POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master degree course in Computer Engineering

Master Degree Thesis

Lane detection algorithm for automotive applications

Implementation of a GOLD-based machine vision algorithm on a high-performance system on chip

Supervisor
prof. Massimo Violante

Candidate
Alberto Riorda
Student ID: 252788

July 2020

Alla mia famiglia, per i sacrifici fatti e il supporto datomi lungo tutti questi anni di percorso universitario, e a Serena, senza la quale non avrei raggiunto questi traguardi.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the most important research and commercial topics in the automotive industry today is the design and the development of systems capable to support the human driver when traveling by vehicle, with different levels of intervention on the vehicle control. These systems are called Advanced Driving Assistance Systems (ADAS) and they are inserted into a larger research field, which is the autonomous driving task. This topic, also named as Dynamic Driving Task (DDT) following the Society of Automotive Engineering denomination, is one of the most important multidisciplinary engineering challenges of the incoming years.

In this scenario, one of the most crucial tasks needed to improve the quality of a system autonomously controlling the vehicle is the environment recognition, namely the determination of areas where the car is allowed to go and the detection of obstacles and hazardous situations. In particular, the lane detection task is the one which is responsible to recognize the drivable area by detecting the lane markers and elaborating the obtained information to describe the geometry of the road ahead of the vehicle.

In the first section of this thesis, the autonomous driving and the lane detection topics will be addressed. In the first chapter, a general definition of the autonomous driving task will be given, and the state of the art of the most used system architectures and software/hardware components will be presented.

In the second chapter, the lane detection problem will be presented, analyzing the state of the art about this kind of applications, the most common system structures and examples of implementations.

In the third chapter of the introduction section, a specific lane detection application will be addressed, the GOLD system. The algorithm used for detecting lane

markers in this system will be the base of the implementation of the thesis application.

The core topic of this thesis is the implementation of a real-time lane detection system on a high-performance system-on-chip running an Embedded Linux distribution as operating system. The main lane detection algorithm is based on the GOLD system, with some differences in certain low-level operations.

The system goal is to recognize the lane markers from a given image of the road ahead the vehicle and to compute a visual and a geometric description of the lane center. The system is composed by: a low-level module that takes as input a stream of frames and provides as output a stream of frames in which the lane markers and the center line of the current lane are highlighted; a graphical application, developed using the Qt framework, responsible to visualize the output stream on a screen.

In the second section of the thesis, the software architecture of the system will be firstly presented, describing the methods and the optimizations applied to reach a real-time elaboration of each incoming frame. Then, the description of the hardware components used during the development and the testing phases will be addressed.

Finally, in the third section of the thesis, the testing results will be analyzed in terms of computational time performances and in terms of effective correct detections.

As appendices, two topics will be addressed: the 3D to 2D image transformation, to deeply explain the process followed during the pre-processing module of the algorithm; the Embedded Linux and Yocto system, to explain how the operating system used in this thesis is composed and how it was adapted to the application needs.

Part I

General introduction to addressed topics

Chapter 2

Autonomous driving

2.1 Introduction to autonomous driving

Nowadays, one of the most important and complex challenges in the automotive field is the specification and the implementation of an automated system capable to fully control and drive a vehicle, without the direct intervention of a human driver. This is a very complex task, gathering multidisciplinary know-hows coming from different engineering disciplines and applying the newest technologies in the topics of sensors, actuators, electronics, computer vision, artificial intelligence and many others.

Several examples of feasible and working systems already exist in the field of the railway transportation (e. g. the Turin automated underground) and in the aeronautics industry (e. g. auto pilot implementations controlling the flight on planes). Moreover, some driver-less automated applications will be soon released in the ambit of the autonomous driving farming machines. Conversely, a real implementation of a fully automated system concerning automotive and commercial vehicles is not present on the market and it is still in an open research phase. The reason of this discrepancy can be found on the complexity of the different working scenarios. In the case of a train or an underground wagon, the path is constrained by a railway, theoretically free of any incoming obstacle. In the aircraft case, the flight trajectory is previously computed and obstacles (usually other planes) are sensed at high distances and should not suddenly appear in front of the system. The road scenario, instead, is composed by several elements of different types and with a very dynamic and unpredictable behavior. For example, in an urban crossroad, an automated driving system should take into account the traffic light, the lane boundaries, the other vehicles position and behavior, possible other actors like pedestrians and bikes that should suddenly cross the road, etc. A system operating in such environment must detect and distinguish every possible obstacle in a reliable and efficient way, must be reactive to external threats and must be capable

to dynamically update its behavior in response to those inputs. Moreover, it must work in every condition, for example during a heavy rain or snowfall, with daylight or at night.

