of 5 is:

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \tag{5.2}$$

The Gaussian kernel is symmetric, and all its values are symmetric. The **σ** of the Gaussian kernel 5 x 5 written above is one. In this way, a Gaussian 5x5 kernel filter (optimal size in most cases) is used to attenuate the image to reduce noise sensitivity.

Edge detection aims to recognize the boundaries of an object in an image in order to identify it in space. This operation can be performed on an image by searching for in areas where there are rapid changes in brightness and color.
The 2D image, seen as a matrix of pixels with rows and columns with different intensities, can be represented as a mathematical function of x, which crosses the width (columns) of the image and of y, along the height of the image (rows), so it is possible to perform mathematical operations on it just like any other function. In the image converted to grayscale, taking its derivative, which represents only a measure of the change of this function with respect to x and y at the same time, the so called gradient is calculated where the brightness of each pixel corresponds to the strength (magnitude) of the gradient at that point. A small derivative means small change between two adjacent pixels and the other way around, a large derivative means big change.
In this way, it is possible to measure how fast pixel values are changing at each point in an image and in which direction they are changing most rapidly. It is therefore possible to obtain the edges of an image by following the pixels that characterize the strongest gradient using the Canny algorithm (**cv2.Canny()**).
By identifying the edges, in fact, it is easier to recognize objects based on their shape.
In the canny algorithm the low threshold and the high threshold are also expressed, which determine the force with which the edges must be detected. In particular, the detection of the edges of an image and consequently the threshold values using Canny method, are strongly dependent on its content, that is, on the level of brightness, darkness, contrast. The last 3 functions are enclosed into **do_canny()** function.

Specifically, Canny method, using the image filtered by noise, acts in this way [31]:

- **Computing the Gradient of the Image:**
  The image is filtered with a Sobel kernel in order to obtain both the first derivative in the horizontal direction (G_x) and in the vertical direction (G_y):

$$\nabla f(x,y) = \begin{bmatrix} \dfrac{\partial f(x,y)}{\partial x} \\ \dfrac{\partial f(x,y)}{\partial y} \end{bmatrix} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} \tag{5.3}$$

  Therefore at each point of the image both the M(x,y) - magnitude of the gradient (edge strength) and α(x,y) - direction of the gradient is calculated:

$$M(x,y) = |\nabla f(x,y)| = \sqrt{\left(G_x^2 + G_y^2\right)} \tag{5.4}$$

$$\alpha(x,y) = \arctan\left(\frac{G_y}{G_x}\right) \tag{5.5}$$

Considering the fact that the gradient direction is always orthogonal to the edges.

- **Edge localization applying Non-maximum Suppression:**
  After calculating the magnitude and direction of the gradient, the image is scanned as a whole in order to eliminate any unwanted pixels that may not represent the edge of the object considered.
  Therefore, on each pixel (x,y) of the image, the following control is performed: if its magnitude is a local maximum in its neighborhood in the direction of the gradient it is accepted, otherwise it is discarded (inseting a pixel value of 0, i.e. black)

- **Hysteresis Thresholding:**
  The algorithm will first detect the pixels of the strong edge (i.e. where the magnitude gradient is high) above the high threshold and will reject the pixels below the low threshold (weak edge) maintaining the low to high threshold ratio to 1:2 or 1:3.
  Subsequently, pixels with values between high-threshold and low-threshold will be included as long as they are connected to strong edges (i.e. those above the high threshold). Finally, output edges are represented as a binary image with white pixels (i.e. with value of 255, corresponding to logical 1) that identify the edges of an object and blacks if they do not identify them (i.e. with value of zero, corresponding to logical 0). Therefore, it is obvious how important it is to adequately select the low threshold and high threshold values to obtain a correct detection that depends on the situation and on the type of image with which one works.

After that, a trapezoidal quadrilateral mask is then created to segment the lane area of interest and discard the non-relevant areas in order to increase the effectiveness of the algorithm. To do this, a completely black mask is generated, that is constituted by a matrix with all its pixel values equal to zero (**np.zeros_like()**), of identical size to those of the original image and it is filled with the quadrilateral, which contains the vertices that constitute the region of interest (**cv2.fillPoly()**), with maximum intensity equal to 255 (i.e. completely white).

To correctly and precisely select the region of interest from the rest of the image, a callback function has been exploited that is able to print on screen the coordinates of x and y at each left mouse click on the image (**cv2.setMouseCallback()** in **find_ROI_coordinates.py** python script).

Note that in the display of an image the origin of the axes is placed in the top left corner and the coordinates of y-axis increases from top to bottom, while those of the x axis increases from left to right.
Subsequently, in order to display the characteristics of the individual area of interest, a bitwise AND operation is performed (**cv2.bitwise_and()**) between the image previously obtained with the Canny method and the mask just created, so obtaining the binary cropped image. The white pixels of both the edges of the canny image and the white quadrilateral within the black mask both correspond to one.
The result of the binary AND is only 1 when both are 1, so the aforementioned AND operation produces an image where the lane lines of edge detection are only visible within the region of interest (ROI). These functions are summarized into **do_segment()** function.

At this point, the Hough transformation, which is able to identify straight lane lines in the previously filtered image, can be used. To find the lane lines, it is necessary to first adopt a model and then fit that model to the assortment of dots/segments in the cutout box of the edge-detected image (with canny method).
Considering the fact that the image is a function of x and y, a line can be represented uniquely by the following equation in the **cartesian coordinate system** in explicit form:

$$y = mx + b \tag{5.6}$$

In this case, the model includes two parameters: **m** and **b**, where m represent the angular coefficient of the line and q is the altitude at the origin of the line. In image space, a line is plotted as x versus y, but in parameter space, also called Hough space, the same line can be represented as m versus b instead.
The Hough transformation is simply a conversion from the image space to the Hough space. Therefore, a single point in the m-b position in the Hough space will represent the characterization of a line in the image space (Figure 5.2).

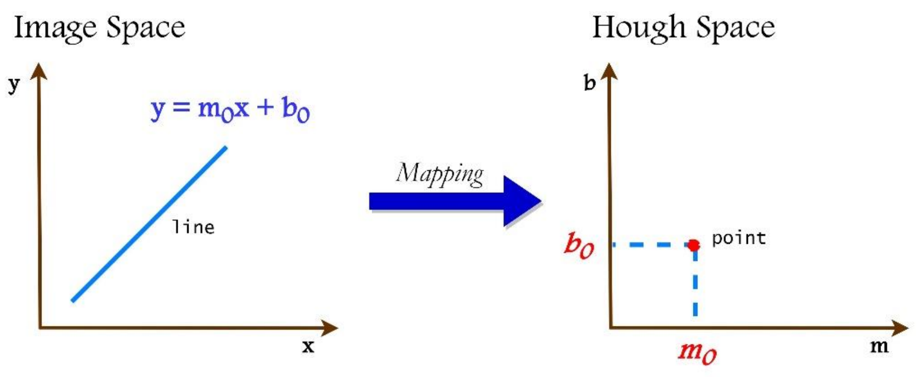


Figure 5.2 - Mapping of a line to the Hough space

Hence, the strategy to find lines in image space will be to look for intersecting lines in Hough space. To do this, it is possible to divide the Hough space in a grid in such a way to define intersecting lines and all lines passing through a given grid cell (accumulator cells). The Hough space is discretized into accumulator cells (initially zero matrix) which represent the best method to correctly determine whether certain points should be considered as part of a line or not. Each cell takes on a particular value determined by the number of intersections present within it, and the one with the highest value represents the line with the parameters of m and b best suited for adaptation.
After executing the canny edge detection algorithm to find all points associated with edges in the image, and considering every point in this edge-detected image as a line in Hough space, where many lines in Hough space intersect, it is easy to find a collection of points that describe a line in image space. Vertical lines have an infinite slope (if $m \to \infty$, the gradient is infinity) in the m-b representation, indeed they cannot be represented by the equation 5.6, so it is necessary to obtain a new parameterization.
If redefine out line in **polar coordinates system** as follows:

$$\rho = x cos\theta + y sin\theta \tag{5.7}$$

The equation representing a line can in turn be rewritten as:

$$y = -\frac{cos\theta}{sin\theta}x + \frac{\rho}{sin\theta} \tag{5.8}$$

Where the two parameters **ρ and θ** describe respectively the perpendicular distance of the line from the origin, and the angle of the line away from horizontal (Figure 5.3).

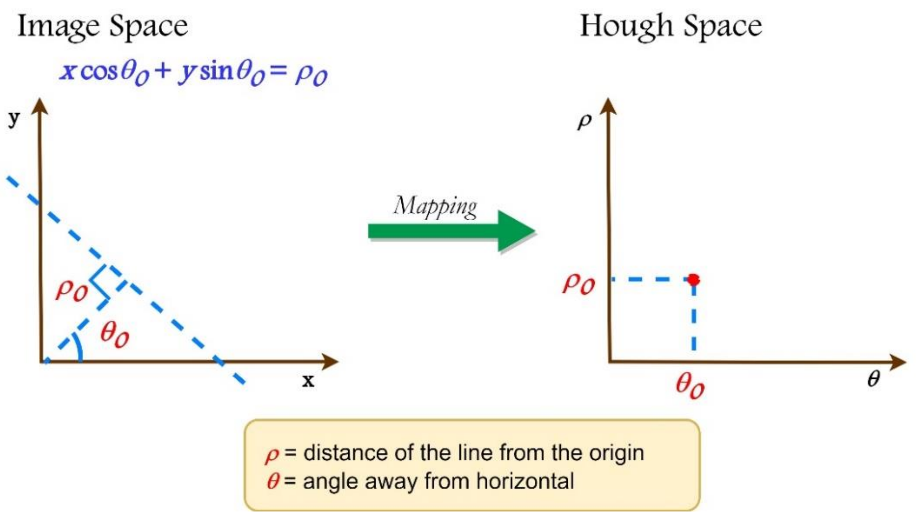


Image Space
$x\cos\theta_O + y\sin\theta_O = \rho_O$
y
x
ρ_O
θ_O
Mapping
Hough Space
ρ
ρ_O
θ_O
θ
ρ = distance of the line from the origin
θ = angle away from horizontal
Figure 5.3 - Mapping from Image Space to Hough Space

Now each point in image space corresponds to a sine curve in Hough space. Taking a whole line of points, it translates into a whole bunch of sine curves in Hough space. In general, when several curved lines intersect in the Hough space it means that the line represented by that intersection corresponds to more points.
The intersection of those sine curves, in **θ - ρ** space gives the parameterization of the line. In order to carry out the task of finding the lane lines, it is necessary to specify some parameters to identify the type of lines to be detected using an OpenCV function called Probabilistic Hough Line Transform: **cv2.HoughLinesP()**. This function implements an optimized version of Hough Transform, which uses a random subset of points sufficient for line detection. The function uses as arguments: the image frame, the resolution in pixels of the distance and the angular resolution in radians of the accumulator cells (Hough grid), the threshold that represents the minimum number of votes that must be used to nominate a line as such, a place-holder, and finally the minimum number of pixels to form a line and the maximum distance in pixels between disconnected lines. This last function is collected in **do_hough()** function.

Finally, the last operation that will be performed will be the optimization one. In particular, an average of the right and left lane lines is performed to obtain the coordinates for a single line one for the left and one for the right lane lines. Note that given these coordinates of points (x1, y1 and x2, y2) for both side, it is immediate to calculate their slope using the equation:

$$m = \frac{y2 - y1}{x2 - x1} \tag{6.9}$$

This is done using the numpy library and applying a least squares polynomial fit (**np.polyfit()**). This will fit a first-degree polynomial the linear function of y = mx + b, in order to fit the polynomial to the coordinates (x, y) points and return a vector coefficient of slope in y-intercept that minimises the squared error.
To be considered of fundamental importance is the fact that the lines of the lane have a particular fixed property that can be exploited through the slope formula. This property consists that the lines on the left are inclined towards the right and so have a negative slope, while the lines on the right are inclined towards the left and they therefore have a positive slope (see **calculate_lines()** function for more details).
It is therefore possible to draw and highlight with green color lane lines up to a length that starts from the lower part of the image and reaches about 2/3 of the total height of the image (**calculate_coordinates()** and **draw_lines()** functions). This is done in such a way as to obtain the same length of the lines identified at each iteration. Finally, the **cv2.addWeighted()** function superimposes the green lines previously detected through Hough method with the image frame in a weighted way. The final pipeline is visible in Figure 5.4. For more details, see SLLD.py in Appendix.

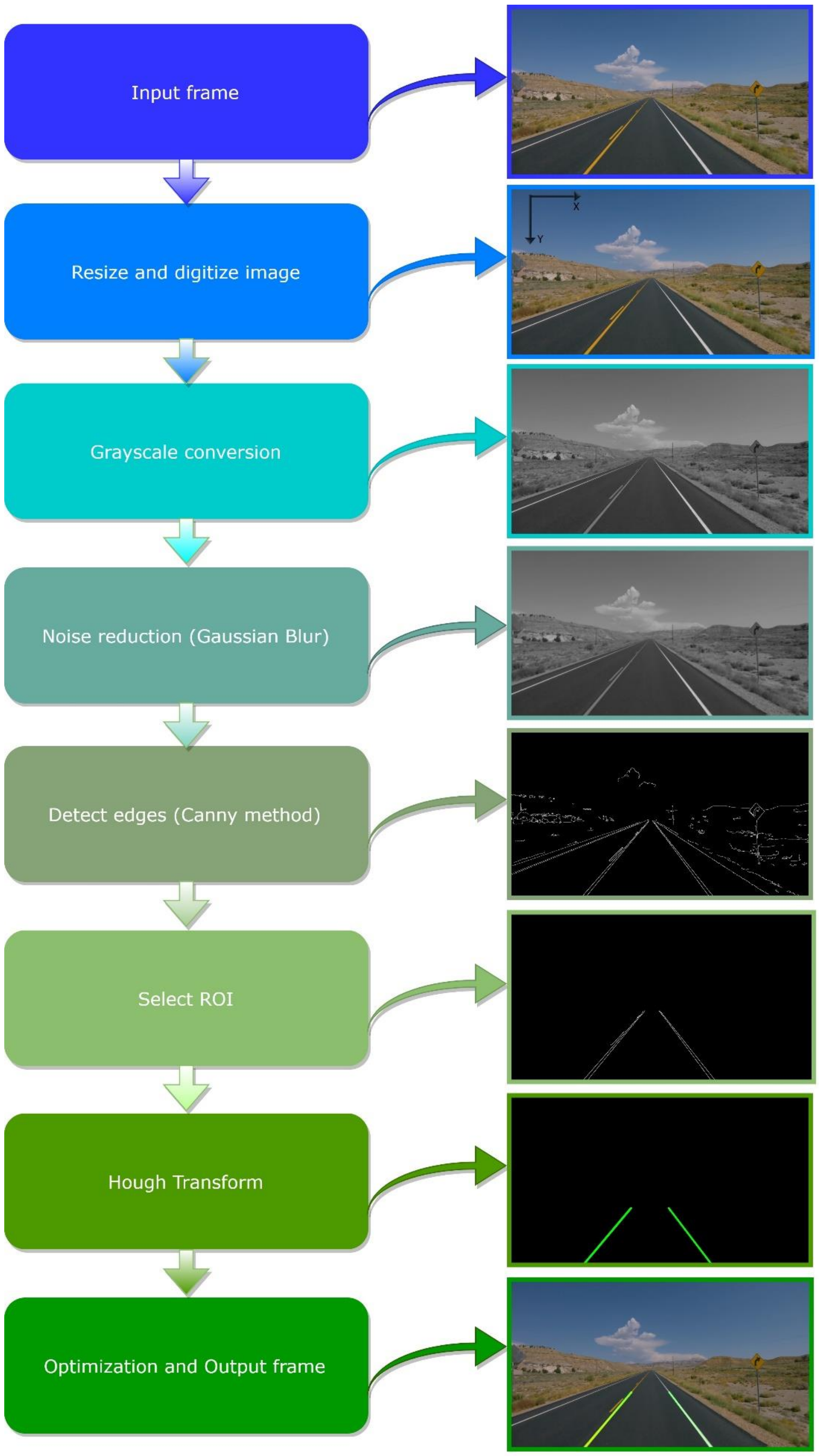

Input frame
Resize and digitize image
X
Y
Grayscale conversion
Noise reduction (Gaussian Blur)
Detect edges (Canny method)
Select ROI
Hough Transform
Optimization and Output frame

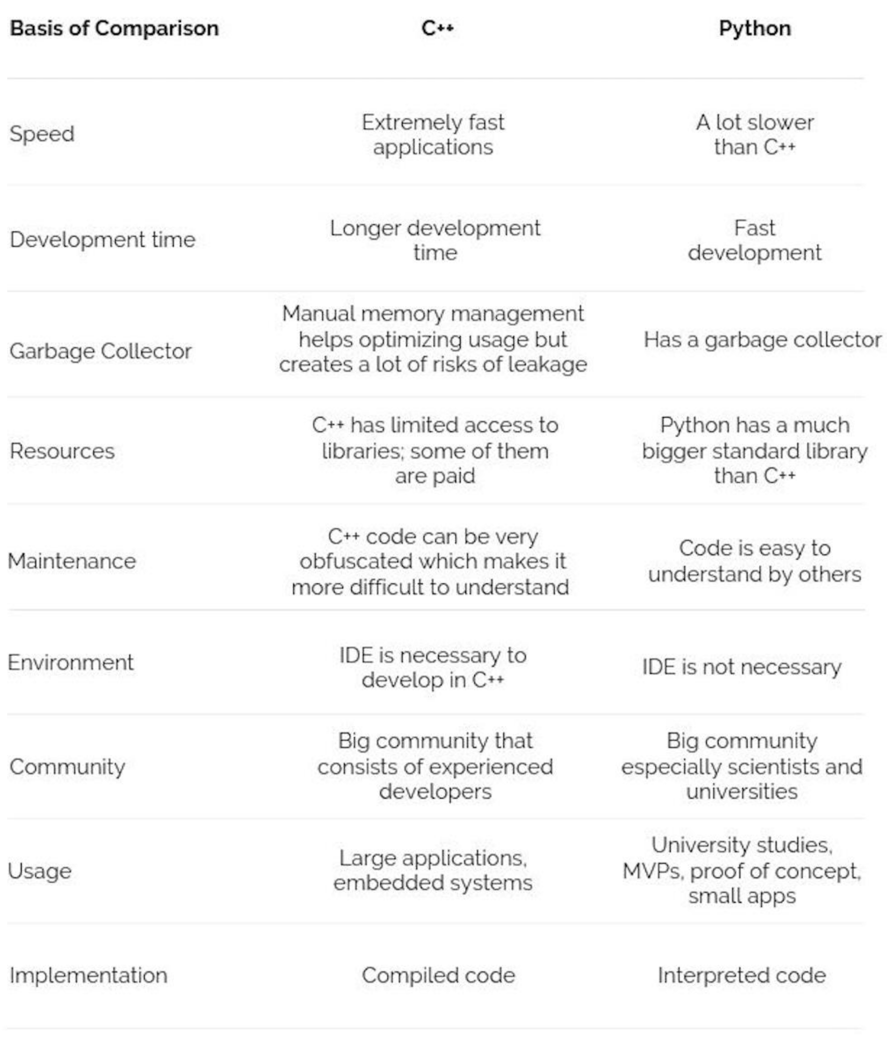
Figure 5.4 – SLLD Algorithm's pipeline

6. Advanced Lane Line Detection

The fundamental requirement for the implementation of methods that act as a support for lane maintenance systems lies in calculating the exact position of the vehicle with respect to the lane lines. For this purpose, it is necessary to use more in-depth computer vision techniques to recognize particular features used to determine lane detection.
This part is entirely dedicated to the analysis of an advanced method compared to the previous one in the identification of lane lines, which also allows adding useful information such as the radius of curvature of the lane and the position of the vehicle with respect to the lane center, using OpenCV library (and only for python script in combination with matplotlib library).

The algorithm was implemented using what was learned from the Udacity course on self-driving cars[4], enhancing and improving detection methods. The program was written in python and subsequently translated into C++ language.
This choice was dictated by the fact that there are several programming languages present throughout the world, and many of these are exploited for various applications, purposes and disparate uses. Among all, Python language is often compared to other languages, in particular with C++ language. Although they are very similar, these two languages have some not negligible differences. An example is the fact that the C++ language provides the particular functionality compilation that python does not have.
A big difference between the two is that Python is a high level interpreted language, it is dynamically typed and it has a dynamic memory allocation system, while C++ is a compiled intermediate level programming language and it is statically typed (i.e. you must declare the data type before using it). For this reason, C++ has a more complex syntax than python. Python, on the other hand, is built for its simplicity and portability and has a series of optimized libraries that are highly suitable for image processing like matplotlib library. Bearing in mind that the interpretation of the code is always slower than its compilation, C++ is significantly faster than python (more than twice), creating more efficient programs in terms of time, and for this reason it is widely used in most embedded systems. Moreover, C++ is able to communicate directly with the processor, and for this it is possible to optimize a system so that the application performs its task faster.
For a concise comparison of the features between the two (C++ and Python) programming languages see Figure 6.1.

[4] https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013



Basis of Comparison	C++	Python
Speed	Extremely fast applications	A lot slower than C++
Development time	Longer development time	Fast development
Garbage Collector	Manual memory management helps optimizing usage but creates a lot of risks of leakage	Has a garbage collector
Resources	C++ has limited access to libraries; some of them are paid	Python has a much bigger standard library than C++
Maintenance	C++ code can be very obfuscated which makes it more difficult to understand	Code is easy to understand by others
Environment	IDE is necessary to develop in C++	IDE is not necessary
Community	Big community that consists of experienced developers	Big community especially scientists and universities
Usage	Large applications, embedded systems	University studies, MVPs, proof of concept, small apps
Implementation	Compiled code	Interpreted code

Figure 6.1 - Main comparison between C++ and Python [33]

The steps of this project, which exploit various and complex artificial vision techniques to extract particular characteristics of road lanes from image frames are the following:

1. Read the frame image and resize it to the appropriate resolution
2. Apply the combined Gradient Thresholds to the image frame returning a binary output image to detect edges
3. Apply the combined Color Thresholds to the image frame in order to create a binary image that correctly identifies the pixels that represent lane lines through color selection
4. Merge both solutions of points 2 and 3 in a single binary image
5. Apply a perspective transform (BEV) using the ROI vertices to correct the binary image
6. Apply a lane lines finding method using histogram peaks and sliding window
7. Identify lane line pixels fitting their positions using a second order polynomial fit to find the lane boundary

8. Determine the radius of curvature of the lane and the offset of the vehicle with respect to the center of the lane
9. Use the inverse perspective transformation to get back the original image and draw the lane lines on it
10. Display the final result with the information associated to the point 8

In order to be able to extract particular information from a lane in an image in a reliable and precise way, it is necessary to look for and highlight some typical features in the image. First, the image is read and resized to HD resolution (Figure 6.2) through OpenCV library in such a way to be able to subsequently identify more easily the ROI vertices.

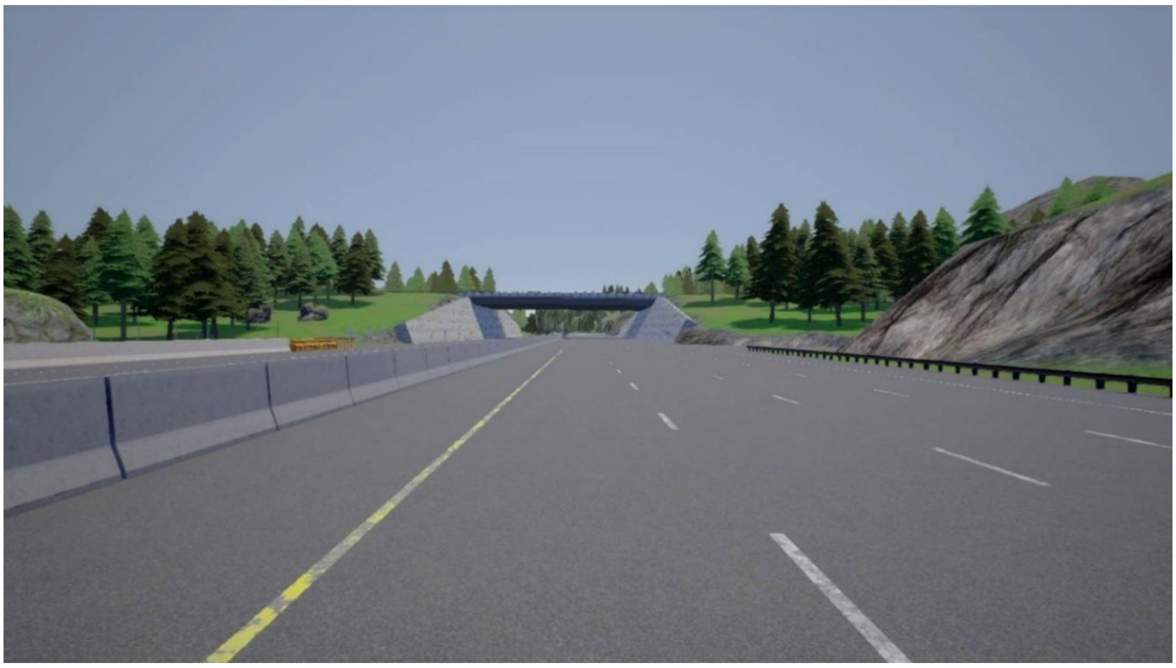


Figure 6.2 - Input image (snapshot) taken from Carla Simualtor Town04 using ADAS_scenario.py

In the following 2 phases, several transformations are applied to the initial image, which are then intelligently combined together in order to obtain a clear and reliable binary image capable of accurately detecting lane lines. Previously, using the Canny method it was possible to find all the possible lines in an image, but in addition to the detection of the lane lines the algorithm also provided many edges on landscapes, cars and other objects of no interest which were then discarded. Regarding the search for lane lines, it is known in advance that the lines we are trying to identify can be approximated by vertical lines and both lane lines generally have a high contrast with respect to the road.
For this reason, it is possible to take these advantages by using gradients in a smarter and more efficient way to detect steep edges that are more likely to be identified as lanes in the first place.

The Sobel operator is the main core of Canny's edge detection algorithm. Applying the Sobel operator to a given image it is possible to obtain the derivative of the image in the x or y direction in order to filter what we are looking for. Keeping in mind that the kernel size can be represented by any number, as long as it is odd, and using the minimum size of a kernel (which implies a 3x3 operator in each case) the Sobelx and Sobely operators, respectively, can be represented as follows:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{6.1}$$

Note that $S_x = S_y^T$ and $S_y = S_x^T$

Using a larger kernel involves applying the gradient over a larger region of the image obtaining a uniform gradient. If an image is flat in a given region (this occurs when there is a small change in values across the given region), then the resulting gradient will be zero. This can be achieved by summing the element-wise product of the matrix operator and corresponding image region pixels. The gradient in both direction can be calculated as follow:

$$G_x = \sum(region * S_x) \tag{6.2}$$

$$G_y = \sum(region * S_y) \tag{6.3}$$

Equations 6.2 and 6.3 are equivalent to the equation 5.3 seen before. If, on the other hand, the Sobel operator is applied in an image region in which the values decrease from left to right (as regards Sx) or from top to bottom (as regards Sy), the result of the gradient will be negative, and this therefore implies a negative derivative.
Conversely, if Sobel operator is applyed to an image region where the values increase in the same direction as in the previous case (both for Sx and Sy), then the gradient result will be positive and therefore this implies a positive derivative.

By obtaining the gradients of the images in both the x and y directions, it is possible to notice how these detect lane lines and collect different edges. In particular, the gradient in the x direction highlights the edges closest to the vertical, while the gradient in the y direction emphasizes the edges that are closest to horizontal lines. The Sobel operators combine Gaussian smoothing and differentiation, this implies that the output will be noise resistant.
The **abs_sobel_threshold()** function (both in python and in C++ programs) calculates the image derivative by convolving the image with the appropriate kernel. Moreover, examining the function in detail, this is able to calculate the gradient (first derivative) along the x or y direction (Sobel x or Sobel y operator) of original input image converted to

grayscale (1 channel) by convolving this image with the appropriate kernel size (always-odd number, in this case 3x3). Then it takes the absolute value of the derivative and scales the result to 8 bit (0 - 255) by returning an output binary image, where pixels have a value of 1 or 0, based on the strength of each gradient applying a threshold determined by threshold min and threshold max values. By properly tuning the threshold parameters, it is possible to obtain a good estimate of the lane lines.
The Sobel operator used to obtain the gradients both in the direction of x and of y is also used to obtain images with magnitude and direction thresholds. The magnitude of the gradient represent the cornerstone of the detection of the edges of the Canny algorithm and that is why Canny works well to collect all the edges. It is calculated as seen before in the equation 5.4 and it is implemented through the **magnitude_threshold()** (both in python and C++ programs) function, where taking as input the converted grayscale image, the gradient in x and y is computed separately (with the same kernel size: 3x3) for calculating the magnitude. After that, scaled the result to 8 bit, the binary image is returned as output delimited by the two thresholds (min and max).
On the other hand, the direction of the gradient instead, is much noisier than the gradient magnitude, but it is useful calculate it to find particular features by orientation. The function that performs this operation on the converted grayscale image is **direction_threshold()** (both for python and C++ programs) where the direction is computed following equation 5.5. In particular, it takes the absolute value of the x and y gradients, computed separately, and calculates the direction returning a binary mask where direction thresholds (min and max) are met. The different thresholded binary gradients results images are visible in Figure 6.3.

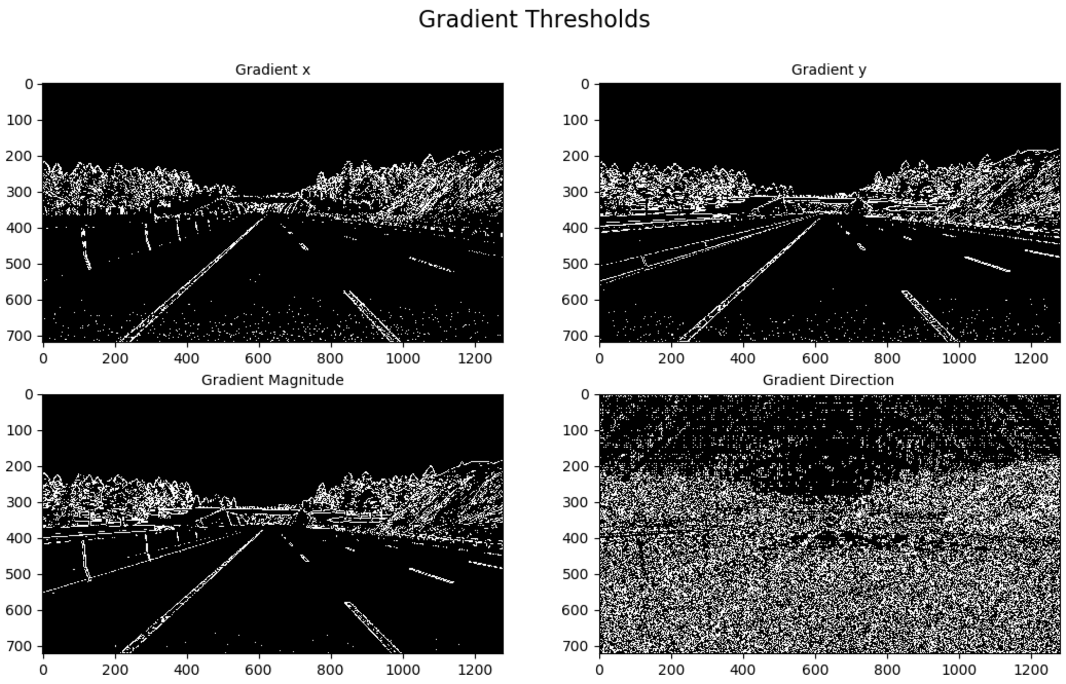


Figure 6.3 - Thresholded binary gradients images

It is possible to exploit combining the information of the x and y gradient threshold, the amplitude of the general gradient and the direction of the gradient, based on threshold values, to focus on the precise identification of the pixels that determine the lane lines.
The **apply_thresholds()** function (both for python and C++ programs) performs this task, by doing 2 bitwise AND operation between the gradient in the x and y direction and between magnitude and gradient, combining the results using a bitwise OR operation. The combined result is visible in Figure 6.4.

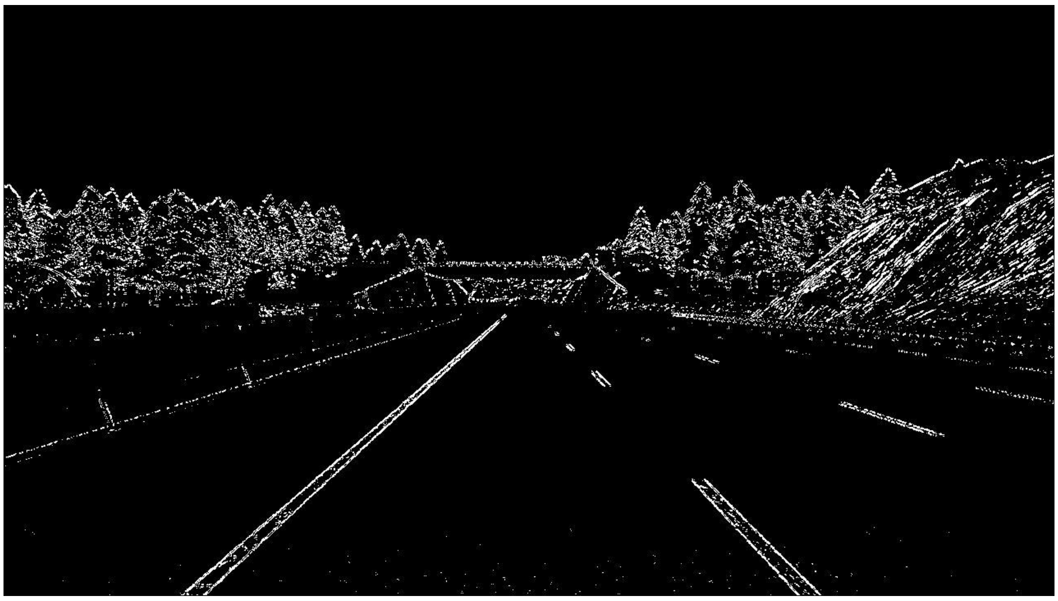


Figure 6.4 - Combined Binary Gradient

These operations were performed on the original image converted to grayscale to make it easier to detect the edges. However, when performing this conversion, valuable information on color, such as yellow, is lost, which would be useful to give more strength to the identification of lane lines. For this reason, it is useful to take advantage of color spaces that provide more information about an image than just the gray scale in order to consistently detect objects under varying light conditions.
A color space is a particular organization of colors providing a way to classify colors and represent them in digital images. This space can be considered as a 3D space, in which any color can be represented by a 3D coordinate of values R (red), G (green) and B (blue). For example, white has the coordinates (255, 255, 255), and therefore has the maximum value for R, G and B. However, RGB thresholding does not work that well in images that include the variation of light conditions or whenever lanes are of a different color like yellow. It works best on white lane pixels. Therefore, it is possible to split an image into separate R-G-B components, which are often called channels. The brightest pixels indicate higher values of red (R), green (G) or blue (B) respectively. The R and G channels detect white and yellow lane lines and are therefore the most useful for isolating lane pixels, while channel

B does not detect the yellow lane line. All channels vary according to different brightness levels.

There are many other ways to represent colors in an image besides the RGB values (Figure 6.5 shows the most common ones) inspired by the human vision system. The most common color spaces used most frequently in image analysis and processing [34] are transformations of a Cartesian RGB color space: the HSV color space (which stands for hue (H), saturation (S) and value (V)) and HLS space (which stands for hue (H), lightness (L) and saturation (S)).
For both of these, H has a range, which span from 0 to 179 for degrees around the cylindrical color space. HLS Color Space isolates the lightness (L) component, which varies the most under different lighting conditions. H and S channels stay consistent in shadow or excessive brightness.
Hue is the value that represents color independent of any change in brightness, on the other hand, Lightness and Value represent different ways to measure the relative lightness or darkness of a color and finally saturation is a measurement of colorfulness. The **cv2.cvtColor()** function from OpenCV library is able to converts images from one color space to another one.

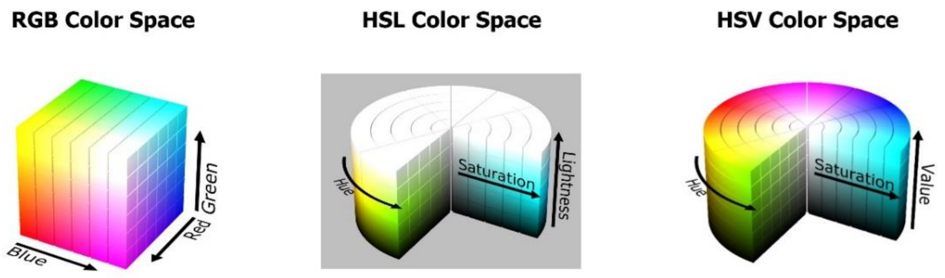


Figure 6.5 - Example of different color spaces [35]

Therefore it is important not only to view each color space channel in such a way as to be able to distinguish and choose the one that is best able to identify lane lines but also use a channel that is more robust than the others and under changing conditions. Assuming that lane lines can be either yellow or white, different color spaces channels were tested using different thresholds.
At the end, among all, 4 different color spaces were used: RGB, HLS, LUV and LAB to help detect lane lines of different colors and under different lighting conditions.
Consequently, it has been concluded that the best combination to detect the lane lines of the road is to use the following channels:

- The R channel of the RGB color space: it has the appropriate information to identify the white lane lines as well as yellow

- The S channel from the HLS color space: it was used to highlight the yellow lane lines uniformly
- The B channel from the LAB color space: it is useful for enhancing the identification of the yellow lanes especially in low light conditions since it has the highest signal-to-noise ratio
- The L channel from the LUV color space: it is able to detect white and yellow lane lines almost perfectly

This combination is very effective and robust to detect lane lines in different lighting and road conditions as it offers redundancy and therefore the possibility of detecting the pixels of both lane lines (white or yellow) even in the case where 1 of them should fail. The results can be seen in Figure 6.6.

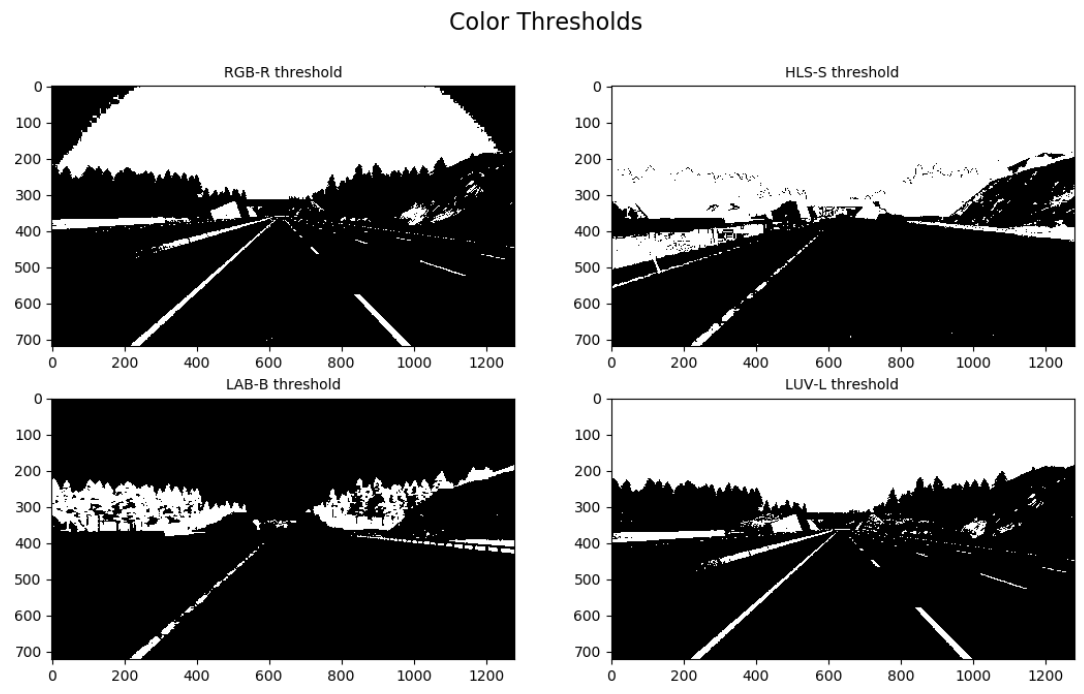


Figure 6.6 - Thresholded binary color channel images

These channels are filtered through a logical AND operator ("&") with particular thresholds (threshold_min and threshold_max) that are different for each type to correctly isolate the pixels of the lane lines. Threshold values strongly depend both on the type of image being analyzed and on the type of channel taken into consideration. After that, their result is combined together via a logical OR operator ("|") to make the binary output image robust and reliable in detects lane lines pixels even with different colors.
The **apply_color_threshold()** function (both for python and C++ programs) summarizes these operations. The result is visible in Figure 6.7.

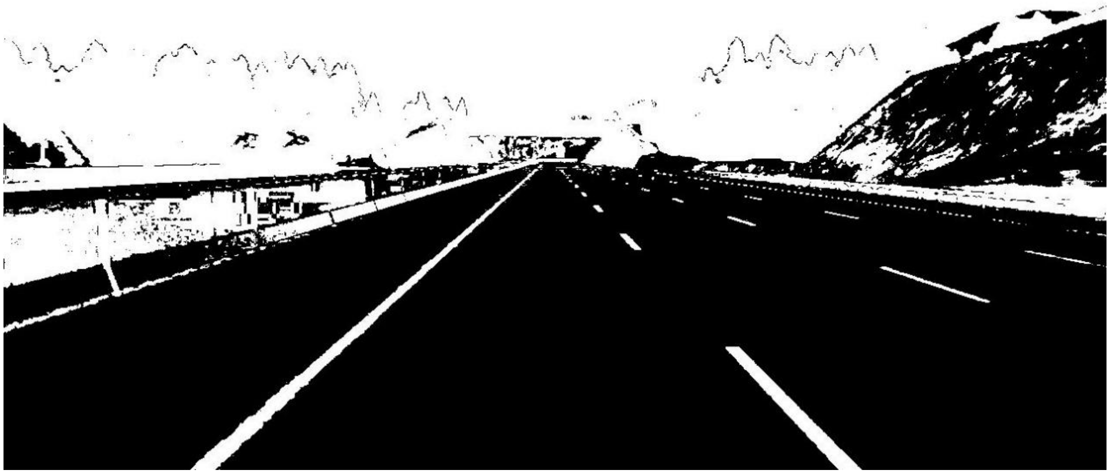


Figure 6.7 - Combined binary color

Finally, to minimize false detections and make identification even more effective, an OR operation was performed between color binary image and gradient binary image previously calculated (combining them), putting together in a single image the peculiarities of both solutions (**apply_combined_threshold()** both in python and C++ programs). Below, Figure 6.8 summarizes schematically the combination of the thresholds between gradient and color in order to create a binary image with combined threshold. Figure 6.9, instead, show the output result.

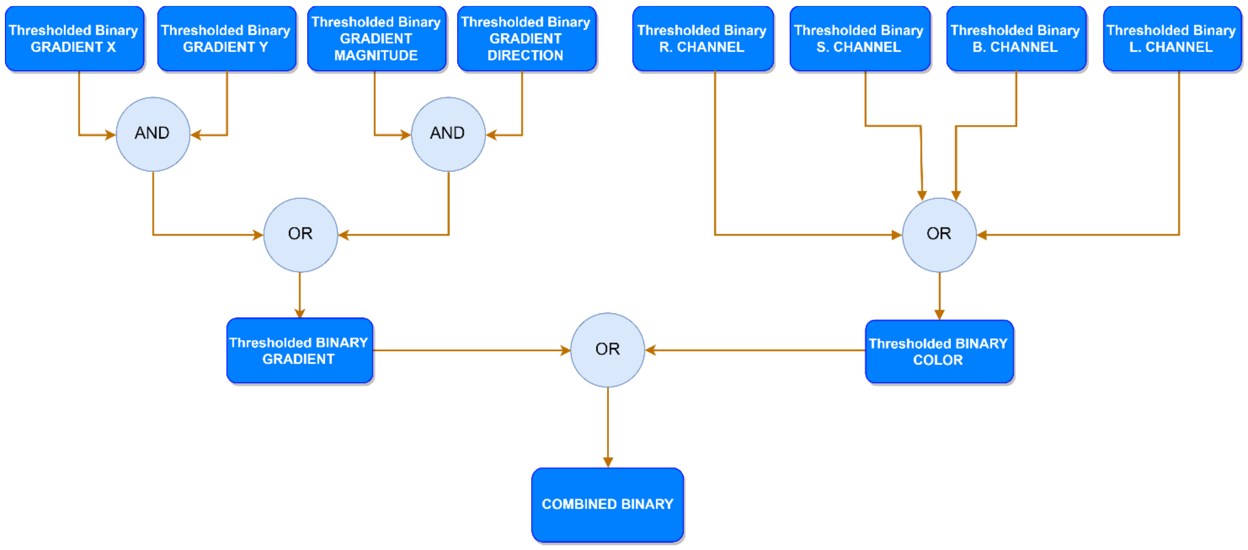

Thresholded Binary GRADIENT X
Thresholded Binary GRADIENT Y
Thresholded Binary GRADIENT MAGNITUDE
Thresholded Binary GRADIENT DIRECTION
Thresholded Binary R. CHANNEL
Thresholded Binary S. CHANNEL
Thresholded Binary B. CHANNEL
Thresholded Binary L. CHANNEL
AND
AND
OR
OR
Thresholded BINARY GRADIENT
OR
Thresholded BINARY COLOR
COMBINED BINARY

Figure 6.8 - Summary scheme to create a combined binary image

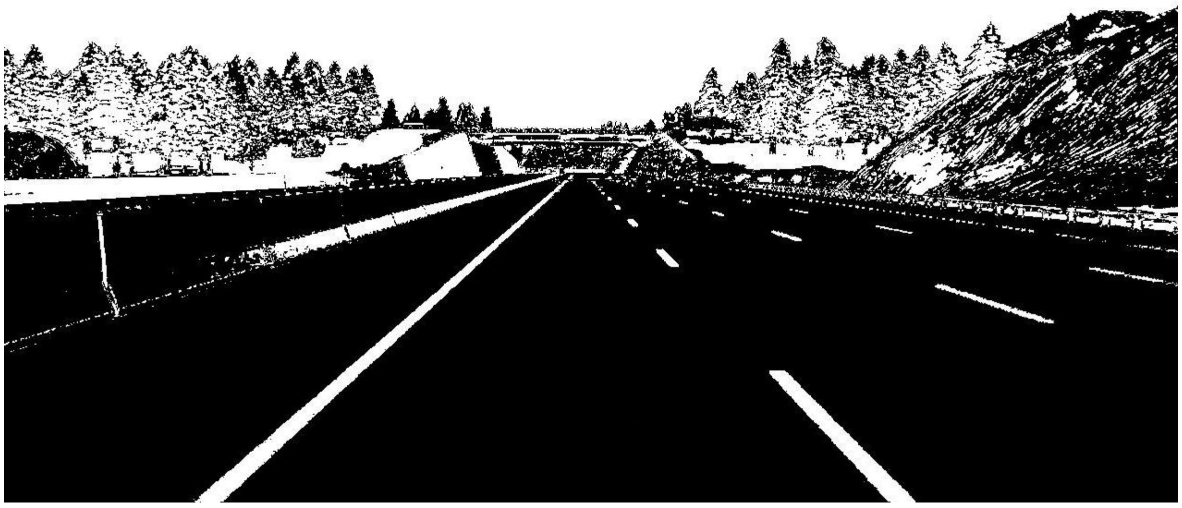


Figure 6.9 - Combined binary image

Using the current perspective space is not convenient for the calculation of lane distances. For this reason, to eliminate the perspective effect and simplify the calculations it is useful to exploit the perspective transformation, in particular the bird's eye view, which represents a valid tool both for the detection and for the adaptation of the lanes.
Assuming that the lane lines lie on a flat 2D surface, it is possible to take advantage of a bird's eye perspective transformation (that is, observe the same image from above) both to see the parallel lane lines (more or less depending on curvature), and to enlarge those lane lines distant from the vehicle, which otherwise would be small and therefore difficult to identify. This is extremely useful, especially for road images, since it allows visualizing the lanes from above in such a way that it is easier not only to apply model fitting to this top-view image that can accurately represent the lane, but also to calculate the curvature of the lane and the vehicle's offset.
The idea behind the perspective transformation is based on mapping the pixels of the original image (front view) to an image with a new perspective (viewed from above). The BEV (bird's eye view) perspective transformation can be applied to an image through 2 openCV function: **getPerspectiveTransform()** to calculate a perspective transform matrix based on four pairs of points (source points and destination points) and **warpPerspective()** function to apply it on the original image (Figure 6.11). These 2 functions are used by the **binary_transform()** function (both for python and C++), which through the source and destination points computes the perspective transformation matrix and its inverse, and applies it to the binary image previously obtained, so returning a binary warped image (see Figure 6.12).

The source points of interest represent the 4 coordinate points in the original image that lie on the plane in the physical world that are mapped to the destination points of interest, which represent the 4 coordinate points in the warped image. It is necessary to note that the four source points must be chosen accurately as they represent the fundamental task of selecting the region of interest (ROI) having the typically trapezoidal shape, clearly visible in Figure 6.10.

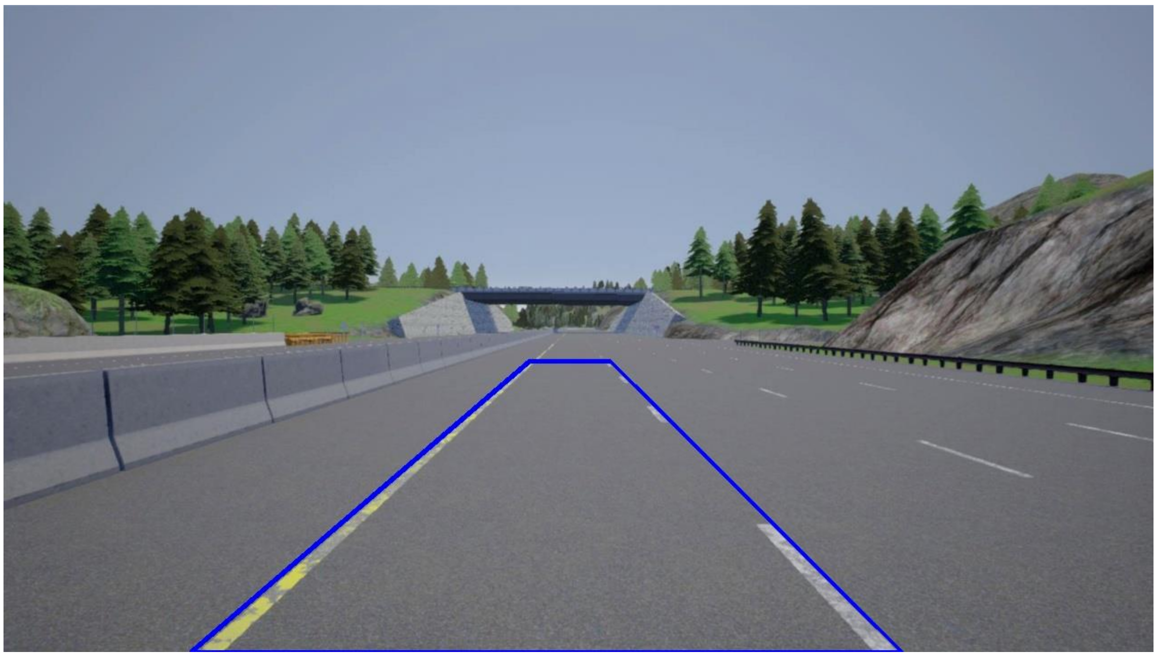


Figure 6.10 - Image ROI (Region of Interest)

In particular, the upper part of the trapezoid must be sufficiently high in the original image to be able to detect the lanes located near the horizon to allow the algorithm to both adequately calculate the curvature of the lane and to increase the number of line segments in order to perform a better fit. To search for these points correctly and quickly, as in chapter 5, the **find_ROI_coordinates.py** python script can be used.

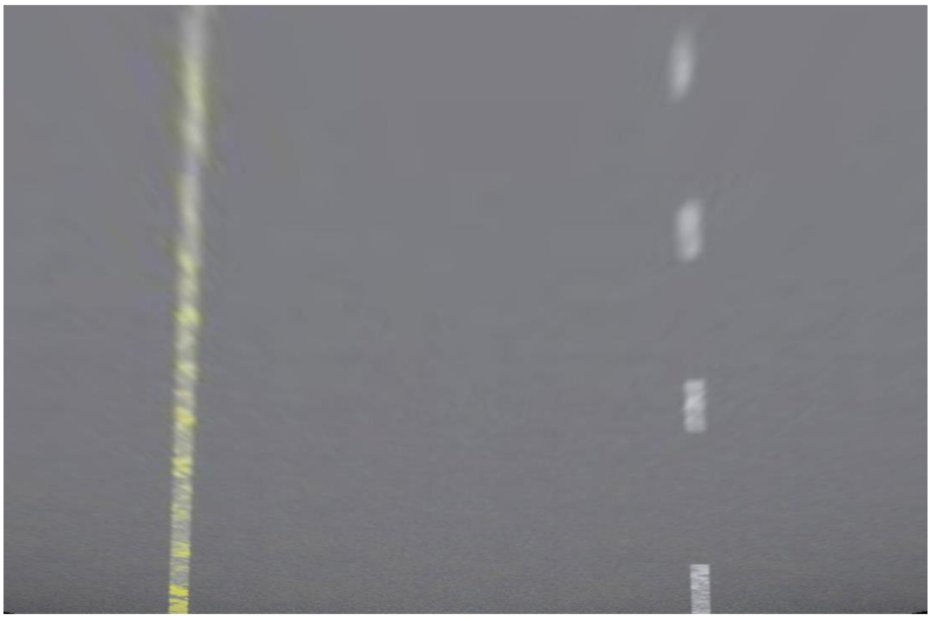


Figure 6.11 - Warped image

It is important to note that the trapezoid deriving from the coordinates of the source points in the original image is transformed (mapped) into a rectangle in the warped image.
Another thing that is worth noting is that, when the algorithm reconstructs the warped image through the bird's eye view, this is more blurred in all those points where the lane lines are more distant than the vehicle because there are fewer pixels for the reconstruction bringing the algorithm to lower the resolution. By obtaining the perspective transformation, it is possible to visually check whether the lines are approximately straight or not. After applying a correct perspective transformation to the road image, the output obtained will be a binary image in which the lane lines stand out visibly (Figure 6.12).

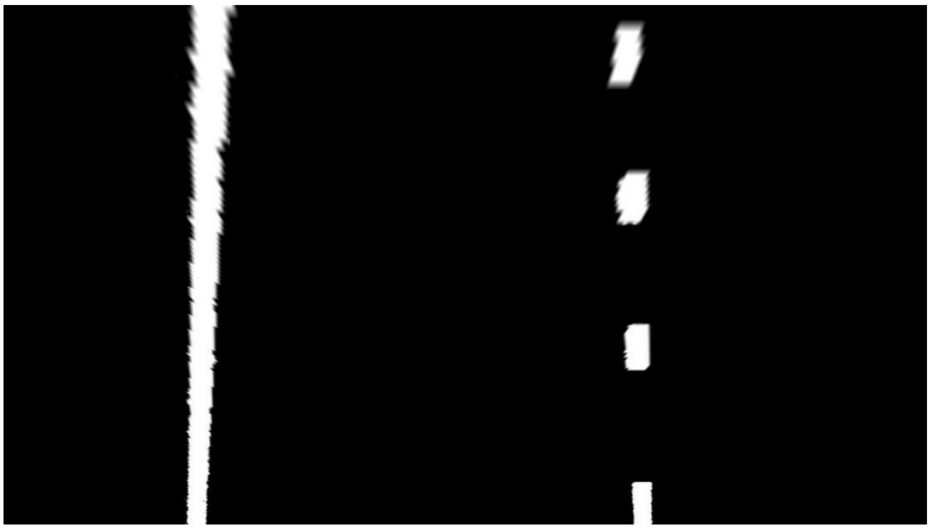


Figure 6.12 - Binary warped image

However, it is necessary to establish which pixels must be part of the lane lines and especially which ones belong to the left or right line. A quick and efficient solution to this problem lies in representing a histogram capable of tracing binary activations in order to find the starting point of lane lines, following the same approach used in GOLD [16].
The **get_histogram()** function (both for python and C++ programs), fulfills this work. In this way, using the histogram, the values of each pixel are summed along each column in the image, since lane lines are likely to be mostly vertical nearest to the car.
In the deformed binary input image, the pixels forming the lane lines are represented by white (logic value 1) instead the pixels that are not part of them are represented by black (logic value 0). Thus, the two most important peaks (i.e. those with the highest value) in the histogram can be easily identified as candidates that will indicate the x position of the starting point of the lane lines. The result is shown in Figure 6.13.

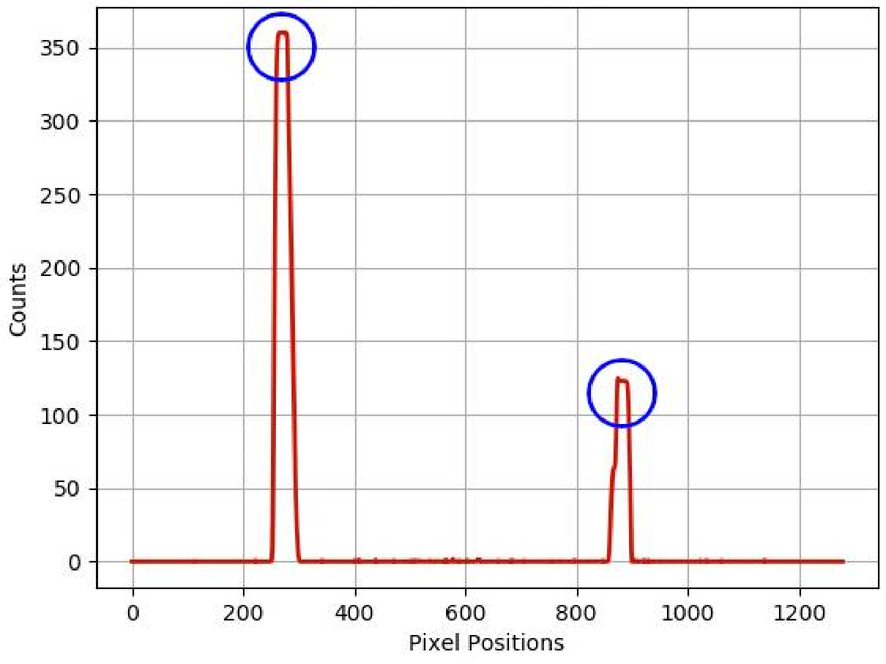


Figure 6.13 - Histogram Peaks

The histogram is divided exactly in half, based on its midpoint value, a left and a right part in order to do the computation more easily. Then the maximum point in each part is calculated, which will be taken into consideration when the windows around these maxima will be created. In this way, it is possible to exploit together the deformed binary image and the histogram to be able to use the sliding window method. It is placed in the central position of the lane lines for both the right and left lanes, and it is used to find and follow the lines to the top of the entire image frame (i.e. at the maximum height) including the lines distant from the vehicle further along the road to determine their direction.
The steps to execute the sliding windows method described below are searchable in the **sliding_window()**, **skip_sliding_window()** and **display_poly()** functions for

Python program and **sliding_window()** and **draw_poly_lines()** functions for C++ program.
Firstly, the histogram of the bottom half of the image is taken and it is split into two sides, one for each lane line in order to get the left and right peak. These peaks will represent the starting point to reconstruct the left and the right lines. After entering the parameters a priori to set the appearance of the windows, also called hyperparameters (like the number of sliding windows, the height of windows, the width of the windows margin, the minimum number of pixels found in order to re-center window), and having previously obtained the lane lines starting points using the histogram, it is possible to trace the curvature of the right and left lane lines.

For both the right and left lane lines, the window will scroll to the left or right if it detects that the average position of the pixels activated within the window (i.e. if there are white pixels identifying the line) has moved. The steps required to implement the sliding window method [36] are the following:

1. Loop through each window in *nwindows* (total number of sliding windows)
2. Identify the current window boundaries, this is based on a combination of the current window's starting point (*leftx_current* and *rightx_current*), as well as the margin setted before
3. Draw windows on the image using **cv2.rectangle()** OpenCV function to be able to visually locate them in the space
4. Find out which activated pixels (nonzero pixels in x and y) from nonzeroy and nonzerox are inside the window
5. Enter these values in a list of indexes (to *left_lane_inds* and *right_lane_inds*), both for the right and left lane line
6. If the number of pixels found in step 4 is greater than the *minpix* parameter (i.e. the minimum number of pixels found needed to re-center window), re-center the window (ie *leftx_current* or *rightx_current*) based on the average position of these pixels
7. Extract left and right line pixel positions storing their x and y coordinates into a vector (leftx, lefty, rightx, righty)

After finding all the pixels belonging to each lane line through the sliding window method, it is necessary to fit a polynomial to both line. In particular, *polyfit()*, a OpenCV function, applies a least squares polynomial fit of order 2 to the lane lines, returning polynomial coefficients (see Figure 6.14). To identify the pixels of the lane lines, their x and y pixel positions were used to adapt to a polynomial curve of the second order in the polyfit equation form:

$$f(y) = Ay^2 + By + C \tag{6.4}$$

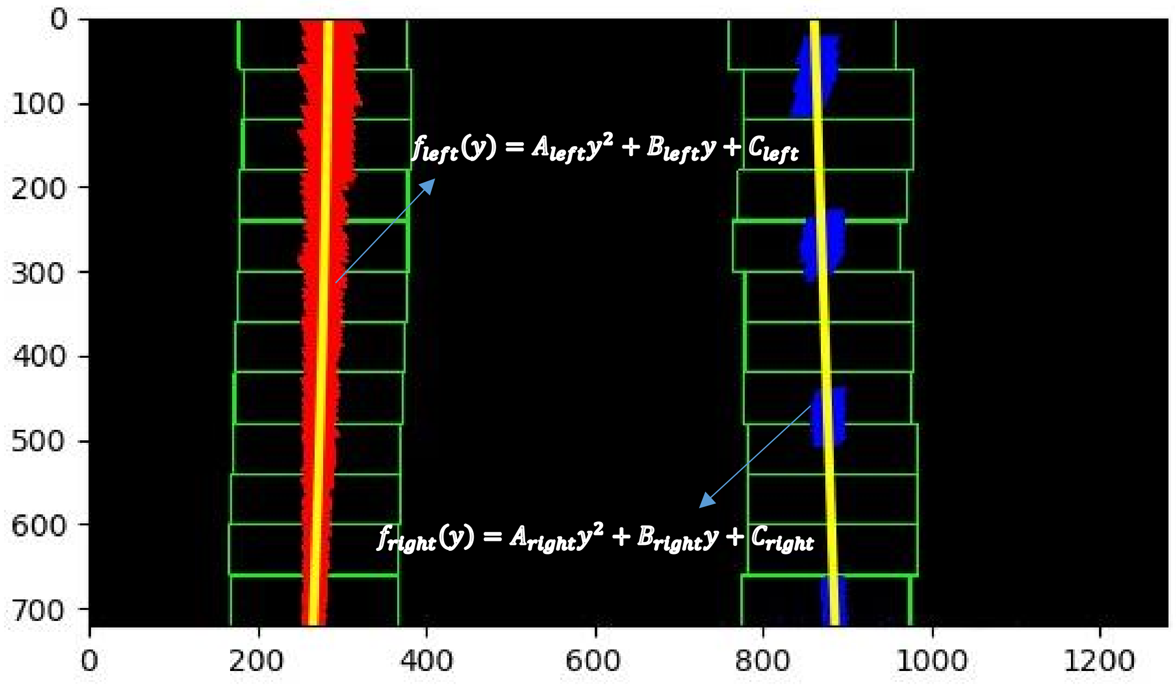


Figure 6.14 - Sliding Window method

It has been adapted f (y), rather than f (x), because the lane lines in the image are deformed almost vertical and may have the same x value for more than a y value. Once the method that uses the sliding windows to trace the lanes in the distance has been created, it is necessary to keep in mind that the use of this for each frame (for example of a video) could take a long time for the processing and therefore be inefficient.
Since the lane lines do not move much from one frame to another in a video, it is not necessary to search for lanes again using the algorithm from the beginning, but more intelligently it is possible to search in a margin (set a priori) around to the position of the previous line (skipping the sliding window). This works as a sort of "Kalman filter" to predict the position of the lane lines in the next frame within an area of margin defined a priori. In this way, knowing exactly the position of the lines in a frame, it is possible to perform a targeted search around them in the next frame (Figure 6.15). This makes it easier to follow lane lines, as this method is equivalent to applying a custom region of interest for each frame. In particular, polynomial functions are used to determine which activated pixels fall within the areas of interest established by a margin (+/-).

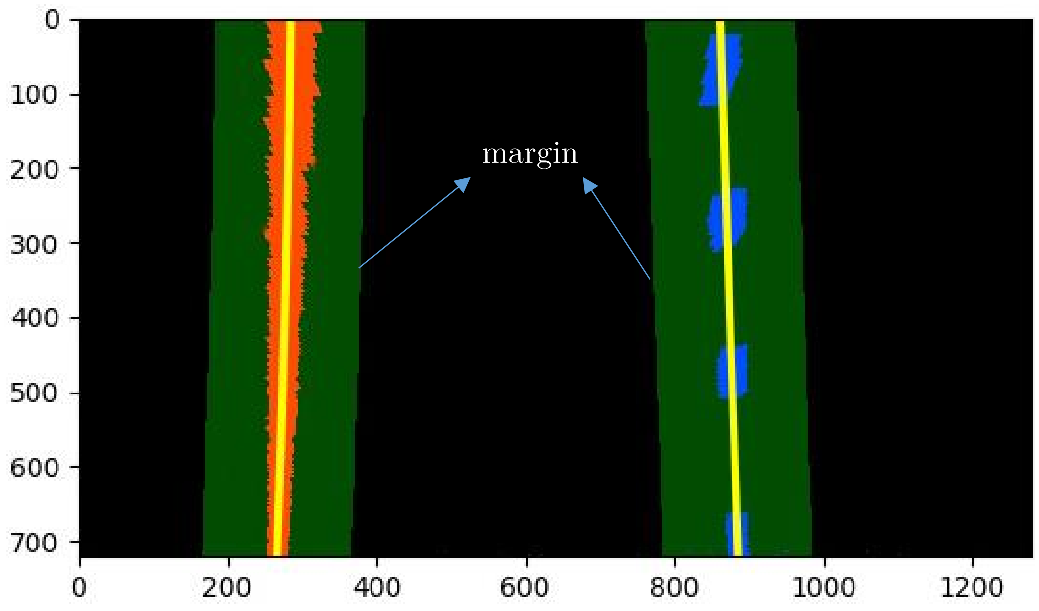


Figure 6.15 - Skipping sliding window

In the event of losing the traces of the lane lines, it is possible to return to using the method of the previously addressed sliding window to know again their position (the position has changed compared to before).
After obtaining the polynomial adaptation for the lane lines, it is possible to calculate the radius of curvature of the road. The curvature radius of a curve at a given point can be defined as the radius of the approximate circle at that specific point (Figure 6.16). The curvature is dependent on the radius: the smaller the radius, the greater the curvature and vice versa, the greater the radius, the smaller the curvature.
In other words, a huge approximate circle, with an enormous radius of curvature, implies that at that point a straight line can represent the curve. The radius of curvature R can be calculated as follows [37]:

$$R = \frac{\left[1 + \left(\frac{dx}{dy}\right)^2\right]^{\frac{3}{2}}}{\left|\frac{d^2x}{dy^2}\right|} \tag{6.5}$$

It is obvious that the radius must be positive; this is evidenced by the presence of the absolute value in the denominator of the fraction. The radius of curvature is the reciprocal of the curvature K:

$$R = \frac{1}{K} \tag{6.6}$$

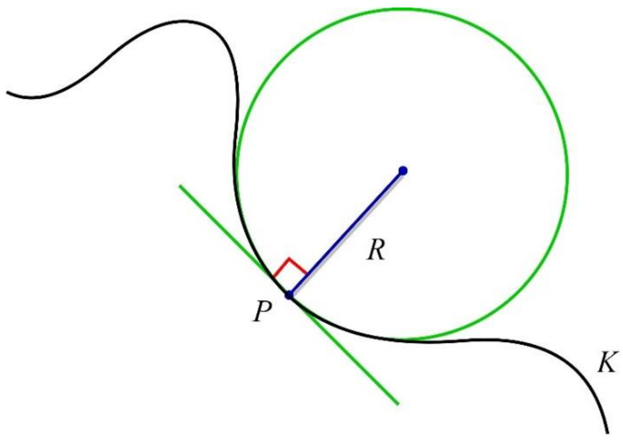


Figure 6.16 - Graphic representation of the radius of curvature R of a curve K and of its osculating circle which is tangent in the point P

Using the second order polynomial (equation 6.4), the first and the second derivatives are:

$$f'(y) = \frac{dx}{dy} = 2Ay + B \quad and \quad f''(y) = \frac{d^2x}{dy^2} = 2A \tag{6.7}$$

Therefore, using equations 6.7, the equation 6.5 can be re-written as:

$$R = \frac{(1 + (2Ay + B)^2)^{\frac{3}{2}}}{|2A|} \tag{6.8}$$

The radius of curvature was initially calculated based on the values of the pixels, then it was decided to choose a more realistic measurement that reflects reality and therefore before executing the calculation of the radius of curvature, the x and y coordinates have been converted from pixel space to real world space. This involves measuring the length (y) and width (x) of the lane section we are projecting into our deformed image. This data was extracted from CARLA simulator or also consulting the documentation [18]. In this way, using the **measure_curvature()** function (both for Python and C++ programs), the radius of curvature of the right and left lane lines in meter has been calculated by fitting a new polynomials to x and y in world space, computing also the difference and the average between the two curvatures.
Moreover, it is also possible to calculate the distance and direction of the vehicle with respect to the center of the lane, in order to know exactly where it is. This can be done, based on two assumptions:

- The camera is mounted in the center-front part of the car, so that the center of the lane corresponds to the midpoint in the lower part of the image between the two lines detected
- The width of the road lane must be known (3.5 meters in Carla Simulator)

The offset of the center of the lane from the center of the image (converted from pixel to meters) represents the distance of the vehicle from the center of the lane (see **measure_vehicle_offset()** function both for Python and C++ programs). In detail, the distance to lane center is computed as follows (see also Figure 6.17):

$$lane\ center = \frac{right_{Pos} + left_{Pos}}{2} \tag{6.9}$$

$$camera\ center = camera\_width/2 \tag{6.10}$$

$$Offset_{pixels} = lane\ center - camera\ center \tag{6.11}$$

$$Offset_{meters} = Offset_{pixels} * x_{\frac{meter}{pixels}} \tag{6.12}$$

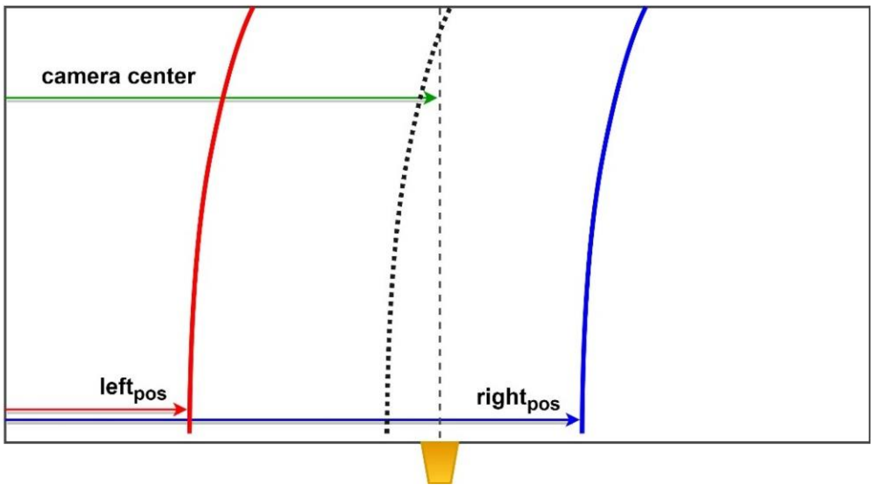


Figure 6.17 - Graphical representation of the parameters used to calculate the distance from the center of the lane

The offset of the car from the center of the lane was obtained by averaging the coordinates of the starting points for identifying the left and right lane lines, subtracting the latter from the middle point of the camera. Finally, to obtain a compliant measurement, the result is multiplied by the so called pixels-to-meters ratio, i.e. the ratio between the effective width in meters of the lane (3.5m in this case) and the pixels of the lane (720) so as to obtain a conversion from pixels space to meters (world space).

In addition, based on the offset value, it is also possible to understand in which direction the vehicle is moving in relation to the centre of the lane: if the offset is positive, the vehicle is positioned to the right with respect to the centre of the lane, and vice versa. At this point, it is therefore possible to project and highlight the lane lines by means of **fillPoly()/fillConvexPoly()** and **polylines()** OpenCV functions on the original image obtained using the inverse perspective transformation (IPM, **warpPerspective()** using inverse transformation matrix "Minv") to un-warp the image (going from a bird's-eye view to the original undistorted image).
In conclusion, a merger operation is executed using **addWeighted()** OpenCV function, blending the images into a single output image, showing at the same time on the screen the information about the radius of curvature and on the vehicle's offset (see **draw_lane_lines()** function both for python and C++ programs). The final result was thus obtained that is visible in Figure 6.18. For more details, see ALLD.py in Appendix.

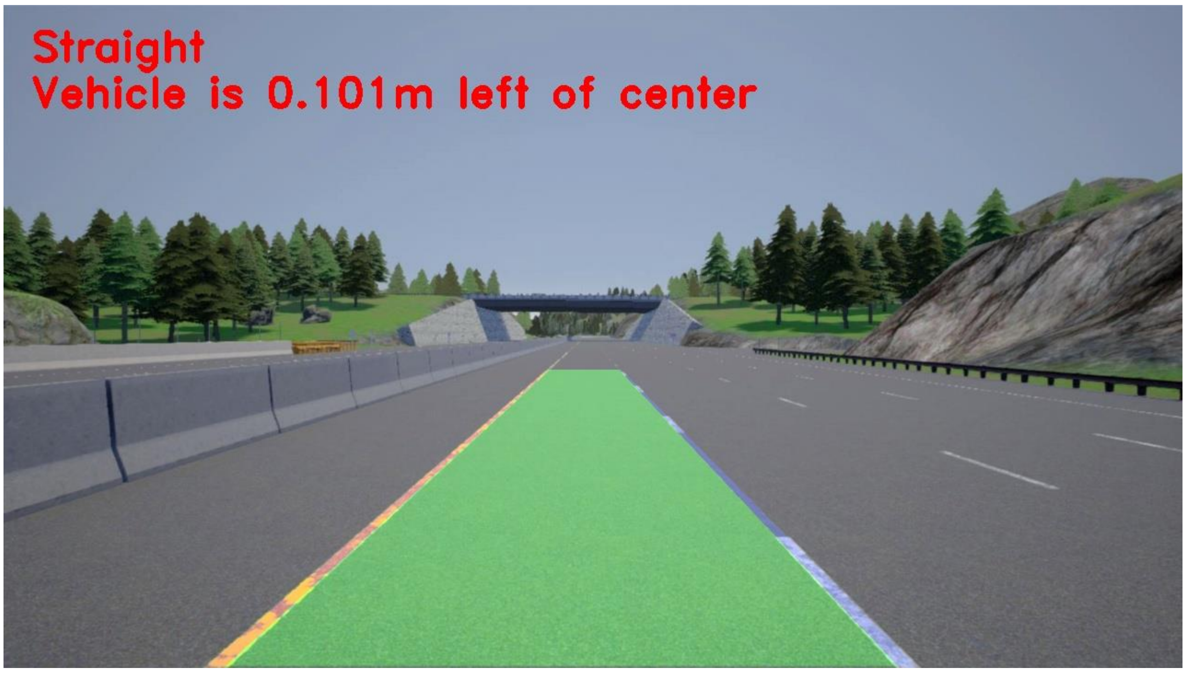


Figure 6.18 - Final output image

7. Validation of Lane Detection Algorithms and Comparison

A task of fundamental importance for self-driving cars is the identification of lane lines to ensure that the vehicle is subject to certain lane restrictions while driving, to safeguard the safety of people, but also to reduce as much as possible collisions with objects, or other cars in nearby lanes. Once the operating principles of the lane lines detection algorithms are understood, they can be validated. This process is a crucial point since it determines whether the system is reliable and robust in meeting the safety requirements.
In the two previous chapters, two algorithms for identifying lane lines were presented. In this chapter, instead, their performance and results will be analyzed through a comparison between the two, highlighting how one (SLLD) is simpler and faster, but less effective, while the other (ALLD) is more complicated and more slow, but much more robust and efficient. Specifically, the algorithms will be examined and tested in different circumstances within the previously generated CARLA simulator scenario in order to understand how different factors such as weather conditions and lighting in CARLA simulator affect system performance.
As also mentioned previously, the dataset used to perform the validation procedure was obtained and developed in the CARLA simulator through the python API ADAS_scenario.py, using an RGB camera placed in the central front part of the vehicle slightly inclined towards the road, in such a way as to faithfully follow the typical position used by this device for image acquisition in the automotive field.
For this purpose a road of the main Town04 freeway has been used as a reference, as it has peculiarities that satisfy the system requirements of both the algorithms implemented for the detection of the lane lines, i.e. flat road with lanes present and visible. It has been also decided to use lane lines with different colors and with a continuous line (the yellow left one) and a discontinuous line (the white right one) with no traffic to be able to effectively and reliably evaluate the performance of the two lane detection systems.
In order to correctly evaluate the performance of the algorithms under different lighting conditions, from ADAS_scenario.py snapshots were taken for each different time of the day: morning, sunset and night. In addition to this data set, other images based on the different weather conditions available in CARLA simulator (by pressing on the keyboard the key C in ADAS_scenario.py) have also been considered, which represent a "critical" environment for the effectiveness of the algorithms.
The behaviors of the two lane detection algorithms with different driving scenarios will be applied and analyzed. Their performances has been simulated and tested on the same images (in HD resolution).

7.1 Light Conditions

The first factor that can be analyzed for each type of image that faces a visual process lies in ambient lighting. Sunlight is represented by the electromagnetic radiation emitted by the sun, which intensifies or weakens depending on the time of day.
To test the sensitivity level with respect to the different light variations of the lane detection algorithms, photos were taken for three different types of ambient lighting: morning, afternoon (sunset) and night.

Morning

In this first part, the moment when there is the best possible lighting (in the morning), when the sky is clear and bright and where there are no shadows caused by the sun on the asphalt will be considered. Figure 7.1 shows the input of the original image, where, given the quality of the image, details such as horizontal and vertical road signs as well as jersey barriers and trees can be accurately distinguished.
After properly setting the threshold parameters for both algorithms, it is possible to view the result obtained by both in the Figure 7.2. The codes have been developed in such a way as to show and save of all the intermediate steps, of which for practical reasons only the final output will be shown. The absence of imperfections of any kind makes the final result of both algorithms very satisfactory and, especially for the ALLD system, shows that the curvature and offset information of the vehicle are correct.

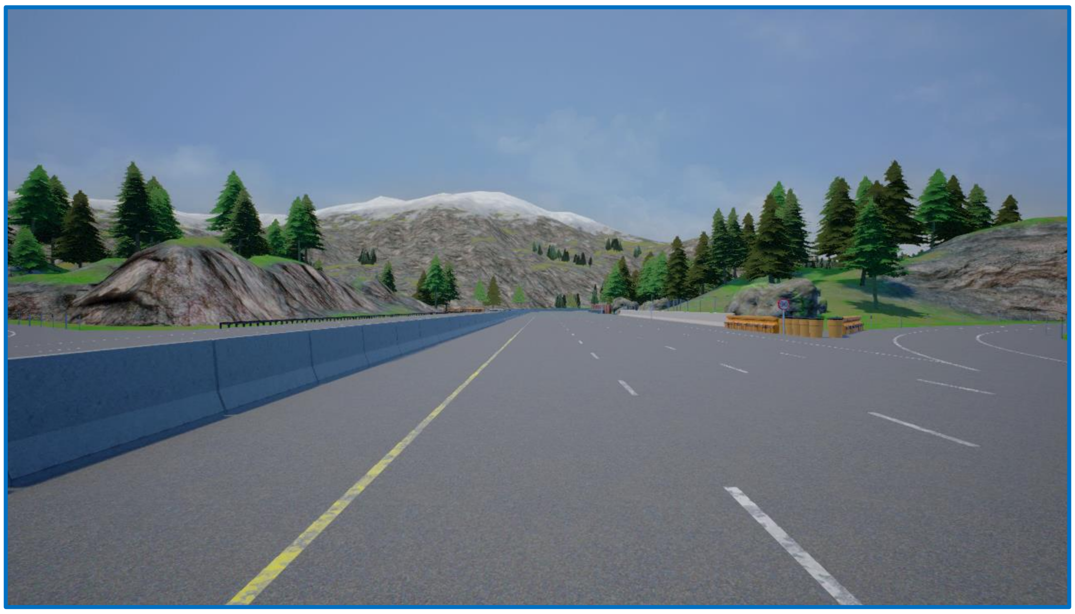


Figure 7.1 - Input image during the morning

SLLD **ALLD**

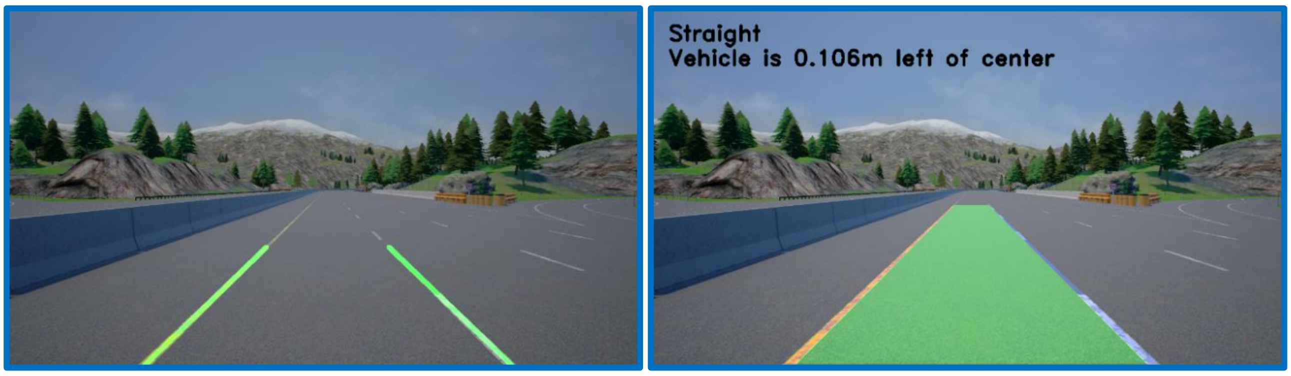


Figure 7.2 – Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning

Sunset

During sunset and the onset of darkness, ambient lighting visibly decreases with worsening visibility. In the Figure 7.3, it is possible to notice the presence of dark shadows that stand out on the asphalt with different intensity and darkness, characterizing that particular moment of the day that precedes the night. The output in the Figure 7.4 shows how even in the presence of a bit of darkness the two algorithms are able to adequately detect the lane lines thanks also to the fact that the contrast between the objects is increased.
Finally, to compensate for poor lane illumination due to deteriorating lighting conditions and maintain acceptable performance, the sensitivity of the two systems has been increased by varying (reducing) the threshold parameters. It is noteworthy how, even in the presence of shadows, the ALLD algorithm shows even better results than in the morning thanks to the combination between gradient and color thresholds.

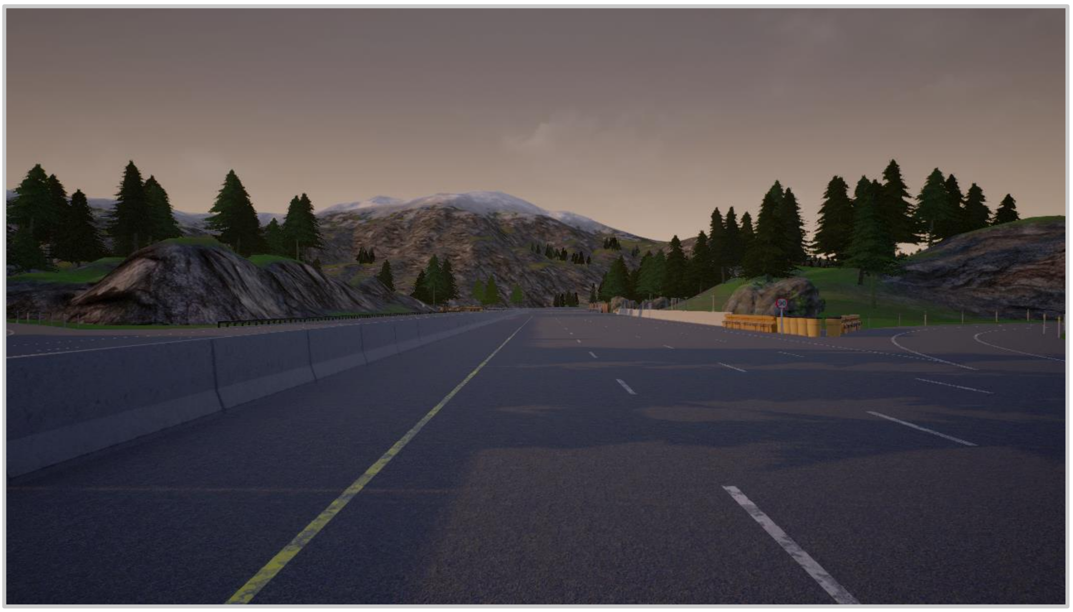


Figure 7.3 - Input image during the sunset

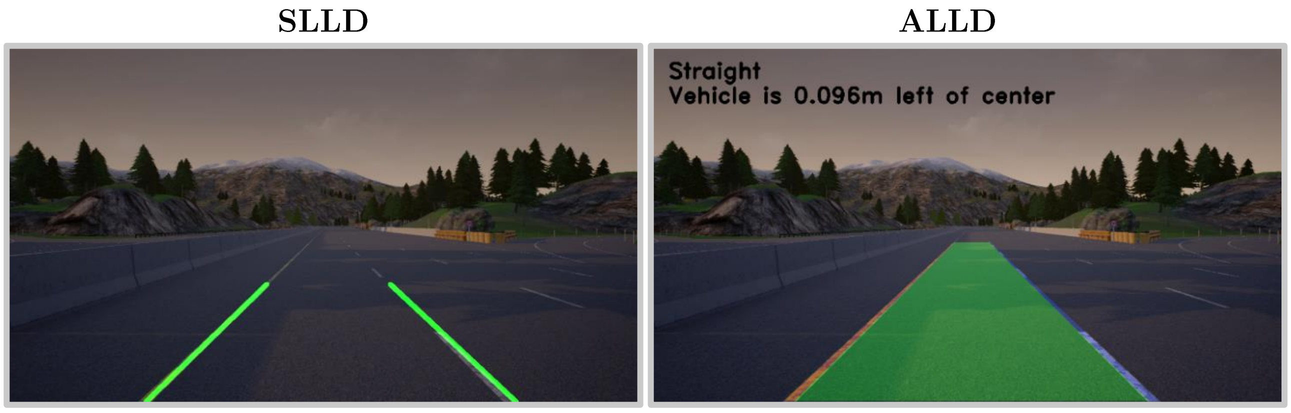


Figure 7.4 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the sunset

Night

Finally, in relation to the light conditions the worst moment of the day has been considered: the night. As can be seen from the Figure 7.5, it is clear that there are visibility problems due to the total absence of both environmental and artificial light (there is only the presence of car's lights). Despite this, the results obtained in the Figure 7.6 show how, further increasing the level of sensitivity to obtain acceptable performances with a consequent increase in noise, the lane lines have been identified with good approximation, especially the continuous one on the left.

What can be easily noticed is how the reduction of the illumination generates a slight worsening of the performance of the SLLD algorithm. This is mainly due to the fact that with darkness, the lane lines become less evident and therefore the algorithm finds it more difficult to detect them. In fact, this cannot accurately detect the white dotted line on the right resulting in a slight shift from where it actually is.

On the other hand, instead, the ALLD algorithm behaves very well even in precarious light conditions, showing to be more robust than the previous algorithm. In general, it can be said that the car’s lights have improved lane detection.

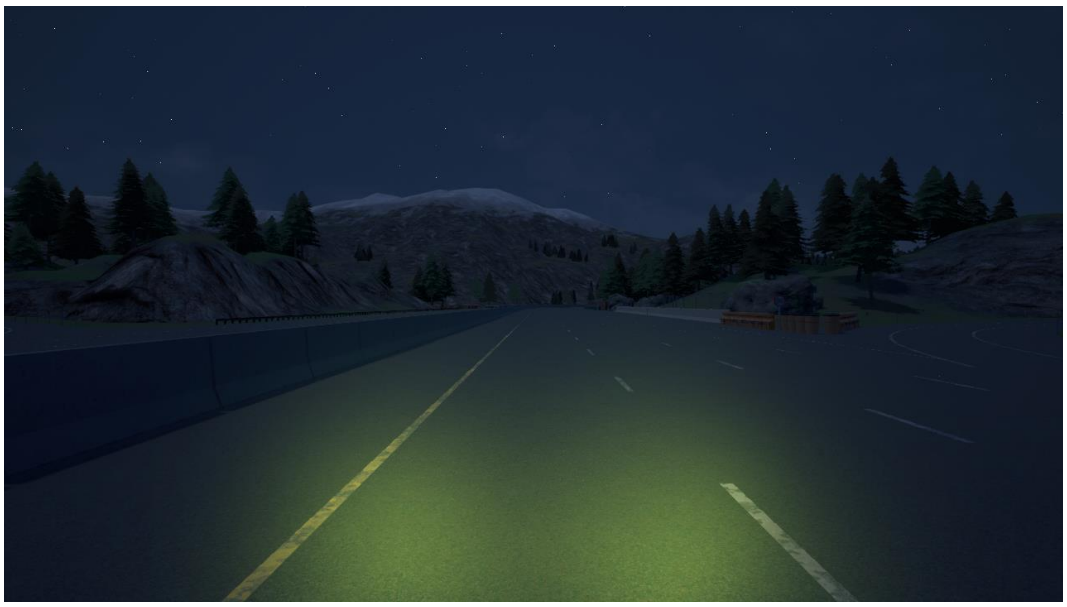


Figure 7.5 - Input image during the night

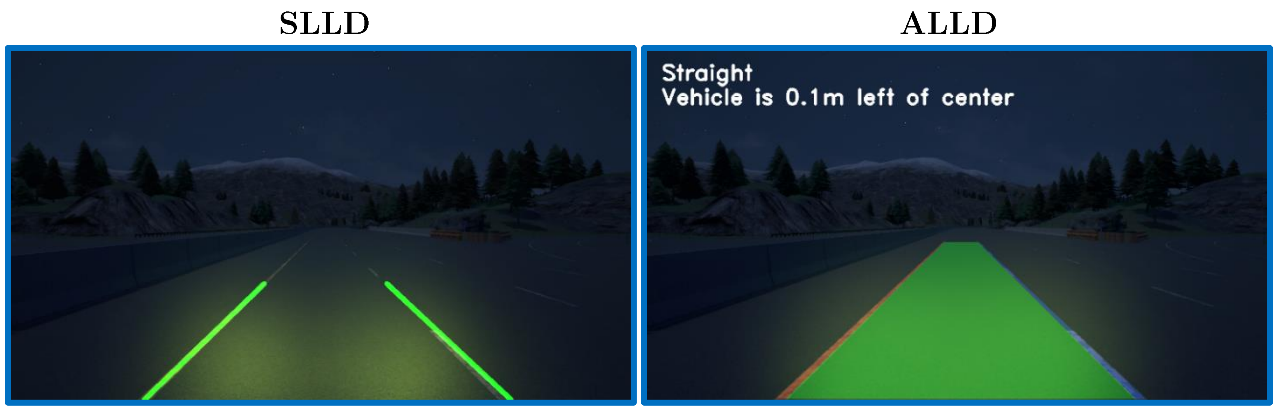


Figure 7.6 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the night

7.2 Weather conditions

In this section, the algorithms will be validated according to the different weather conditions: soft rain, fog and heavy rain. The performance of the algorithms will be related to the different levels of visibility.

Fog

The first condition to be addressed in this first part is the fog, which is very dangerous because it greatly reduces visibility causing serious accidents. Based on its density, the fog can reduce visibility from a few meters to a few centimeters, and for this reason, it represents one of the worst weather conditions for the driver, as it must constantly maintain high attention. The Figure 7.7 shows how even during the day the fog is very disabling due to poor visibility. For this reason, the performance of the algorithms is influenced especially for the SLLD algorithm, which cannot correctly detect the lane lines even increasing the sensitivity to the maximum.

For what concern the ALLD algorithm, instead, by using a high sensitivity by modifying the thresholds appropriately, a good detection is obtained even if the dotted lines on the right have put the effectiveness to the test.
In this situation, the worst results emerged using the SLLD algorithm, so in Figure 7.8, it is possible to observe the ALLD's output result, which proves, even if with some uncertainty, to be more robust than the previous one despite the reduced visibility caused by the fog.

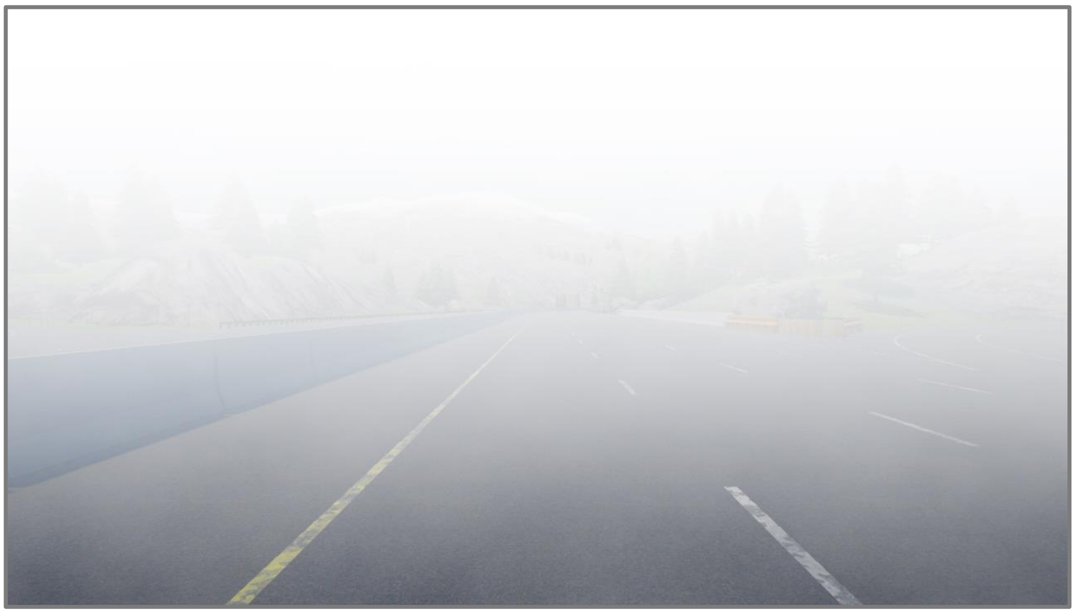


Figure 7.7 - Input image during the morning with fog

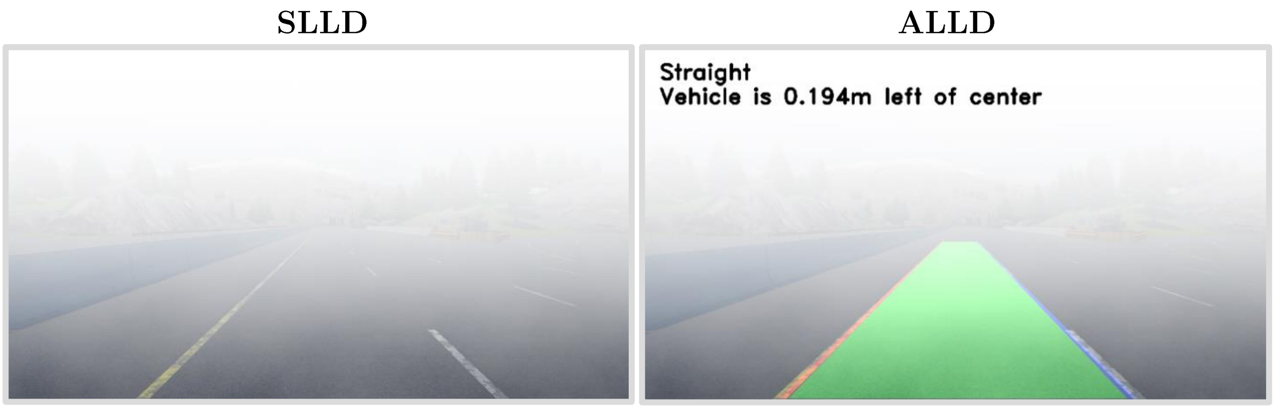


Figure 7.8 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning with fog

Rain

As the second candidate of this analysis, the most common atmospheric precipitation will be considered, namely rain with two different intensities: soft and heavy. This, in addition to making the asphalt more slippery with the possibility of making the vehicles skid without a suitable tread, also affects visibility. Especially when heavy rain is present as it could cover the vehicle's camera in addition to the windscreen, distorting the captured image (Figure 7.9 and Figure 7.11).
Test results have shown that rain is the number one enemy for lane detection algorithms, as it causes a decrease in their effectiveness and performance. As in the previous case, especially in case of heavy rains, the SLLD algorithm cannot detect both lane lines due to interference with raindrops. ALLD, on the other hand, is able to detect lane lines even if with some inaccuracy, especially with regard to data on the radius of curvature. In addition, it should be noted that raindrops cause incorrect detection because they interfere with the image when detecting edges, producing vertical lines that alter the final result.
For this purpose, the sensitivity has been reduced in order to try to eliminate the lines generated by the rain drops that interfere with the original image. The results can be visible in Figure 7.10 and Figure 7.12.

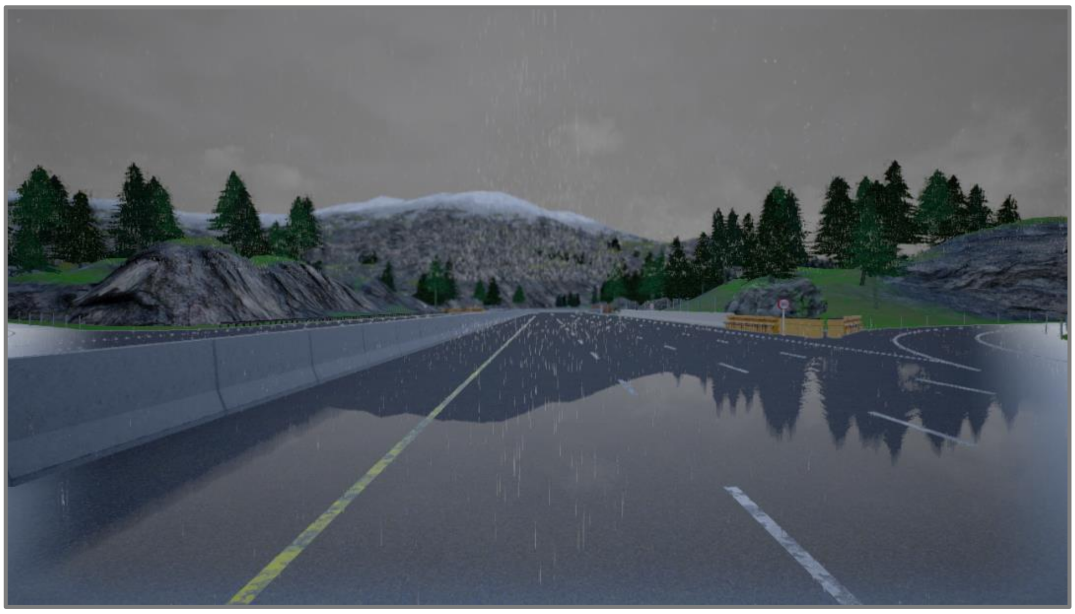


Figure 7.9 - Input image during the morning with soft rain

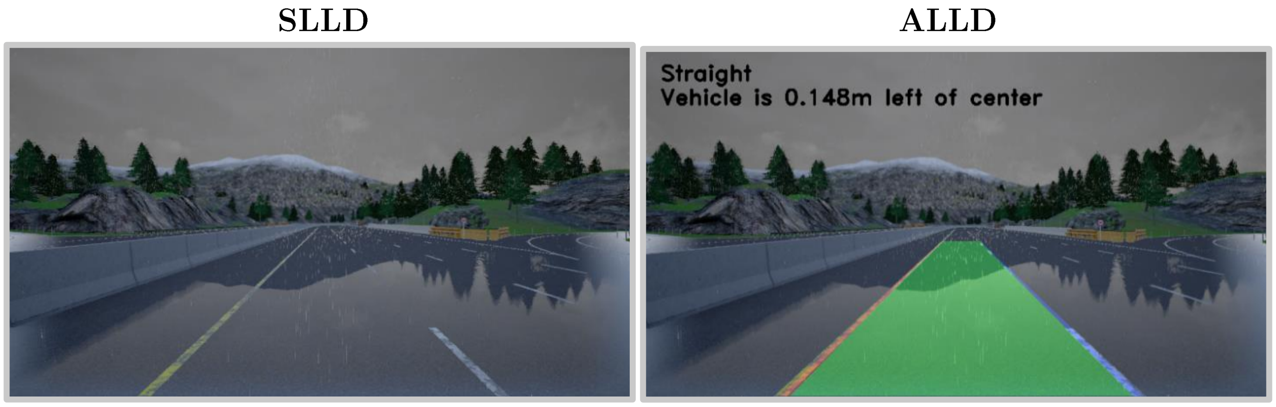


Figure 7.10 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning with soft rain

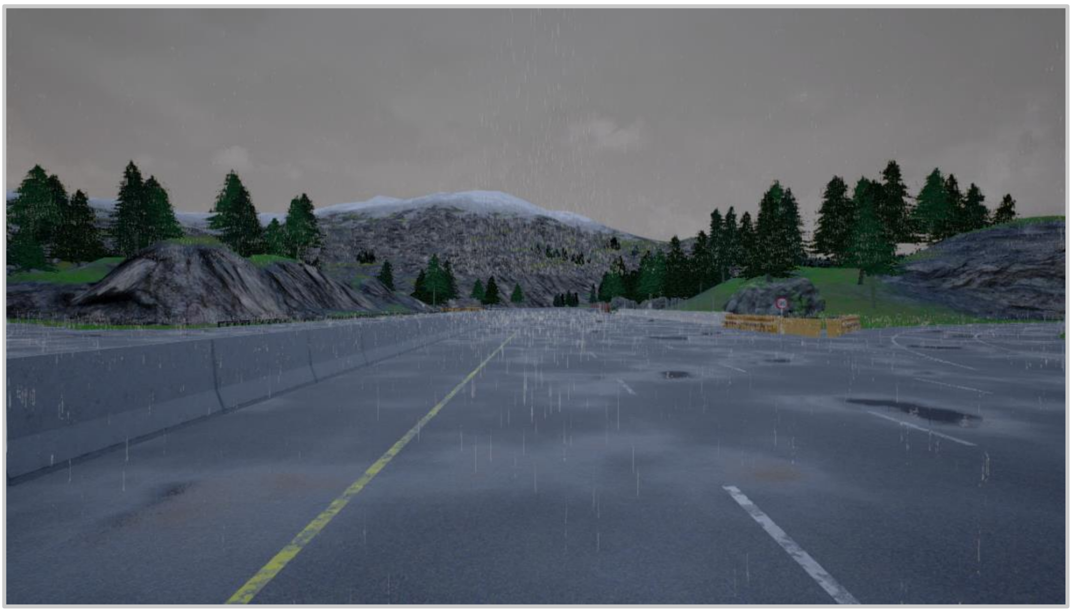


Figure 7.11 - Input image during the morning with heavy rain

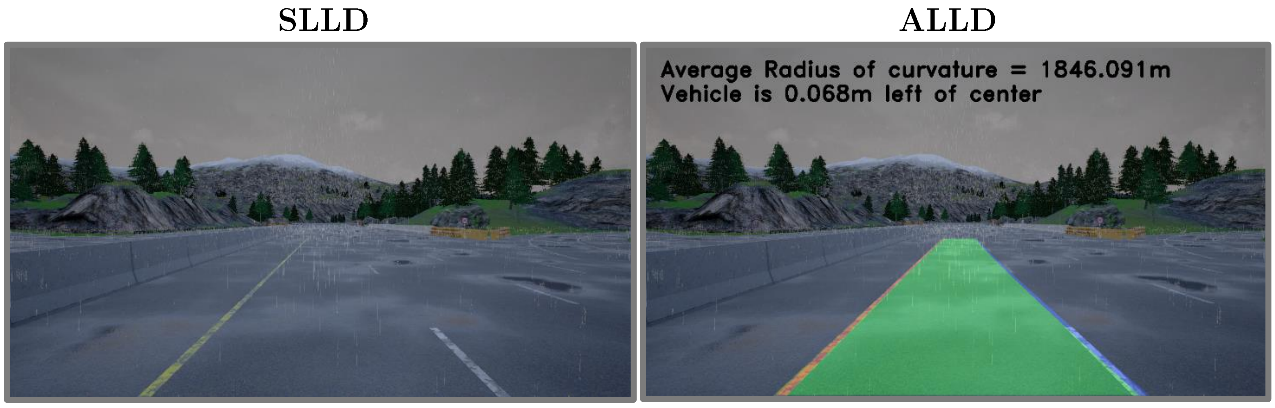

SLLD
ALLD
Average Radius of curvature = 1846.091m
Vehicle is 0.068m left of center

Figure 7.12 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning with heavy rain

7.3 Final Consideration

In conclusion, what is evident, looking at the results, is certainly the fact that the second algorithm (ALLD) offers better performance in terms of identifying lane lines in different weather and light conditions, even when these lines are hardly visible on the road surface or when they are disturbed by external agents.
In particular, unlike the SLLD algorithm, the ALLD algorithm succeeds in indentifying them correctly even when these are curves and the vehicle travels at high speed, helping to provide useful information to the driver both on the radius of curvature and on the offset of the vehicle with respect to the lane. To see the results of the two algorithms on curved lane lines see the Figure 7.13 for the input image and Figure 7.14 for the output images.

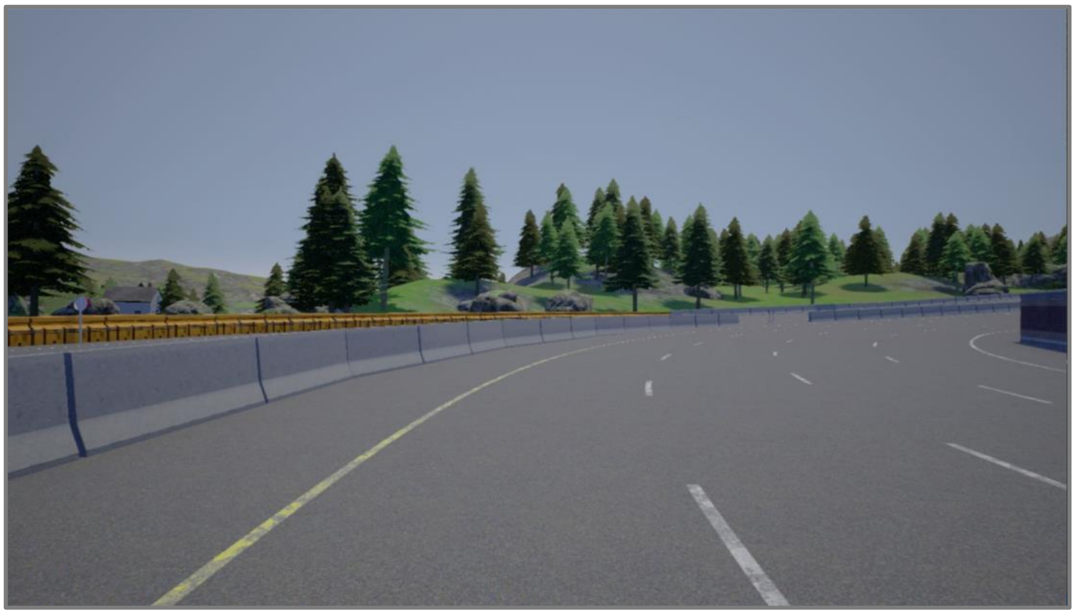


Figure 7.13 - Input image during the morning with curved lines

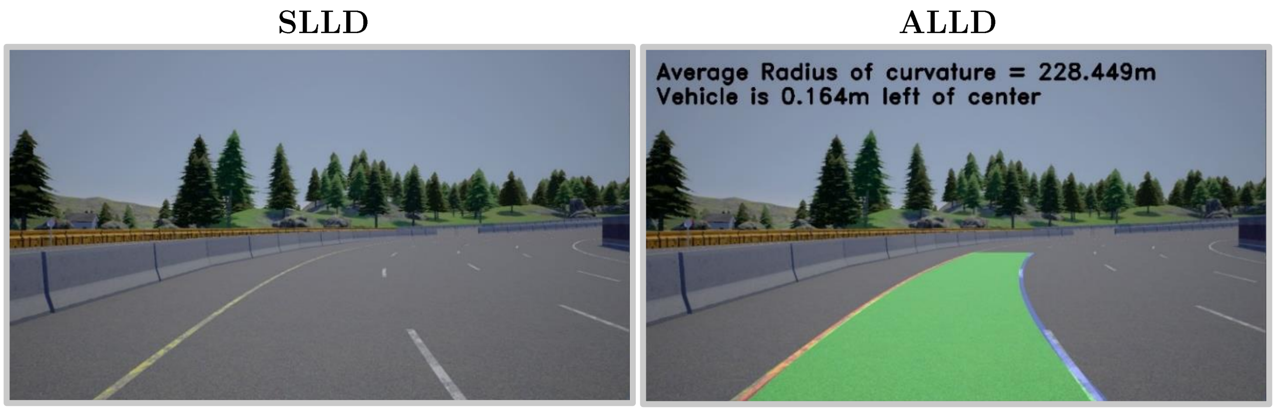

SLLD
ALLD
Average Radius of curvature = 228.449m
Vehicle is 0.164m left of center

Figure 7.14 - Output image of SLLD algorithm on the left and ALLD algorithm on the right during the morning with curved lines

Furthermore, as can be seen from the various tests performed, the ALLD allow not only to identify the lane lines with greater robustness, even in less than optimal conditions or even in the presence of dashed lane lines with different colors, but also allow to accurately determine both the curvature to which the vehicle is subjected and to estimate its position with respect to the middle of the lane. Analyzing the final output on different scenarios, it is evident how the lane lines detection algorithms have advantages and disadvantages, which are summarised in the tables below (Table 7.1 and Table 7.2).

SLLD

ADVANTAGES	DISADVANTAGES
Simple and fast algorithm to identify straight lane lines	The Hough transformation method does not work correctly for curved lanes or sharp turns. Moreover, the presence of road signs on the surface of the asphalt of the lanes, such as arrow or stop signs, can confuse the lane detection algorithm
The Hough transformation works adequately on straight lanes based on continuous straight lines	The mask's parameters cannot dynamically adapt to different types of road environments. There are no universal parameters that can be used, therefore, in order to obtain the correct parameters, several algorithm changes are required that depend on each particular situation to be analyzed (such as canny threshold values, ROI vertices and Hough transform parameters)
Low computational effort	The ROI formation points set out assume that the camera in front of the vehicle remains stable in the same position and that the lanes are flat
With appropriate hardware it can be used for real time applications	Lane detection is not very effective against lanes that have dashed or barely visible lane lines. It does not work on non-lane roads, ie not marked with road signs
	The algorithm as a whole is very sensitive to road visibility and it is not effective in the presence of adverse weather conditions such as fog or rain

Table 7.1 - Pros and Cons of SLLD (Simple Lane Line Detection) algorithm

ALLD

ADVANTAGES	DISADVANTAGES
The algorithm is able to accurately detect lane lines, even in the presence of a curved lane	High computational effort
Accurate determination of both the curvature to which the vehicle is subjected using the polynomial adaptation and the position and direction of the vehicle with respect to the center of the lane using the camera and road parameters	To obtain clear and visible lanes in the binary image, the threshold parameters must be optimized for each different type of scenario. Moreover, the hyperparameters must necessarily be correctly tuned to get the correct results
The algorithm is able to work both with white and yellow lane lines, continuous on both sides, or with broken lines on one side only, or on both side	The detection of lane lines may not be efficient when there are not enough pixels to identify lane lines. In fact, the almost or total lack of data (pixels) prevents the algorithm from performing any type of polynomial adaptation correctly.
It use a combination of effective approaches (like the combination of gradient and color thresholds) that make the end result more robust against weather conditions and road visibility	As for the previous algorithm, the points to establish the ROI must be set a priori and depend on each situation
It can also be used when the vehicle is travelling at high speed	Without appropriate hardware it cannot be used for real-time applications

Table 7.2 - Advantages and Disadvantages of ALLD (Advanced Lane Line Detection) algorithm

The two algorithms can also be applied to a video, as it can be represented by a series of multiple image frames. However, using the available system hardware, process the video using the algorithms takes a long time, so to test their effectiveness, it is necessary to run them on high-performance machines. On the other hand, in order to analyze the characteristics of each algorithm to measure their reliability, an example of some of the results obtained through a video sample using the hardware described in the final part of chapter 2 are reported below.
For what concern the SLLD, measuring performance using OpenCV with the default Optimization value set to True (**useOptimized()**), lead to an average processing time of 0.12 seconds for each frame. Regarding the videos, the accuracy, which strongly depends on the type of lines that defines the lane, is calculated as follow:

$$accuracy = \frac{Number\ of\ frames\ with\ lines\ detected\ correctly}{Total\ number\ of\ frames} \tag{7.1}$$

The Table 7.3 below shows an example of the analysis of the performances with tuned algorithm parameters. It is based on an HD (1280 x 720) video lasting 39.5 seconds, at 30 frames per seconds, in a scenario where there is a continuous yellow line on left side of the lane and a dashed white line on the right and in the final part there is a slight curve.

Total number of frames	1185
Number of frames with lines detected correctly	889
Accuracy	75.02%
Number of frames with lines detected incorrectly[5]	296
Total processing time	142.2 s

Table 7.3 - Example of SLLD algorithm performance testing

Considering the ALLD algorithm, instead, unlike the previous case, to verify that the detection makes sense, a so-called Sanity Check was performed on the lane lines, i.e. it has been verified that the left and right lane lines have a similar curvature, and that they are approximately parallel. It is possible to understand when the lane lines have not been correctly detected if the sanity check has not been successful. If this occurs, the previously identified frame that passed the test is used.
Using the same video of the previous example (at 30 fps), the processing time of the entire pipeline of the algorithm takes about more than 45 minutes. Hence, processing each frame on average in about 2.3 seconds with an accuracy (applying the formula 7.1) of about 89% (see Table 7.4 below).

Total number of frames	1185
Number of frames with lines detected correctly	1055
Accuracy	89.03%
Number of frames with lines detected incorrectly	130
Total processing time	2725.5 s

Table 7.4 - Example of ALLD algorithm performance testing

Comparing the lane detection algorithms (SLLD and ALLD) and analyzing their respective performance it is clear that the second algorithm addressed (ALLD) uses a time of about 20 times higher than the first algorithm (SLLD) to process each frame. Despite this, the ALLD uses a higher precision in detecting lane lines, also showing important information on the screen, not only when the lines are straight, as only the SLLD does, but also when they are curved or there are adverse weather conditions. On the other hand, the SSLL does not require large computational efforts and cost, but it could be used for simple purposes

[5] i.e. when the algorithm fails to detect left lane line or right lane line or both in a frame

in real-time embedded systems. By contrast, the ALLD has high processing power requirements and only with an adequate and powerful platform is it able to find applications in embedded systems. Future works will be able to verify the real-time capabilities of this last lane line detection algorithm presented in this thesis (ALLD).
In conclusion, the complex of computer vision techniques used for image analysis has proved to be a very strong way to identify lanes along the way. Although certain predefined scenarios have been taken into consideration, this validation procedure has highlighted the efficiency and robustness of the lane lines detection systems including their defects and limits. In general, observing the results, it can be stated that the two algorithms are able to correctly identify lane lines in normal conditions, i.e. without visual disturbances, producing very good outputs, while when there is poor visibility due to factors mentioned above like fog or rain, their performance decreases. This is especially true for the SLLD algorithm, which is unable to detect lines under these conditions, while the ALLD, although with some inaccuracy, still manages to detect lane lines.
A final aspect to be considered, highlighted by the various tests performed, is the fact that in order to obtain the best performance in every type of circumstance, it was necessary to lower or raise the sensitivity level of the threshold parameters. Finally, once the validation procedure has been completed, it is possible to state that based on the level of performance obtained in the different scenarios, the most robust system that can be used safely is definitely the ALLD.

8. Object Detection

The growing importance linked to artificial intelligence in the field of self-guided cars is a matter of interest in the coming years, especially the increase in the performance and efficiency of the algorithms.
Among these algorithms, the object detection has certainly become a computer technology, linked to the artificial vision, which is indispensable nowadays for mainly safety reasons (just think for example of the importance of pedestrians' identification near a vehicle). This method is associated with image processing and is concerned not only to detect objects that belong to a certain class, but also to locate it by a so called "bouding box" within a given input image or video. The following formula is taken into account:

$$detection = classification + localization$$

For several years the recognition and classification of objects present in images or especially in videos has represented a huge problem, this is due to the difficulty for traditional algorithms in recognizing the same object in different positions, angles and in a variety of environments. In practice, a common object such as a chair needs to be classified based on a series of features that make it fit into that specific category or class. We can define a chair as a piece of furniture with four legs and a backrest, however not all the chairs follow these rules, because there may be chairs with only three legs or less, or even chairs with armrests or wheels.

Several ways to detect objects in an image or video exist and these fall into two types of approaches: those based on machine learning (classical approach) or those based on deep learning. For the former, it is necessary to carry out the definition of the features as a preliminary operation and then use a technique to classify these characteristics into a specific class. For the latter, instead, it is not necessary to specify the characteristics of each class as they are able to detect end-to-end objects through the use of convolutional neural networks (CNN). The most famous machine learning algorithms include [38] [39]:

- **Viola – Jones object detection framework:** based on Haar features, which uses a map generated by simple binary classifiers, contains a set of characteristics of a given class. This map is then used to train a SVM (Support Vector Machine) as a classifier. This algorithm is simple and fast, but it is not able to recognize objects that have slight variations to those used for learning.

- **Histogram of oriented gradients (HOG):** is used for edge detection taking into account the gradients' orientations in each area of the image, performing the procedure of "sliding window" Hog. The functions thus calculated are sent to a SVM (Support vector machine) to create classifiers. Although slower than the previous one, it is possible to obtain much more precise results. This algorithm is used to detect pedestrians, faces and many other use cases for object detection in real-world applications.

A common feature for these types of algorithms resides in using a technique called "sliding window", that is a scan area to the entire image by analyzing a portion of the time window. Furthermore, for these algorithms there is still the problem of poor robustness in generalizing every single object and the consequent difficulty in recognizing dense images of objects.
Later, the development of convoluted networks with lot of deep learning models gave a strong push in the development of a method capable of overcoming machine learning problems presented aboveIn fact, implementing convoluted neural networks (CNN) it is possible to use the same "sliding window" process but with different window sizes, so as to obtain for each object identified a prediction associated with a certain degree of confidence (i.e. the probability that object belongs to a certain class). In this way it is possible to detect all the objects present in a given image through the sliding window approach, by sliding a small window (small enough to capture even the smallest objects in the image) on the entire image and verifying the presence of objects inside each of the windows created. Afterwards, the window slide using small strides, to ensure to identify any object and to determine the location of objects. This process is repeated until the entire image is covered. In this thesis the use of convolutional neural networks (CNN) in the recognition of objects has been addressed as it allows to carry out its work with simplicity and effectiveness, even if it nevertheless requires a great deal of computing power to be able to train the network in accurate manner.

8.1 Concepts behind CNN

Convolutional Neural Networks, also called CNN or ConvNets, is a type of class of deep artificial neural networks used especially for pixel data recognition and processing in images. The CNN consists of a series of layers that learn to extract relevant features out of any image through convolutions between the image and weighted kernels, these last ones tuned in the training step.
The peculiarity of a convolutional neural network lies in its design. In fact, this is usually made up of different types of layers that process visual information: in general, an input level, an output level and of multiple layers hidden in the middle of this two. These hidden layers are formed by convolutional layers, pooling layers, normalization layers, fully connected layers and ReLU layers (rectified linear unit representing the activation function). A CNN acts in this way: it takes an input image and then passes it through these levels.

The first layer is the convolutional one and has the task of processing the input image in a direct way. It is constituted by a set of convolutional filters (i.e. grids of weigths) which extract a specific type of function such as detecting the edges of an object. The output of this layer is represented by a set of feature maps (also called activation maps) which are nothing more than filtered variants of the original input image where each filtered image extract certain features. The ReLU layer is usually positioned after a convolutional level in order to transform the non-linear output in such a way as to perform the backpropagation more efficiently, and effectively train the network.
The Pooling layer contain an image (usually filtered) and produces a reduced version of that image. There are different types of pooling, the most common is the Maxpooling that is able to choose and maintain the maximum value of pixels in an area of the input image, in a new area of reduced size. The fully connected layer, which is placed after a series of convolutional and pooling layers, consists of producing an output of the desired shape given the input. Specifically, it converts a matrix of image's characteristics into a vector of features with dimensions 1xC, where C represents the number of classes. This vector compresses the feature maps' information into a single feature vector. Finally, two additional layers that perform important functions can be added to a CNN, these are the softmax function and the dropout layer:

- *The softmax function* is used to transform the output vector of a completely connected layer into a vector of the same length where each number of the resulting vector represents the probability (between 0 and 1) that a given input image object belongs to a particular class. The higher this probability, the higher the object detection confidence. The output produced is also called class score, from which it is possible to extract the most likely class to be able to classify the image.

- *The dropout layer*, instead, is used to prevent overfitting, reducing the probability that only a few important nodes dominate the process. This essentially disables certain nodes of a layer with a certain probability, to allow all nodes to have the same possibilities of trying to classify different images during the training.

CNNs have a great advantage, namely they are variations of multilayer perceptions designed with a reduced preprocessing compared to other image classification algorithms. The network in fact behaves just like the human brain, applying and learning a series of different image filters, also known as convolutional kernels, to an input image. The size of each kernel for each level must be set a priori. For images in RGB the kernel will be h*w*d, where h = Height, w = Width, d = Dimension (d = 3 channels in this case).
The Stride controls the amount of kernel's shift in which the filter convolves around the input volume. The resulting filtered images have different appearances. Filters can extract features such as edges of objects in an image or colors that distinguish different classes of

images. When the CNN trains, it is able to update the weights that define the image filters in its convolutional layer through backpropagation (usually the final convolution, used to get more precision). The final product is a classifier with certain convolutional levels that have learned to filter images to extract specific characteristics. The Figure 8.1 below shows a classic complete CNN model to process an input image and classify the objects inside it.

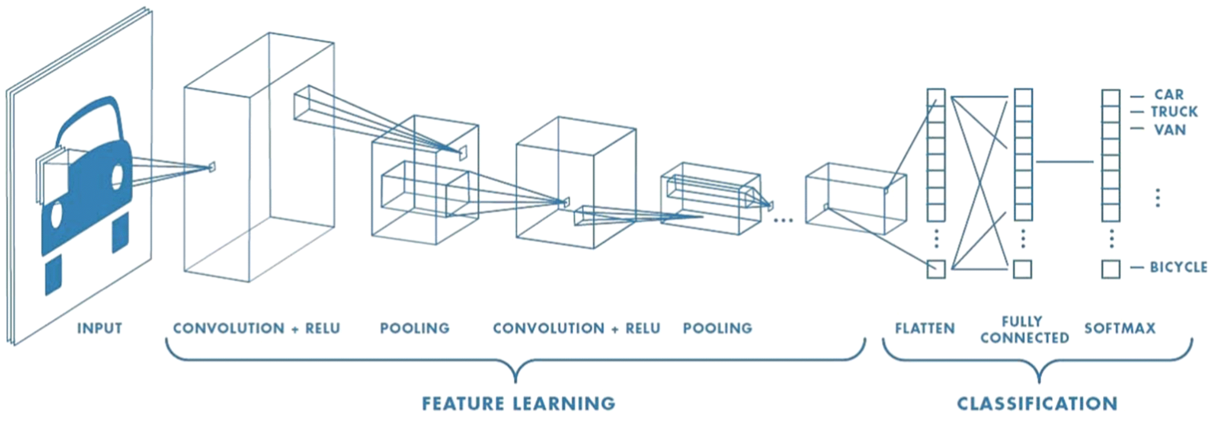


Figure 8.1 - CNN architecture with layers [40]

8.2 Deep Learning approach

There are various techniques for object detection using the Deep Learning approach, which unlike classical approaches use a more accurate classifier based on convolutional neural networks. Among these certainly of great application are [41] [42]:

- **Region-based Convolutional Neural Networks (R-CNN):** Extracts probable objects using a region proposal method called Selective Search, utilized to reduce the number of bounding boxes (up to about 2000) that feed the CNN classifier that is able to extract features from each region. This in turn is followed by SVM to predict the class to which each object belongs in each region. This algorithm is able to achieve excellent results, but not without flaws. This is due to training: to train the network it is necessary to classify 2,000 regional proposals per image. It cannot therefore be implemented in real time since it takes about 50 seconds for each test image. Furthermore, the selective search algorithm is a fixed algorithm (i.e. it does not involve any learning), which means that it could generate proposals for regions of incorrect candidates. For what concerns the performances in terms of mean Average Precision (mAP), this method has a mAP of 59%.

- **Fast R-CNN:** This approach is the evolution of the previous one as it uses a purer deep learning: it makes the training end-to-end possible and adds the regression of the bounding box to the training of the neural network itself. In a similar way to R-CNN, it uses selective search to generate object proposals by extracting their features using SVM classifiers not independently, but applying a CNN on the entire image to generate a convolutional feature map using both the Region of Interest (RoI) Pooling with a feed forward network for classification and regression. These two important changes reduce the overall training time and increase accuracy as they allow the system to be completely differentiable and therefore easier to train. Fast R-CNN shows significant improvements not only for the time-to-process an image, but also for what concerns training time and mAP. However, regional proposals represent the bottleneck in the Fast R-CNN algorithm as they affect its performance.

- **Faster R-CNN** [43]**:** This type of algorithm replaces the selective search, used to discover the regional proposals that had characterized both the previous algorithms, with a very small convolutional network called Region Proposal Network to generate regions of Interests. This is due to the fact that using selective search significantly affects network performance, since it is not only a slow process but also time-consuming. With this new algorithm the convolutional network receives an image as input and provides a convolutional functionality map using a separate network to predict the region proposals which are in turn reshaped using a RoI pooling to classify the image within the proposed region. This results in a greater speed so as to be also used in real-time object detection applications. Faster-RCNN is about 10 times faster than Fast-RCNN with comparable accuracy of datasets.

Architectures like faster R-CNN are accurate, but the model itself is quite complex and even though trained it is still not fast enough to run in real time.
Although the sliding window approach improves the bounding box and splits an image into a grid in order to perform an efficient object detection, it is very expensive from a computational point of view as it requires to scan the entire image with windows of different sizes that must be included in CNN. The CNN's outputs are analyzed separately in the network trains by using a weighted combination of classification and regression losses.
The methods discussed above have the common denominator of managing the detection of objects in an image as a classification problem. However, there are some algorithms that deal with object detection as a regression problem.
While the previous approaches based on the network of regional proposals (R-CNN and all its variants) need two stages (a first shot to generate regional proposals and a second shot to detect objects present in them), regression-based approaches are able to detect a multitude of objects within an image with a single shot. For this reason, it is easy to

understand how these approaches are much faster than the previous ones. The two most famous are SSD and YOLO.

Single Shot Detector (SSD) [44]**:** This algorithm offers a good trade-off between speed and accuracy. In fact, it manages a convolutional network on the input image only once and calculates a feature map. Subsequently, a convolutional kernel is applied to predict the bounding boxes and classification probability using anchor boxes (a rectangle of predefined proportions) at various aspect ratios. This method is able to achieve about 77% of mAP.

You-Only-Look-Once (YOLO) [45]**:** this algorithm has both excellent results and high speed, allowing the detection of objects in real time. In order to perform the detection, YOLO divides each input image into an SxS grid where each cell will provide N possible bounding boxes with the confidence value (or probability) for each of them. This means that SxSxN boxes are calculated. YOLO is responsible for detection by applying a tensor on characteristic maps (Figure 8.2) in the form:

$$\mathrm{S\ x\ S\ x\ [B} * (4 + 1\ +\ \mathrm{C})]$$

Where S x S represents the grid dimension, B represents the number of bounding boxes predicted from a cell on the feature map, (4+1) represents the 4 bounding box offsets and 1 objectness prediction, and C represents the number of class predictions. Each bounding box is represented by 5 predictions: x, y, w, h and a parameter that represents the probability of belonging to a certain class. The x and y represent the coordinates of the center of the box, w and h represent the width and height of the bounding box respectively, while the confidence score can be calculated as:

$$\Pr(Object) * IoU_{pred}^{truth} \tag{8.1}$$

Where the first term represents the model's confidence of having an object inside the box (this is zero when a cell has no object), while IoU is the Intersection over Union between the predicted box and the ground truth. Furthermore, each cell of the grid has the probabilities of the conditional class C given by:

$$\Pr(Class_i|Object) \tag{8.2}$$

This quantity represents the confidence of having an object belonging to the i-th class, conditioned by the probability of effectively having an object inside that cell. At test time, to know the class-specific confidence scores for each bounding box, the multiplication between the conditional class probabilities and confidence predictions must be executed [45]:

$$\Pr(Class_i|Object) * \Pr(Object) * IoU_{pred}^{truth} = \Pr(Class_i) * IoU_{pred}^{truth} \tag{8.3}$$

During the training phase, YOLO predicts several bounding boxes for each cell of the grid, but at the same time, through a loss function, it selects only one bounding box for each object prediction. The confidence value (equation 8.3) reflects the accuracy that an object in the selection box belongs to that particular class. The bounding boxes, with a class probability that is higher than a certain threshold value, are selected and used to identify the object within the image, while the others, which are below the minimum probability threshold, are discarded. The remaining boxes will undergo a further "non-maximum suppression" step, which is responsible for eliminating duplicate objects, leaving only the most correct one.

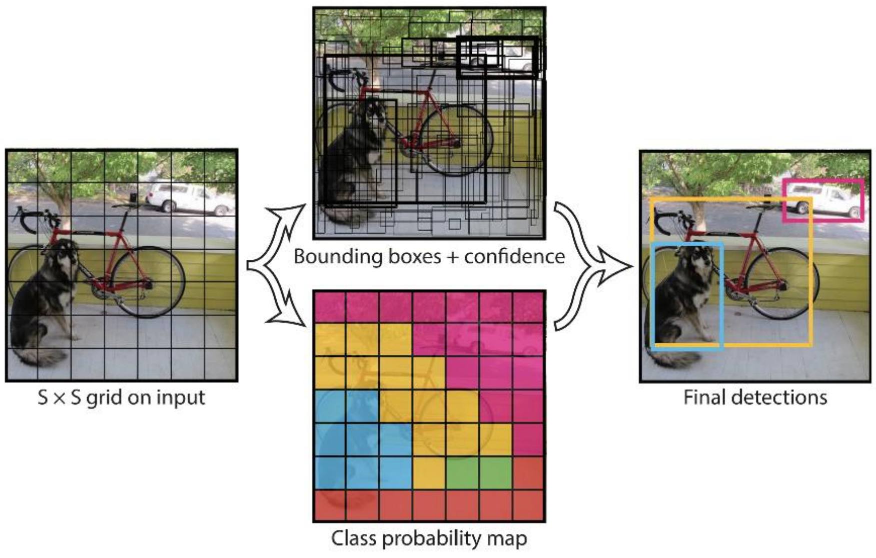


Figure 8.2 - YOLO model [45]

YOLO has a detection network of 24 convolutional levels, which are used to reduce the functional space from the previous levels, followed by 2 fully connected levels, used to perform forecasts (see Figure 8.3). A fast version of YOLO has been designed and implemented in order to detect objects even faster. This architecture, despite being faster and smaller than the previous one (it uses 9 convolutional levels instead of 24), has a lower precision.

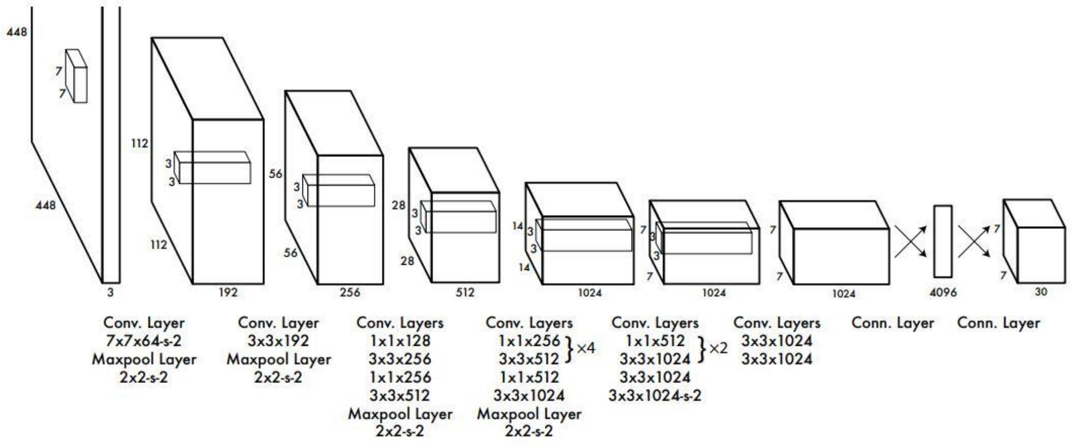


Figure 8.3 - Full YOLO network

The biggest YOLO's flaw is that it has strong spatial constraints that establish only certain bounding boxes and classes for each cell type. In this sense, the number of near and small objects that can be predict is limited (flocks of birds or groups of small objects). Performance results show that YOLO is able to detect objects at 45 frames per second with a precision (mAP: mean average precision) of 63.4%. Fast Yolo, on the other hand, is able to perform the detection even at 155 FPS but with a reduced precision of 52.7% mAP.

8.3 YOLOv3 model

YOLO, which stands for You-Only-Look-Once, is one of the fastest high-performance object detection algorithms at the expense of not much loss of precision used for real-time object detection. The latest version of YOLO, v3 (2018), significantly larger than previous models, uses [46] as architecture a variant of Darknet-53 (a neural network framework), which with its network trained on Imagenet (an image database), uses in total 106 layer fully convolutional for the detection of objects (Figure 8.4).

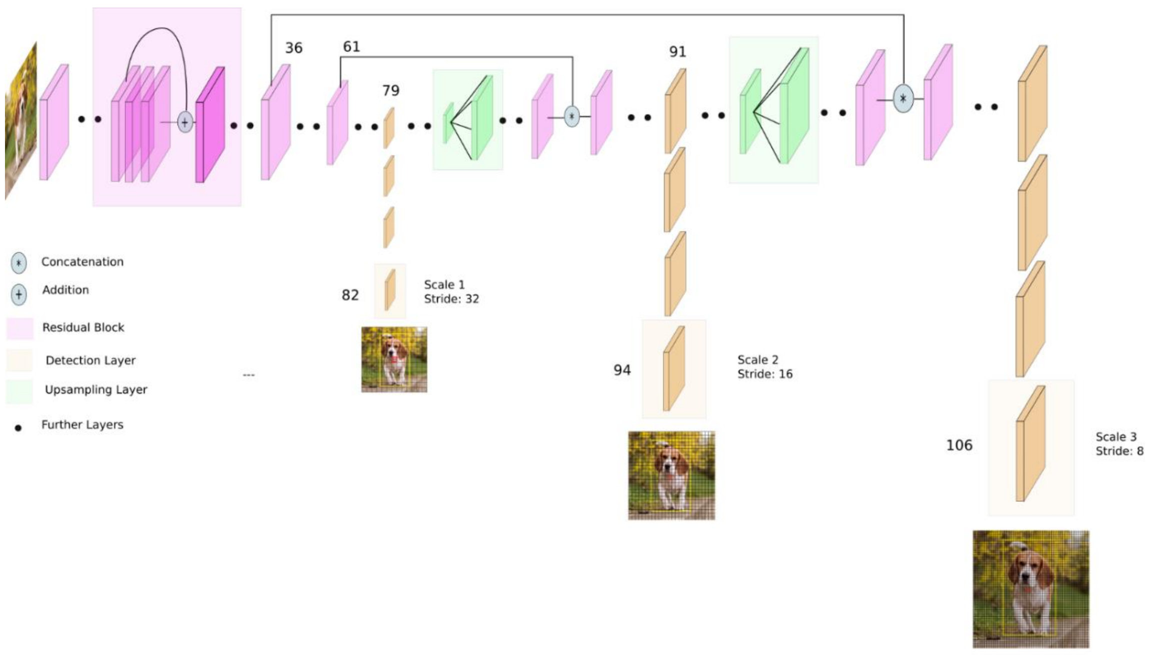


Figure 8.4 - YOLOv3 architecture [47]

In YOLOv3, the substantial difference from previous versions is that the 2 fully linked levels used for the prediction are removed and the forecasts are made using anchor boxes. In YOLOv3, the network predicts 4 coordinates for each bounding box: t_x, t_y, t_w, t_h. If the cell is offset from the top left corner of the image of (c_x, c_y) and the previous bounding box has a width pw and a height ph, then the forecast is equal to [46]:

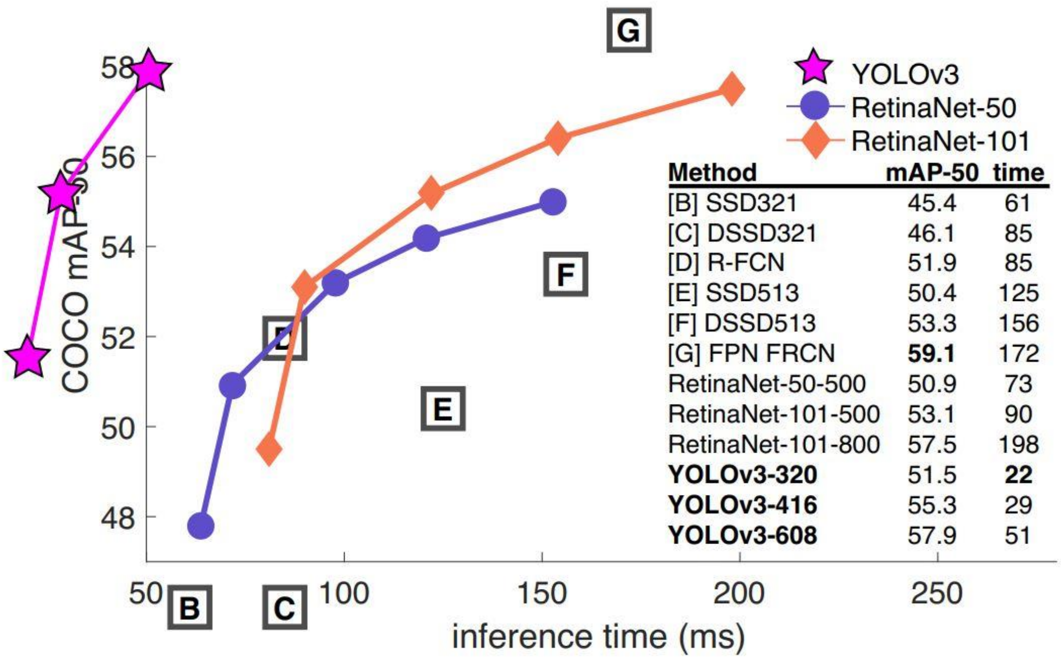
$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma(t_y) + c_y \\ b_w &= p_w e^{t_w} \\ b_h &= p_h e^{t_h} \end{aligned} \tag{8.4}$$

Where σ represents a sigmoid function, which is used to predict the center coordinates of the box. During training the sum of the squared error loss is used and the gradient, which is the difference between the ground truth value and the prediction, is computed:

$$\widehat{t_*} - t_* \tag{8.5}$$

The most important feature of YOLOv3 is to carry out surveys and predictions on bounding boxes at three different scales. These scales (stride) resize the dimensions of the input image by 32, 16 and 8 respectively; using nine anchor boxes (every grid can predict up to 3 boxes through 3 anchors for each scale). This happens because the detections on different layers

(multi-scales predictions) help to obtain greater accuracy even with small objects (a relevant problem in the previous YOLO versions). The ability to detect small objects in front of a vehicle represents a field of fundamental importance for self-driving cars because detection helps provide the time needed to react and avoid the obstacle.
In YOLOv3 each bounding box are predicted using logistic regression with a confidence score. YOLOv3 is much more accurate than previous versions, and despite being a bit slower, it is still one of the fastest algorithms. In Figure 8.5 below, a clear comparison of the performances between the most powerful object detection algorithms is provided.



Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

Figure 8.5 - Performance comparison between different object detection algorithms using COCO dataset [46]

As it can be easily seen, YOLOv3 works in an objectively more efficient way in terms of speed than other detection methods even if with a smaller mean average precision (mAP). Since YOLOv3 presents the best trade off in terms of speed and precision, the next chapter will be focused on an algorithm to implement with the OpenCV library, YOLOv3, which will be later validated in a virtual environment by means of CARLA simulator.

8.4 YOLOv3 Object Detection Algorithm

In this section, an algorithm has been developed both in python and in C++ able to apply YOLOv3 with COCO dataset using the OpenCV library on images and videos. The COCO dataset includes 80 of the most common classes. The YOLOv3 algorithm, as explained in the previous sub-chapter, predict objects, framing them by means of a bounding box to which a confidence score is associated. In addition to the input image/video, two optional argument of confidence threshold parameters can be passed.
The first, "*conf_threshold*", is bound to the confidence score of each bounding box, and in particular, if this is below the confidence threshold parameter it is discarded, otherwise accepted; whereas the second, "*nms_threshold*", controls the non-maximum suppression process. Moreover, input width values (*inpWidth*) and height (*inpHeight*) can be set for the network input image. To obtain greater precision it is possible to increase these values to 608 for both parameters, while if the goal to achieve is speed, sacrificing little precision, it is sufficient to set the parameters to 416 or 320 (faster).

After importing the necessary libraries, Class labels of COCO dataset (*coco.names* contains all the objects of the model), on which YOLO was trained, are loaded, setting random colors for each classes and the network is set using the configuration file (*yolov3.cfg*) and the pre-trained weights (*yolov3.weights*) for the model (**readNetFromDarknet()**).
Furthermore, the DNN backend is set to OpenCV (**setPreferableBackend()**) and the target on the CPU (**setPreferableTarget()**), since it is not possible to test the algorithm using the GPU with the current version of OpenCV (it is not supported yet), and therefore even setting the target on a GPU, Intel would automatically switch to the CPU.

In the next step, it is possible to read the input image and get its spatial information. The input image in a neural network, to be processed, must necessarily be in a certain format called BLOB, which stands for Binary Large OBject and refers to a group of connected pixels in a binary image. For this reason the OpenCV function **blobFromImage()** is used, which allows to make this conversion by creating 4-dimensional blob from the image. This process especially resizes the image pixel values to a target range, which span from 0 to 1 using a scale factor of 1/255. It also resizes the image to the indicated size of (inpWidth, inpHeight) evenly, without clippings.

The resulting BLOB is passed to the network as input (**setInput()** OpenCV function) by performing a forwarding step (**forward()** OpenCV function) to obtain the list of predicted bounding boxes as network output. These boxes go through the post-processing phase to filter those with scores below the confidence threshold previously selected (*conf_threshold*).
In this phase, it is possible to calculate the processing time for each frame using **getPerfProfile()** OpenCV function, which returns overall time for inference and timings

(in ticks) for layers and **getTickFrequency()** OpenCV function, which returns the number of ticks per second. In order to get the names of output layers, the forward function in the OpenCV Net class is used. It requires the final layer of the network (this is obtainable using the **getOutputsNames()** function), which provides the names of the unconnected output layers, i.e. the last layers of the network.

As mentioned before, outputs bounding boxes are represented by the vector of number of classes and by five elements: x and y coordinates of the center of the box, w and h, the width and height of the bounding box respectively, and the fifth element represents the probability that the box encloses an object within it.
The box is designated to the class with the highest score (confidence) to which the object in it corresponds. If the confidence value is below the confidence threshold (*conf_threshold*), the bounding box is discarded and no longer considered part of the solution. On the other hand, the boxes with a confidence value equal to or greater than the confidence threshold are accepted and subject to the non-maximum suppression procedure with **NMSBoxes()** OpenCV function (controlled by the *nms_threshold* parameter) to reduce the overlapping boxes until only one is obtained (the one with the highest confidence value).
For the non-maximum suppression parameter value, it is advisable to use an intermediate value between 0 and 1, since if this value is too low it will not be possible to detect overlapping objects, instead if it is too high, more boxes will be obtained to identify the same object. The function that sums up these operations is **post_process().** Finally, the boxes that have been filtered through the non-maximum suppression can be represented on the input frame with the label of the assigned class associated with the corresponding color and the confidence scores (see **draw_box()** function).
For more details, see Yolo.py in Appendix.

9. Validation of YOLOv3 object detection algorithm

The detection of different classes of objects constitutes, for autonomous driving vehicles, a fundamental requirement that allows obtaining a high degree of safety. In the previous chapter, different types of algorithms able to carry out this task also highlighting the performance have been analyzed in detail.
The one that offers the best trade-off between speed and precision is definitely YOLOv3, which allows to detect objects in real time with good accuracy (mAP), indeed, due to safety requirements, the ability to execute algorithms in real time is a must when referring to the automotive field. As previously mentioned, YOLOv3 is used to detect and recognize different classes of objects. In this section, its performances as well as its robustness will be investigated.
In order to validate this real time method, that is one of the best in terms of speed, a suitable approach to neural networks will be followed. More in particular, the whole system will be tested using a set of input images that allow to estimate the confidence score for each object class. A similar approach used before for lane lines detection system will be performed in this validation process, however in this case only a small but more detailed dataset will be considered.
Through Carla simulator and python API ADAS_scenario.py, a generic scenario, about different moment of the day, has been captured with the intention of examine all the potential real operating conditions on the road. Therefore, the resulting dataset will contain images that take into account different effects due to weather and lighting conditions.
In Figure 9.1, the scenario adopted for validation test is illustrated: it is a typical urban environment of Town04 in CARLA Simulator, which is able to recreate a realistic and heterogeneous environment with elements like traffic lights, pedestrians or cars arranged at different distances. As it can be see, it was tried to incorporate as many elements as possible within the scenario to better evaluate the context. The different performances of the YOLOv3 pre-trained model using COCO dataset will then be discussed in relation to light and weather conditions, highlighting the critical points. The parameters used for object detection with YOLOv3 are:

- *(inpWidth, inpHeight)* = (608, 608)
- *conf_threshold = 0.3*
- *nms_threshold = 0.3*

9.1 Light Conditions

The first parameter that will be analyzed are the natural lighting considering its variation in the 3 main moments of the day: morning, afternoon (sunset) and night. Lighting, as with any vision process, is an important factor in analyzing the different types of objects in an image, as it is able to modify the final output.
In fact, greater lighting allows to highlight some of the characteristics of the objects, such as color. On the other hand, the low brightness does not allow to correctly and quickly identify the edges of objects and their characteristics.

Morning

During the morning the ambient lighting conditions are the most favourable, as they manage to highlight every type of object in the image with its main characteristics. Figure 9.2 shows the result of the test carried out. As can be seen by observing the output, the quality of the objects identification results is very good, indeed all objects, even if placed at different distances, have been recognized.
Note that the closest objects, since they appear larger, have a higher detection rate, more than 80%, compared to the objects placed at a greater distance that appear smaller and therefore less recognizable.

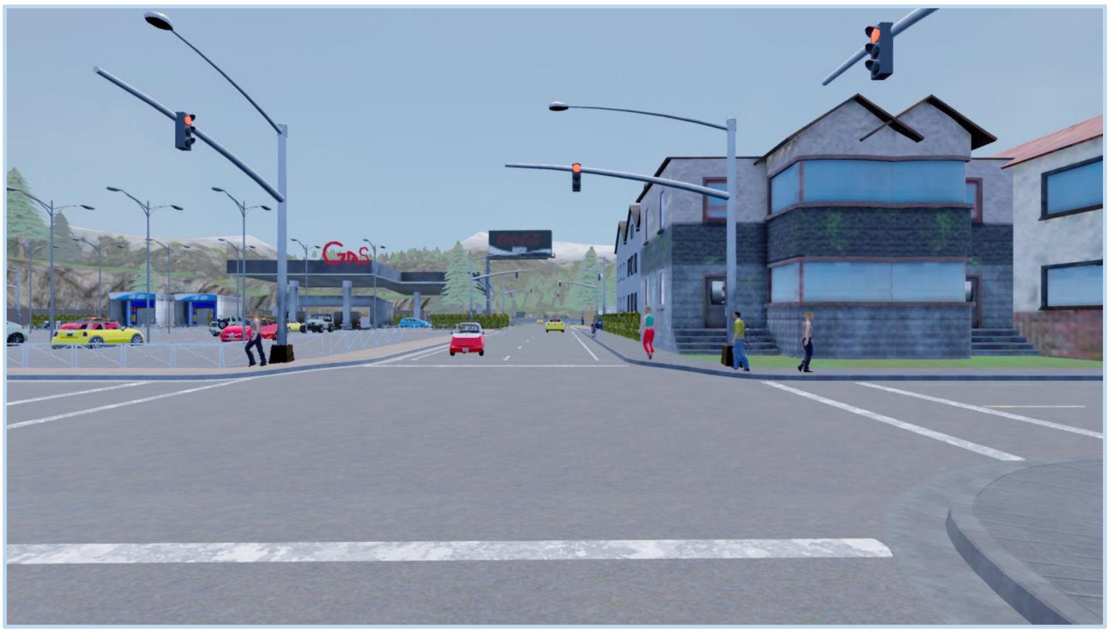


Figure 9.1 - Input image during the morning

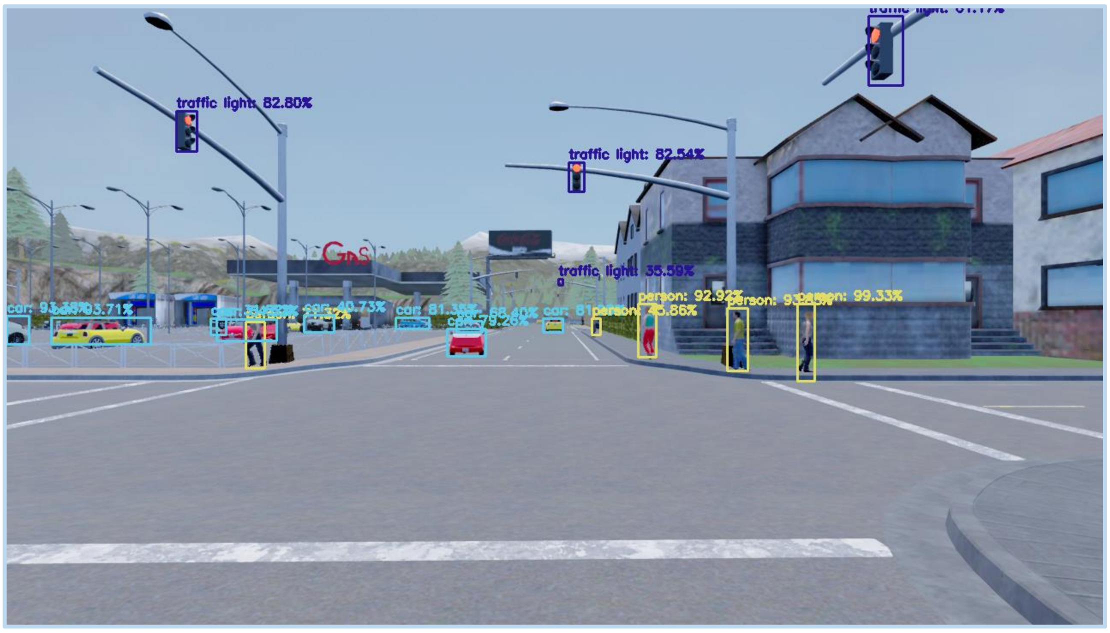


Figure 9.2 - Output image applying YOLOv3 during the morning

Sunset

In this part, on the other hand, the scenario in the late afternoon (at sunset) has been considered, i.e. when the light is lower than in the morning and creates different shadows hitting different objects, generally obscuring the environment (Figure 9.3).
Figure 9.4 shows the results obtained by the algorithm that are comparable to those obtained in the morning or even better (for the traffic lights and pedestrians), because the lower lighting allows to increase the contrast of objects with respect to the asphalt, highlighting the contours. An example is the traffic light at the top left with a percentage higher than 90% compared to the percentage of about 82% in the previous case.

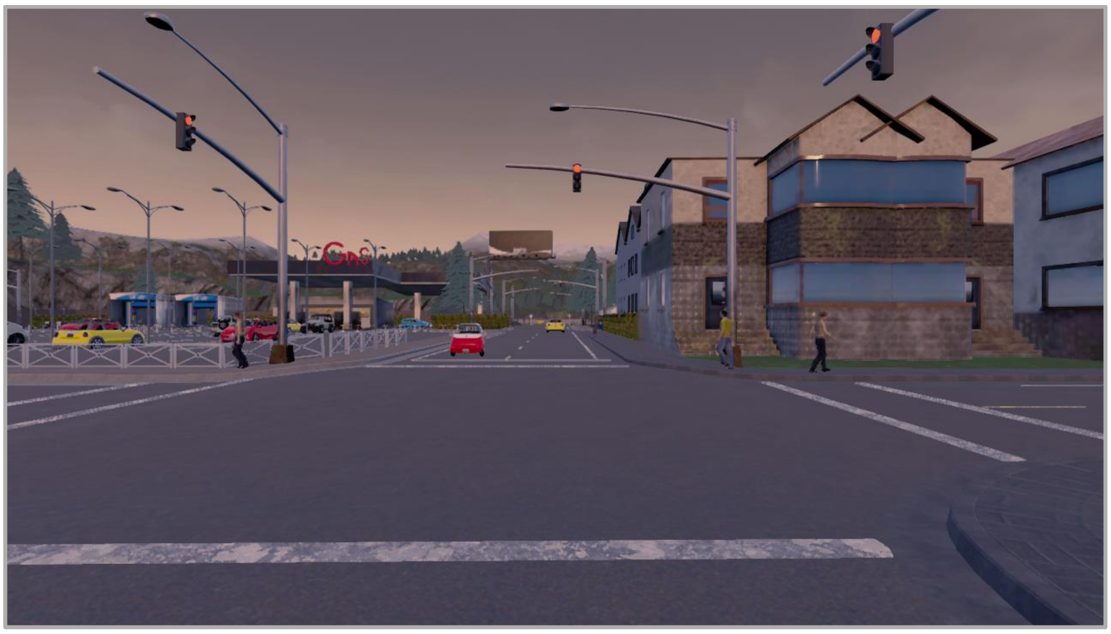


Figure 9.3 - Input image during the sunset

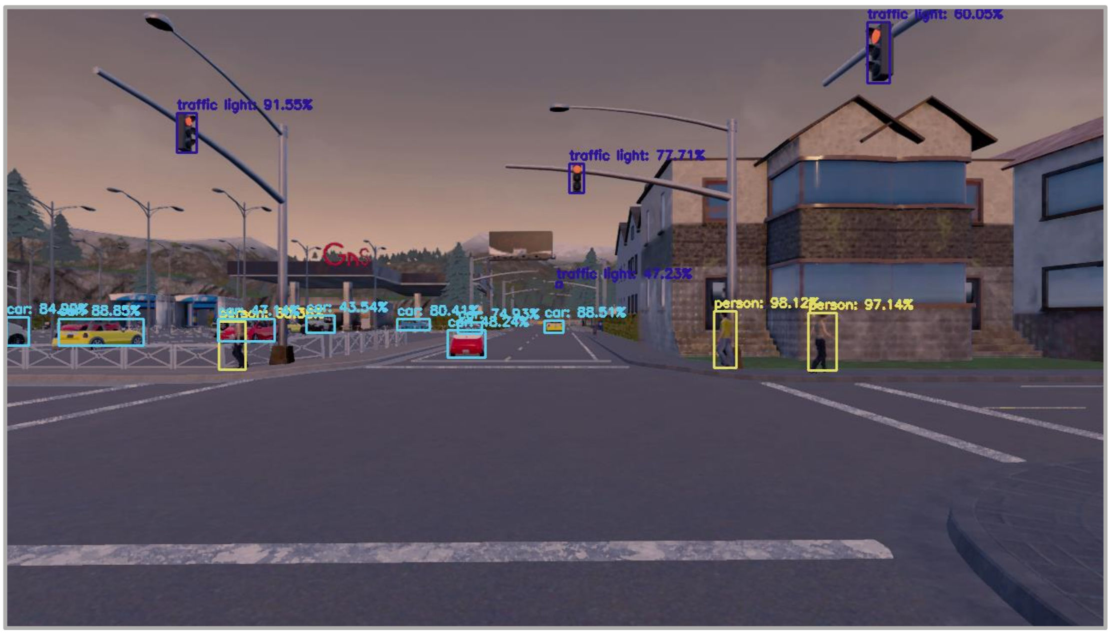


Figure 9.4 - Output image applying YOLOv3 during the sunset

Night

The last analysis to be carried out consists in considering the last scenario in which the lighting conditions are the most critical, i.e. the night. In fact, at night, natural light is completely absent and artificial lights usually come from car lights and streetlights (Figure 9.5). Although the light is almost absent, the results in Figure 9.6 show that the quality of the results is acceptable. It should be noted that the use of car headlights helped improve the confidence score.

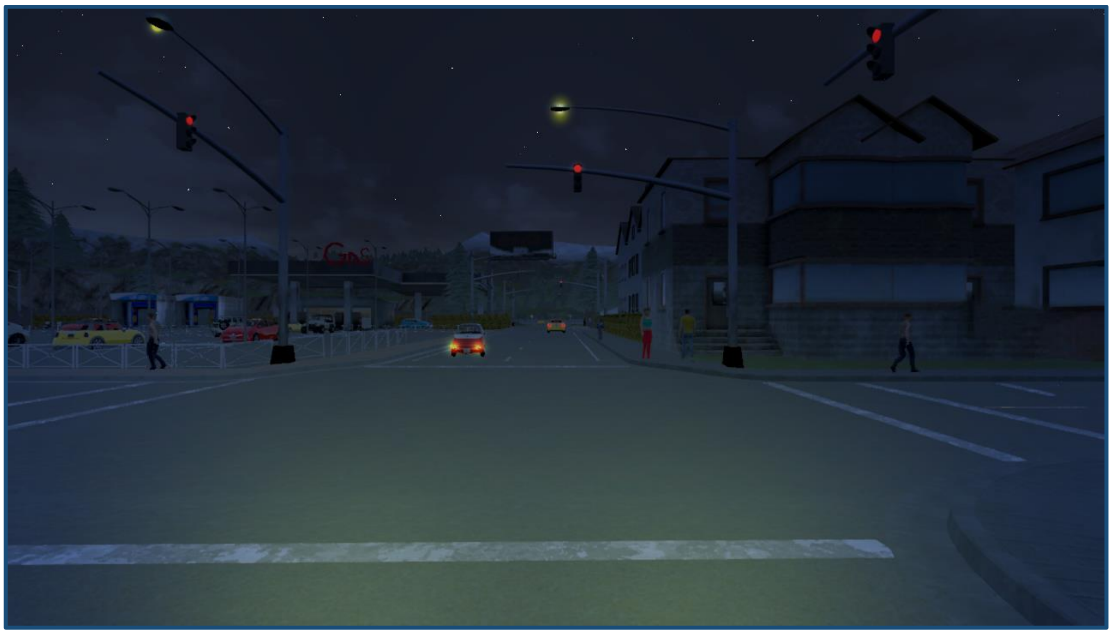


Figure 9.5 - Input image during the night

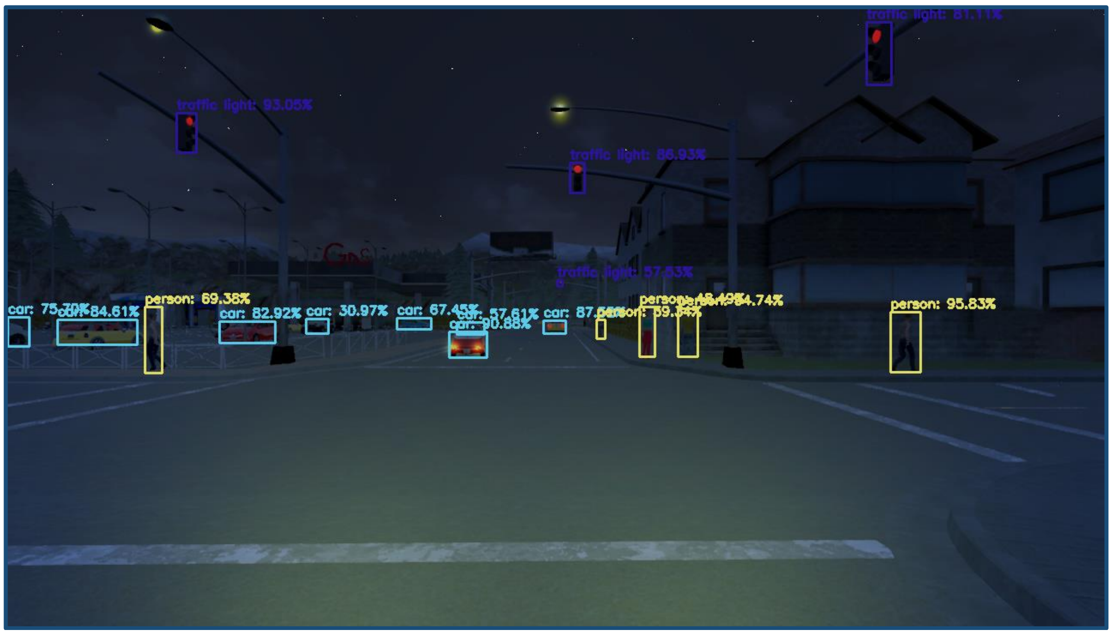


Figure 9.6 - Output image applying YOLOv3 during the night

9.2 Weather conditions

In this phase, the different weather conditions that could affect the performance of the algorithm will be discussed. As already done for the validation of the lane lines detection algorithms, two meteorological conditions have been taken into consideration: rain and fog. These situations represent the most critical points regarding visibility as they can reduce it even a lot. This is especially true for the recognition of objects and in particular for those placed at a high distance.

Fog

In this section, the effects caused by the fog will be observed and analyzed. Fog waddles eyesight, and this leads to a decrease in the performance of all those algorithms that are based on the identification of specific shapes for the recognition of objects such as YOLOv3. In fact, the fog does not allow to correctly recognize objects placed beyond a certain limit of meters (Figure 9.7).
The results obtained, reported in Figure 9.8, show how, based on the reduced visibility caused by the fog, the level of performance has visibly deteriorated. The objects that have not been recognized correctly are the traffic lights (both near and far) and the cars because they are too far away.

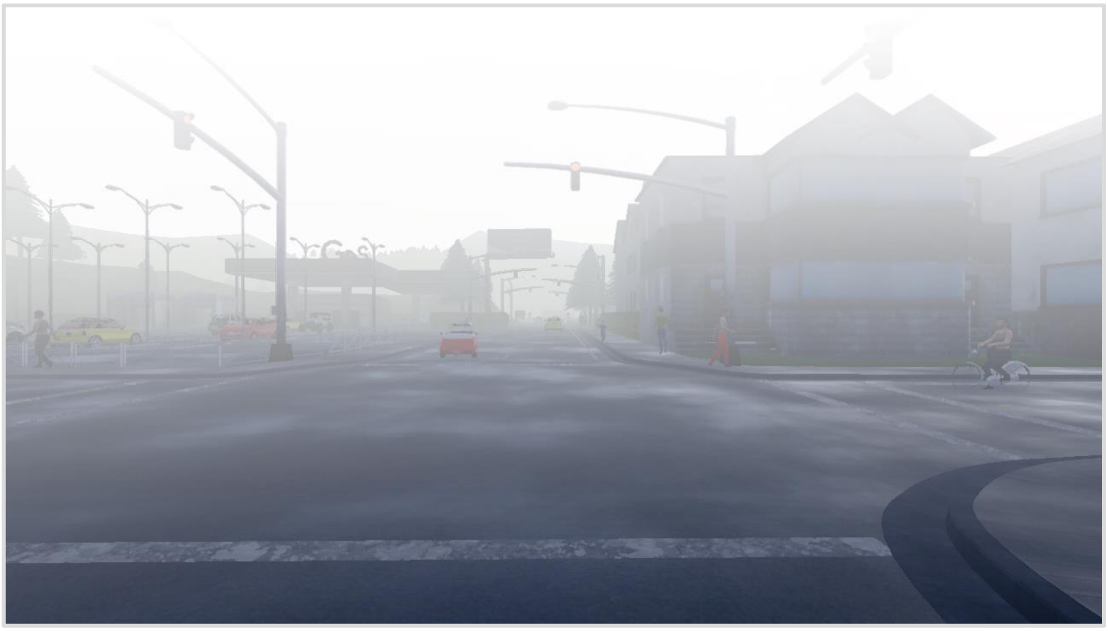


Figure 9.7 - Input image during the morning with fog

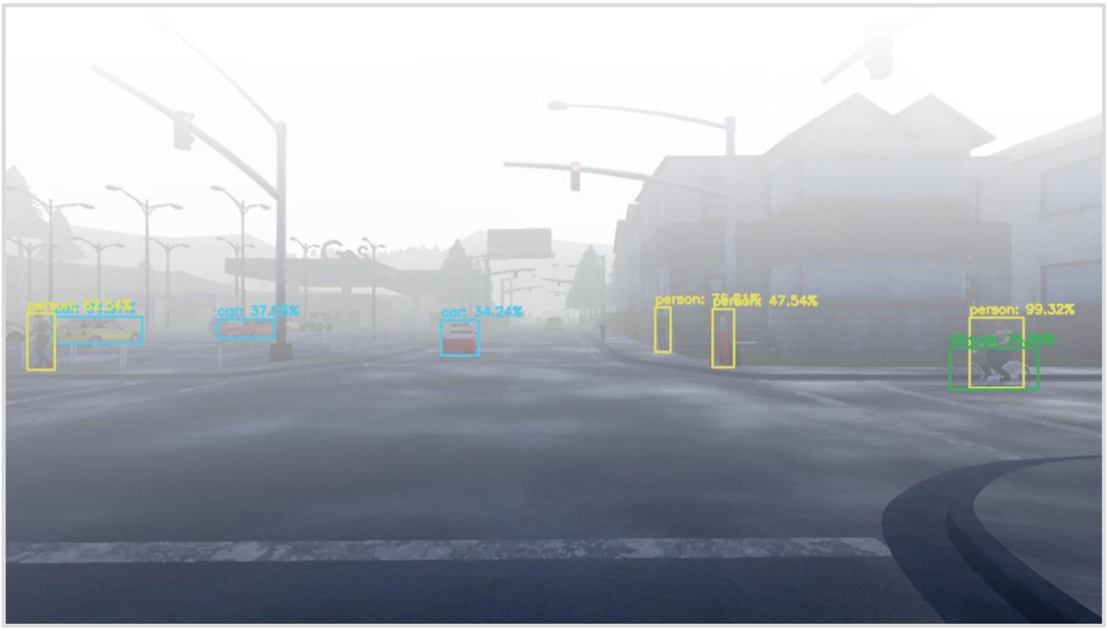


Figure 9.8 - Output image applying YOLOv3 during the morning with fog

Rain

Finally, the second and last meteorological condition was analyzed: the rain. Indeed, due to the waterdrops present, the visualization of the scene is distorted, generating an incorrect detection (Figure 9.9). The result obtained during the test reported in the Figure 9.10, shows how the quality of the results is slightly lower than when the sky is clear.
In fact, the system has not detected all the objects on the scene (such as the yellow car next to the gray jeep on the left). This is due to the fact that in the search for objects the raindrops interfered with their identification. However, the system was able to detect most of the objects in the image.

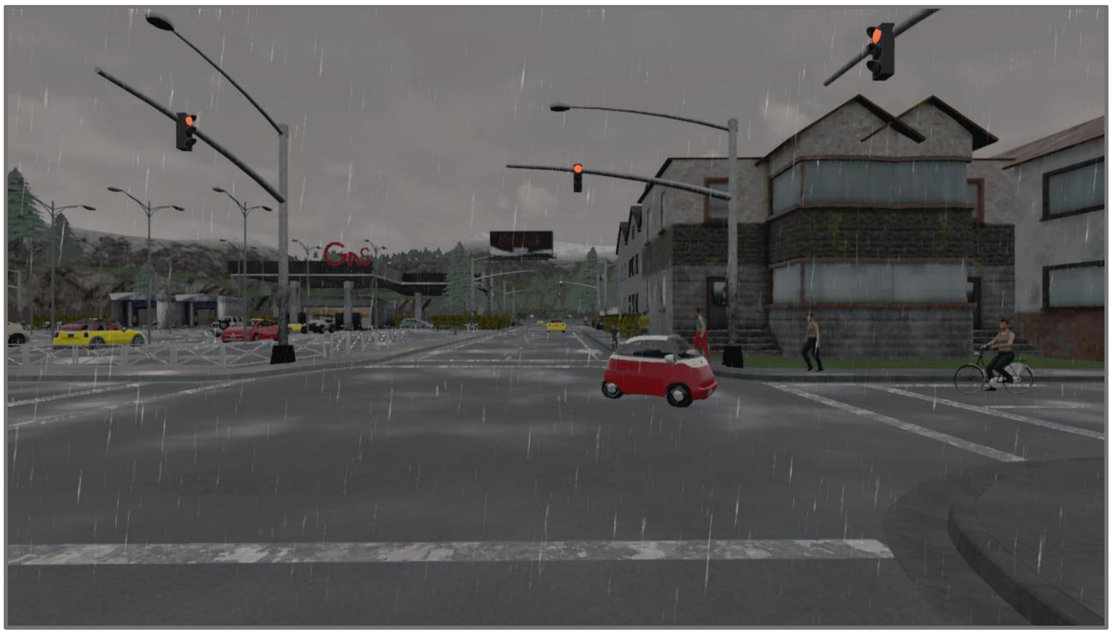


Figure 9.9 - Input image during the morning with rain

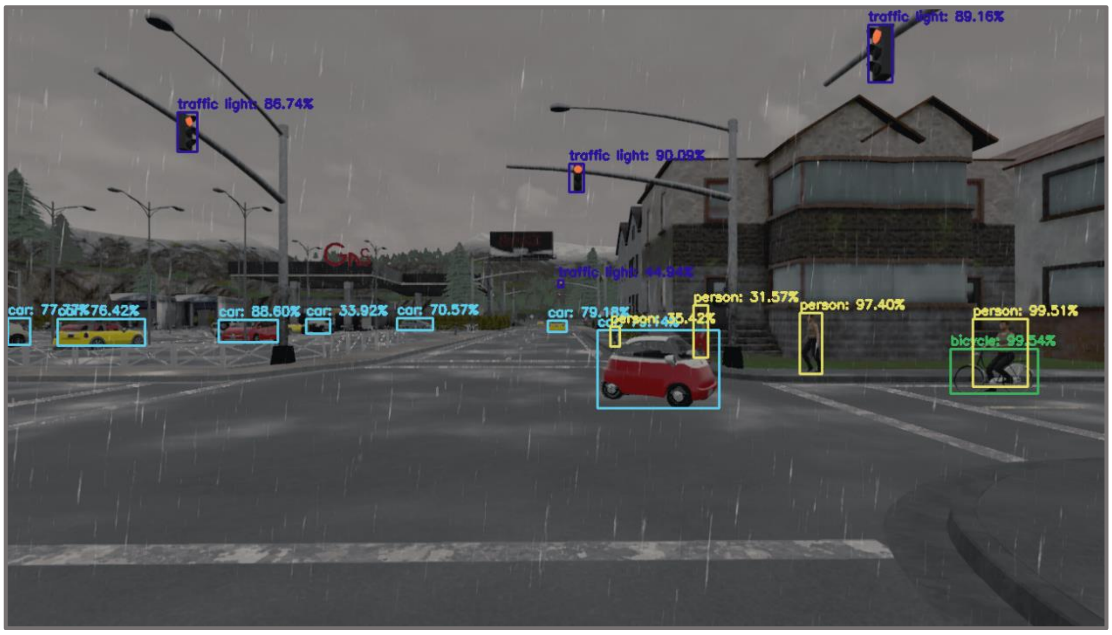


Figure 9.10 - Output image applying YOLOv3 during the morning with rain

9.3 Final Consideration

In this validation procedure, it was attempted to test the algorithm in the most realistic way using a virtual simulation and evaluating the performance on different scenarios based on different environmental conditions.
What can certainly be affirmed by observing the different tests is that YOLO, besides being a fast and precise algorithm to identify objects within an image, is also robust, and it is able to offer good results even in different weather conditions. However, its most obvious limitations can be found in the identification of near and small objects in the presence of low light conditions and in destabilizing atmospheric conditions such as rain or fog.
A conclusive chart in Figure 9.11 contains all the outcomes of the tests carried out with YOLOv3.
Basically, the graph below reports the average confidence scores reached for each object in the lighting and weather conditions considered throughout the validation procedure.

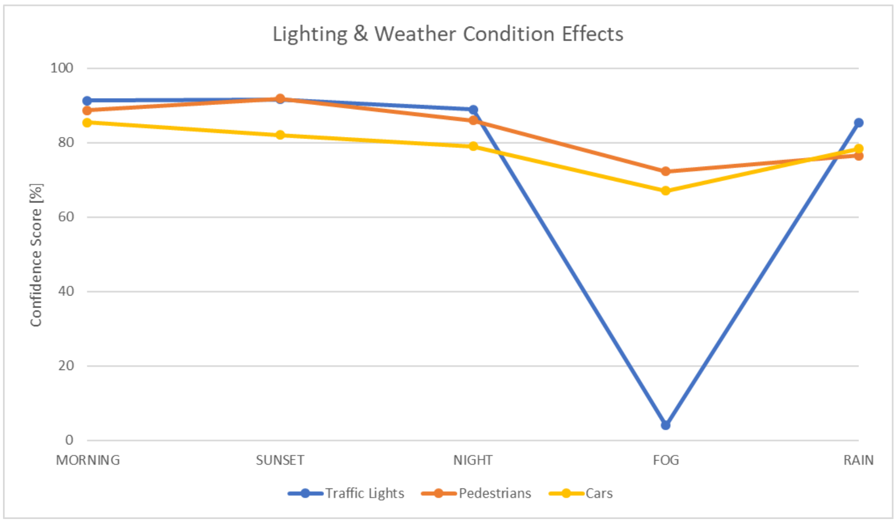


Figure 9.11 - Graphs that summarize the results obtained during the tests

In conclusion, the algorithm can also be applied to a video, considered as a combination of several images in succession. In this sense, using the available hardware, the processing of every single frame takes a long time and therefore, to process a video completely requires a considerable computational power. In fact, based on a video sample in HD resolution at 30 fps lasting 40 seconds, the algorithm, using the 100% CPU to correctly detect objects, processes each frame in about 3.2 seconds.
Therefore, it is evident that, using the available hardware, the algorithm cannot be used for real-time applications. In order to analyze its validity in fact, it is necessary to test it on powerful high-performance machines. This can be considered as a starting point for other future projects.

Conclusions and Future Improvements

The main aim of this work was the evaluation of the most important algorithms related to ADAS: lane and object detection systems. As it can be deduced from the outcomes, these algorithms, without adequate hardware available, are not yet able to guarantee a high level of speed in relation to road safety in all working conditions, because in practice, each time the camera input is affected by disturbances, for instance due to weather conditions like heavy rain or fog, the performances drop. In order to compensate for this situation, thought could be given, for example, to an additional system that employs other type of sensors that, by communicating with each other, embody the concept of sensor fusion, i.e. exploit the benefits arising from different types of sensors operating as a single entity. This project offered the opportunity to solve an interesting and stimulating problem using a wide range of methods.

Regarding lane detection algorithms (SLLD and ALLD), there is a good response under normal conditions, i.e. when the lane lines are clear and visible. The SLLD algorithm behaves well in standard conditions (i.e. without disturbance), but when the horizontal road sign becomes ambiguous (less visible), the algorithm demonstrates that it is unable to detect correctly the road path. However, it is also true that using the advanced alternative better results can be obtained. In fact, the ALLD algorithm behaves very well under even the most difficult light and weather conditions, making it an efficient and robust algorithm, suitable for future developments.
A possible improvement to this lane lines detection project in which different artificial vision techniques were used, could be to exploit various means of automatic learning, able to autonomously configure the parameters, such as the ROI vertices or hyperparameters, necessary to identify the pixels of the lane lines correctly based on situations. In particular, implement a system that is able to dynamically modify the threshold parameters (gradient and color) based on the different types of scenarios.

For what concern the object detection algorithm YOLOv3, this proves to be one of the fastest and most precise algorithms on the market, able to cope, not without flaws, with difficulties such as atmospheric conditions without lights or with difficult climatic conditions. Especially considering the fact that the tests were performed without the use of any specific training, but only with the help of a pre-trained network. This means that through proper training better performance can be achieved.
The set of defects emerged from this careful and detailed analysis represent the key point to understand the principles on which these algorithms are based, to improve them and reach a high level of efficiency. In the near future, improving even more these algorithms will be possible to reach safety standards for automotive sector.

Bibliography

[1] "ASIRT Association for Safe International Road Travel," [Online]. Available: https://www.asirt.org/safe-travel/road-safety-facts/.

[2] European Commission statistics, "MOBILITY AND TRANSPORT - Road Safety Atlas," European Commission statistics, [Online]. Available: https://ec.europa.eu/transport/road_safety/specialist/statistics/map-viewer/.

[3] McKinsey, "Ten ways autonomous driving could redefine the automotive world," June 2015. [Online]. Available: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/ten-ways-autonomous-driving-could-redefine-the-automotive-world.

[4] S. international, "AUTOMATED DRIVING LEVELS OF DRIVING AUTOMATION ARE DEFINED IN NEW SAE INTERNATIONAL STANDARD J3016," 2014.

[5] K. W. R. V. D. H. MENG LU, "Technical feasibility of Advanced Driver Assistance Systems (ADAS) for road traffic safety," 2005.

[6] D. A. S. T. I. Heinz-Peter Beckemeyer, "Get on the fast-track with automotive system innovation with Texas Instruments," 2017.

[7] A. C. Services, "Professional windshield calibration services," [Online]. Available: https://azcalservices.com/.

[8] S. M. -. T. Costlow, "Automated & Connected: Fusing Sensors for the Automated Driving Future," February 2019. [Online]. Available: https://saemobilus.sae.org/automated-connected/feature/2019/02/fusing-sensors-for-the-automateddriving-future.

[9] O. Gietelink, "Design and Validation of ADAS," 2007. [Online]. Available: https://www.dcsc.tudelft.nl/~bdeschutter/research/phd_theses/phd_gietelink_2007.pdf.

[10] T. V. S. K. R. S. Christian Buchholz, "SHPbench – a Smart Hybrid Prototyping based environment for early".

[11] T. Z. Jörg Schäuffele, "AUTOMOTIVE SOFTWARE ENGINEERING," pp. 309-310.

[12] Intellias, "Intelligent Software Engineering," 16 August 2018. [Online]. Available: https://www.intellias.com/three-ways-of-testing-adas-in-autonomous-cars-beyond-a-test-drive/.

[13] intellias. [Online]. Available: https://www.intellias.com/de/wie-die-sensorfusion-fuer-selbstfahrende-fahrzeuge-bei-der-vermeidung-von-todesfaellen-auf-der-strasse-hilft/.

[14] S. D. r. K. J. S. J. H. A. S. J.-F. B. &. i. r. Edmond Awad, "The Moral Machine experiment," *Springer Nature limited,* 2018.

[15] EURONCAP, "Sicurezza dei veicoli," [Online]. Available: https://www.euroncap.com/it/sicurezza-dei-veicoli/.

[16] A. B. Massimo Bertozzi, "GOLD: A Parallel Real-Time Stereo Vision System for Generic Obstacle and Lane Detection," *IEEE TRANSACTIONS ON IMAGE PROCESSING,* vol. 7, no. 1, 1998.

[17] A. D. e. al., "CARLA: An Open Urban Driving Simulator".

[18] C. team, "CARLA Documentation 2019," [Online]. Available: https://carla.readthedocs.io/en/latest/.

[19] AAA. [Online]. Available: https://www.texas.aaa.com/automotive/advocacy/self-driving-cars-autonomous-vehicles-explained.html.

[20] NVIDIA, "How Does a Self-Driving Car See?," 15 April 2019. [Online]. Available: https://blogs.nvidia.com/blog/2019/04/15/how-does-a-self-driving-car-see/?ncid=afm-chs-44270&ranMID=44270&ranEAID=a1LgFw09t88&ranSiteID=a1LgFw09t88-SP.OmR9UaVCgAbTlcrMbmg.

[21] M. V. -. P. d. Torino, "Sensors and actuators for autonomous driving," *Technologies for Autonomous Vehicles,* 2019.

[22] openPR. [Online]. Available: https://www.openpr.com/news/1645942/lidar-sensor-market-2019-global-overview-by-industry-verticals-leica-geosystems-ag-teledyne-optech-inc-trimble-navigation-limited-and-other-companies.html.

[23] M. E. o. A. V. Sensors. [Online]. Available: https://medium.com/@olley_io/how-to-ensure-the-safety-of-self-driving-cars-part-2-5-b4eafb067534.

[24] S. C. Davies. [Online]. Available: https://www.slashgear.com/waymos-self-driving-safety-report-is-all-about-reassuring-passengers-12503766/.

[25] G. B. Greg Welch, "An Introduction to the Kalman Filter," *University of North Carolina at Chapel Hill-Department of Computer Science,* 2001.

[26] J. W. LAHOUD, RGB-D CORRECTION AND COMPLETION AND ITS APPLICATION TO SLAM IN FEATURE-POOR PLANAR ENVIRONMENTS, Beirut, 2014.

[27] S. D. P. e. al., Perception, Planning, Control, and Coordination for Autonomous Vehicles, 2017.

[28] K. F. S. W. Sumanth Pavuluri, ROBUST LANE LOCALIZATION USING MULTIPLE CUES ON THE ROAD, IEEE, 2013.

[29] X. D. a. K. K. Tan, Comprehensive and Practical Vision System for Self-Driving Vehicle Lane-Level Localization, IEEE, 2016.

[30] Udacity, "Self-Driving Car Engineer," [Online]. Available: https://www.udacity.com/course/self-driving-car-engineer-nanodegree--nd013.

[31] O. documentation, "OpenCV," [Online]. Available: https://docs.opencv.org/2.4/index.html.

[32] CodewithC, "CodewithC," 2014. [Online]. Available: https://www.codewithc.com/gaussian-filter-generation-in-c/.

[33] B. High, "C++ vs. Python – Which One To Choose For Your Next Project?," September 2019. [Online]. Available: https://boosthigh.com/cpp-vs-python/.

[34] O.-J. S. U. o. Oslo, "color images, color spaces and color image processing," *Digital Image Processing,* 2017.

[35] wikipedia, "List of color spaces and their uses," [Online]. Available: https://en.wikipedia.org/wiki/List_of_color_spaces_and_their_uses.

[36] Coursera. [Online]. Available: https://www.coursera.org/lecture/machine-learning/sliding-windows-bQhq3.

[37] G. Alfred, Modern Differential Geometry of Curves and Surfaces with Mathematica, 2nd ed..

[38] M. J. Paul Viola, "Rapid Object Detection using a Boosted Cascade of Simple Features," *ACCEPTED CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION 2001.*

[39] B. T. Navneet Dalal, "Histograms of Oriented Gradients for Human Detection".

[40] A. K. S. T. Tariq Mahmood, "Transfer Learning Techniques for Image Recognition: A Systematic".

[41] J. D. T. D. J. M. Ross Girshick, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2014.

[42] R. Girshick, "Fast R-CNN," 2015.

[43] K. H. R. G. a. J. S. Shaoqing Ren, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," January 2016.

[44] D. A. D. E. C. S. S. R. C.-Y. F. A. C. B. Wei Liu, "SSD: Single Shot MultiBox Detector," December 2016.

[45] S. D. R. G. A. F. Joseph Redmon, "You Only Look Once: Unified, Real-Time Object Detection".

[46] A. F. Joseph Redmon, "YOLOv3: An Incremental Improvement," 2018.

[47] A. Kathuria, "What's new in YOLO v3?," 2018. [Online]. Available: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b.

Appendix CARLA Simulator

How to Build Carla Simulator

In this appendix all the steps involved in the Carla simulator configuration [18] for both Windows and Linux environment are listed. In the next pages, a detailed look on how execute CARLA 0.9.6 release will be explained.

Windows

In order to build Carla on Windows the following software requirements must be satisfied:

1. Git – a control system designed to manage projects (https://git-scm.com/about)
2. Make – a tool to maintain groups of programs (http://www.gnu.org/software/make)
3. CMake – a system that manages the build process both in an operating system and in a compiler in an independent way (https://cmake.org/about/)
4. Python 3.7.x x64 – a programming language (https://www.python.org/)
5. Unreal Engine v4.22.x – a complete suite of creation tools (https://www.unrealengine.com/en-US/what-is-unreal-engine-4)
6. Visual Studio 2017 – a complete IDE for developers (https://visualstudio.microsoft.com/it/)
7. MkDocs v1.0 package – a site generator for building project documentation (https://www.mkdocs.org/)
8. Visual C++ Toolset x64 and Windows 8.1 SDK

At this point, the CARLA Simulator repository must be cloned in a path specified by the user that it is essentially where to install CARLA. This can be done using git to download the project through the command:

```
➢ git clone https://github.com/carla-simulator/carla.git
```

to move into the CARLA folder type:

```
➢ cd carla
```

All the CARLA contents can be downloaded directly from:
`http://carla-assets-internal.s3.amazonaws.com/Content/20190710_0097e66.tar.gz`
and its content must be extracted in the following path:
`Unreal\CarlaUE4\Content\Carla`

Now everything is ready to build and launch CARLA: the entire process will download and install the required libraries. The commands that needs to be executed in sequence are:

```
➢ Make clean
➢ Make launch
```

➢ Make package

It is also possible to update CARLA at any time.

Linux

In this environment, to build Carla, there are some tools and dependencies that needs to be built before:

```
sudo apt-get update
sudo apt-get install wget software-properties-common
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add -
sudo apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-7 main"
sudo apt-get update
sudo apt-get install build-essential clang-7 lld-7 g++-7 cmake ninja-build libvulkan1 python
python-pip python-dev python3-dev python3-pip libpng16-dev libtiff5-dev libjpeg-dev tzdata sed
curl unzip autoconf libtool rsync
pip2 install --user setuptools
pip3 install --user setuptools
```

Note that CARLA requires at least Ubuntu 16.04 to work. The compiler version and C++ runtime library used was LLVM's libc++. The following changes to the clang version (6.0) must be done in order to avoid compatibility issues between CARLA dependencies and Unreal Engine:

```
sudo update-alternatives --install /usr/bin/clang++ clang++ /usr/lib/llvm-7/bin/clang++ 170
sudo update-alternatives --install /usr/bin/clang clang /usr/lib/llvm-7/bin/clang 170
```

In this phase the Unreal Engine 4.22 can be downloaded and compiled through these simple commands:

```
git clone --depth=1 -b 4.22 https://github.com/EpicGames/UnrealEngine.git ~/UnrealEngine_4.22
cd ~/UnrealEngine_4.22
./Setup.sh && ./GenerateProjectFiles.sh && make
```

CARLA is built firstly by cloning the project from the GitHub repository:

```
git clone https://github.com/carla-simulator/carla
```

Then the following environment variable needs to be set to ensure that CARLA can see the Unreal Engine's installation folder:

```
export UE4_ROOT=~/UnrealEngine_4.22
```

Now all necessary modules can be executed by means of make command:

```
make launch     # Compiles the simulator and launches Unreal Engine's Editor.
make PythonAPI  # Compiles the PythonAPI module necessary for running the Python examples.
make package    # Compiles everything and creates a packaged version able to run without UE4
editor.
make help       # Print all available commands.
```

To get the latest CARLA version and recompile it, just run:

```
make clean
git pull
./Update.sh
make launch
```

After following all the steps listed above, it is possible to start running the Carla server simulator (CarlaUE4 executable: **CarlaUE4.exe** for Windows or **./CarlaUE4.sh** for Linux):

Run Carla simulator server:

In the folder where CARLA was extracted a command window is opened and the Carla server simulator is started with the following command:

```
C:\carla-master\carla\Unreal\CarlaUE4\Content\Carla> CarlaUE4.exe -carla-
server –windowed -ResX=1280 -ResY=720 -quality-level=Medium –benchmark –
fps=10
```

After executing this command, a new window will open with image resolution at 1280 x 720 pixels, with fixed time step for the simulator equal to 10 frames per second and the simulator's rendering quality level is set to the default value.
At this point, the Carla simulator is running as a server, waiting for a client app to connect and interact with the newly created world from the server.

Run Python API client:

In the folder where the script file named "ADAS_scenario.py" is located, open the command prompt and run it without passing any argument as below:

```
C:\carla-master\carla\Unreal\CarlaUE4\Content\Carla\PythonAPI\examples>
python ADAS_scenario.py
```

In this way the simulation runs and by default:

- Open a new window with a resolution of 1280x720 pixels ('--res');
- Set the map to the "Town04" ('--map')
- Set the simulation at 10 frames-per-second ('--fps');
- Use a Gamma correction of the camera of default value of 2.2 ('--gamma');
- Set no autopilot mode activated at the beginning of the simulation ('--autopilot');
- Add 30 vehicles e 50 pedestrians to the world driving in "autopilot" mode, i.e. controlled by artificial intelligence (respectively '--number-of-vehicles' and '--number-of-walkers'). Cars will continue to drive randomly and pedestrians will continue to move to random positions until we stop the simulation, alternatively, you can use the -n or -w flag to choose how many cars or pedestrians you want to generate within the simulation.

Through the following keyboard commands, it is possible to manage the scenario just created in ADAS_scenario.py:

- P: Activate/Deactivate autopilot mode to the ego-vehicle
- TAB: Change the position of the sensor (from a third person view to first person view and vice versa)
- H: Activate/Deactivate helper on the screen
- ESC/Q: Quit the simulation
- F1: Activate/Deactivate the Head-up display (HUD) used to monitor simulation information
- C: Change the type of weather
- R: Activate/Deactivate recording images to disk (using camera sensor)
- N: Change the type of sensor
- I: Activate/Deactivate a snapshot of the scene of the simulation saving data on disk (using camera sensor)

In order to capture the images, just start the autopilot mode ("P" key) and then capture the snapshots by pressing the "I" key that will save the image in the "snap" folder.
However, in order to get a video, it is necessary to capture a series of images in sequence. To do this, simply press the "R" key to start a recording and press it again to stop it.
In this way, a large number of photos of the "recorded" simulation is saved in the "_out" folder.
In these phases, it is also possible to change the light or weather conditions by pressing the "C" key on the keyboard. Moreover, if you wish to change the view, you can do so by pressing the "TAB" key on the keyboard.
Then, using the "images_to_video.py" script and specifying the folder where the sequence of images is located, it is possible to put together the images to create a video with certain fps specified as argument.

Appendix Structure of project files

This final appendix lists the files in the folders of this thesis project and also briefly describes for each file the associated task and the functions performed. The Source Code is also attached.

CARLA

Software and libraries requirements: Carla version 0.9.6, Python 3.7.x, numpy, matplotlib, OpenCV.

1. **ADAS_scenario.py:** it is the main Python API script that communicates with the Carla Simulator server and so it is launched when the CARLA simulator has already been started. It includes both the configuration of the entire simulation and the spawn of all the actors inside it. It is used to create different driving scenarios on which to test the algorithms. In particular, it is used to extract an image of a snapshot of the simulation, or a series of images that will later be put together by the **images_to_video.py** script to generate a video.
   How to run it:
   > python ADAS_scenario.py

2. **images_to_video.py:** this script is launched immediately after collecting a series of images of Carla simualtor via **ADAS_scenario.py**.
   Given a path (folder containing the CARLA images) and fps specified in the command line, the script is able to put together each image frame creating a video.
   How to run it:
   Example: > python images_to_video.py C:/Users/Desktop/CARLA/_out/ 30

LANE DETECTION

Software and libraries requirements: Python 3.7.x, numpy, matplotlib, OpenCV, moviepy.

PYTHON

1. **find_ROI_coordinates.py:** In order to adequately and accurately choose the region of interest (ROI) for the perspective transformation, this script allows to select, through the left click of the mouse, the source points of the input image to be filtered. The input file is specified via path file in the script.
   The usefulness of this script lies in being able to avoid trial-and-error procedures to select the points correctly.
   How to run it:
   > python find_ROI_coordinates.py

2. **SLLD.py:** the script, better explained in chapter 5, represents the algorithm used to identify lane lines, for both images and a video, through several steps including noise reduction with a Gaussian filter, edge detection with the Canny method, selection of the region of interest (ROI) according to the driving scenario and Hough transform. The script, is able to recognize, based on the file extension, if the

parameter passed through the command line is an image or a video, and consequently apply the algorithm to it.
How to run it:
Example: > python SLLD.py -i=image.png (or -i=input_video.avi)

3. **ALLD.py:** This script, better explained in chapter 6, contains the implementation of an algorithm that can detect the lines of road lanes using different computer vision techniques to correctly identify the lane pixels.
   It uses the color and gradient thresholds, the bird's-eye perspective transformation and the sliding windows method with attached calculation of the polynomial to find the limits of the lane and follow them. The algorithm, in addition to identifying the exact positions of the lane lines via the histogram peaks, is also able to determine and display on the screen the radius of curvature of the lane and the position of the vehicle with respect to the center of the lane.
   The algorithm can be applied both to images and to videos, which are recognized by the script through their extension, when they are passed from the command line.
   How to run it:
   Example: > python ALLD.py -i=image.png (or -i=input_video.avi)

C++

Additional software requirements: opencv-4.1.1-vc14_vc15.exe, visual studio 2017

Note: After installing opencv library, you need to connect opencv with the visual studio 2017. You can do this by following these steps:

- Make sure to include OpenCV to system path: go to “Advanced System Settings” in “Environment Variables” section on “System Variables”. In “Path” section, click on “edit” and then on “New” to add the new environment variable. Insert the path of bin folder present within OpenCV package.
  Example: C:\user\opencv\build\x64\vc14\bin. Finally exit by clicking on Ok.
- In visual studio 2017, use x64 version in debugging environment
- Go to the properties of the project (explore solutions, in the right part of the screen), click on C/C++ and go to General. Copy the include opencv folder path and paste it in the “Additional Include Directories” section (example: C:\user\opencv\build\include) and click Apply
- Go to linker and click on General. Copy opencv library path files and paste it in “Additional Library Directories” section (example: C:\user\opencv\build\x64\vc14\lib). After that, click Apply

- Finally go to Input. Click on edit in “Additional Dependencies” section and paste the .lib file’s name. Choose the .lib file based on current configuration (example: opencv_world412d.lib). Then, click on Apply
- In conclusion, exit the properties by clicking Ok

To compile and run the project, use the visual studio 2017 software as follows:

- To compile the code, choose “Compile Solution” from the “Compile menu”. The results of the compilation process are displayed in the output window
- To execute the code, in the menu bar select Debug, Start without debugging. A console window opens and then the app runs.
- To run the code in a command window, after compiling the app from Visual Studio 2017, go to the Debug folder of the solution and open the command line to run the app

This project file represents the C++ translation of the ALLD.py python script.
Similar to “ALLD.py” as explained in detail in chapter 6, the C++ project is composed as follows:

Header file:

1. ALLD_lib.h: This file contains the function prototypes defined in the relevant file “ALLD_lib.cpp”, used to apply the ALLD algorithm

Implementation of header file

1. ALLD_lib.cpp: This file contains the implementation of the “ALLD_lib.h”.

Main file:

1. ALLD.cpp: This file contains the main program used to apply the Advanced Lane Line Detection algorithm to an image or video.

How to run it:
Example: > ALLD.exe -i=image.png (or -i=input_video.avi)

OBJECT DETECTION

Software and libraries requirements: Python 3.7.x, numpy, OpenCV.
In this section, the YOLO models are required. To download the necessary files, go to https://pjreddie.com/darknet/yolo/. It contain different pre-trained network’s weights (yolov3.weigths), various network configuration (yolov3.cfg) and the coco.names file which contains the 80 different class names used in the COCO dataset.

List of available COCO dataset classes:
https://github.com/pjreddie/darknet/blob/master/data/coco.names

PYTHON

1. **yolo.py:** The script takes care of detecting objects that belong to a certain class (such as people, traffic lights, vehicles, etc.) present both in images and in video streams.
   The input file is passed as argument with the optional parameters like confidence and threshold.
   The objects have been identified using a Deep Learning approaches: by using OpenCV's Deep Neural Network module (dnn) YOLOv3 model on COCO dataset, which is able to detect 80 different types of most common objects.
   How to run it:
   Example: > python yolo.py -i=image.png (or -i=input_video.avi)

C++

Additional software requirements: opencv-4.1.1-vc14_vc15.exe, visual studio 2017
Follow the steps of the previous note in the C++ section to correctly compile the project in visual studio 2017.

This project file represents the C++ translation of the yolo.py python script.
Similar to "yolo.py" as explained in detail in chapter 8.4, the C++ project is composed as follows:

Header file:
1. Yolo_lib.h: This file contains the function prototypes defined in the relevant file "Yolo_lib.cpp" used to apply the YOLOv3 object detection system.

Implementation of header file:
1. Yolo_lib.cpp: This file contains the implementation of the "Yolo_lib.h".

Main file:
1. Yolo.cpp: This file contains the main program used to apply the YOLOv3 object detection algorithm to an image or video.

How to run it:
Example: > Yolo.exe -i=image.png (or -i=input_video.avi)