The reasons that drive the efforts to reach the autonomous driving goal are multiple. The most important one is the safety increasing. According to a recent technical report drawn up by the US body of the National Highway Traffic Safety Administration (NHTSA), the number of road accidents caused by a human error is the 94% of the total ([1]). According to the Italian body ISTAT-ACI ([2]), in Italy the number of vehicular accidents for the year 2018 has an amount of 172,344, causing 242,621 injured and 3,325 deaths. The main causes are distraction, failure to observe precedence rules and high speed, totaling the 40.8% of the cases. Moreover, the highest number of road accidents resulting in death happens during the night, when drivers easily feel tired or drive under influence of substances, with a peak of 9 death every 100 accidents between 5 am and 6 am. All these factors underline the need for a system able to help the driver to avoid dangerous situations for himself/herself and for other road users, substantially reducing the number of traffic accidents.

Another problem that could be reduced thanks to autonomous vehicles is the traffic congestion and, consequently, the emission of polluting gases ([3]). In fact, an automated driving system should be able to collect traffic information from online applications and to find an optimal way to reach the destination set by the user, reducing also the traveling time ([4]). Moreover, controlling directly the engine valve (in case of traditional fuel motors) may help to optimize the fuel consumption.

This kind of system will also permit a reduction of the stress and the fatigue normally generated when driving in traffic situations or long trips ([20]). The time that was used by drivers in such situations should be relocated and invested in more useful activities. Finally, the usage of an autonomous vehicle permits to extend the mobility concept to portions of the population which normally may not or are unable to drive a car. This aspect can have a huge impact on the quality of life and productivity of those portions of population and the people near to them ([4]).

However, reaching the goal of a fully automated driving is very complex, and may lead to a massive increase of the development and production costs of this kind of systems. Some studies, like [5], underlined the fact that this technology may be very expensive for producers, leading to a shared use of autonomous vehicles in driverless taxi service, instead of selling vehicles directly to single customers.

Moreover, the doubts on this technology are mostly linked to the possibility of

a system failure. There exist several examples of accidents caused by failures during tests or during the real-world usage of semi-automated driving systems already present on the market. For example, in 2016 a Tesla Model S equipped with a semi-automated autopilot failed to recognize a white truck against the sky light and crashed into it, killing the driver ([6]). In 2018, an Uber experimental car killed a pedestrian who was crossing a road carrying a bike ([7]). The system, due to the darkness of the view, firstly classified the person as an unknown object, then as a vehicle and finally as a bike, delaying the moment of the emergency braking activation. The reason of these accidents is that systems available today are still not robust enough to handle all the situations that is possible to encounter in real scenarios.

Finally, the concept of a vehicle without the control of a human driver has generated a series of ethical and legal problems. In case of accident due to a sudden system failure, who is responsible of the consequences? If there is a dangerous situation in which the car has to choose between save the passengers or some pedestrian, what will the car do?

2.2 Definition of autonomous driving according to SAE

A precise definition of what the autonomous driving task is can be derived from the SAE, the Society of Automotive Engineers ([8]). This association denominates it "Dynamic Driving Task" (DDT), defining it as "the set of the real-time operational and tactical functions required to operate a vehicle in on-road traffic".

According to SAE, these functions can be gathered into some subsets:

- Lateral vehicle motion control via steering.
- Longitudinal vehicle motion control via acceleration and deceleration.
- Environment monitoring via object and event detection, recognition, classification and response preparation.
- Response execution to objects and events.
- Local motion planning maneuvering.
- Conspicuity enhance via lightning, signaling and gesturing.

The trip scheduling by means of destination choose and way-points computation is excluded from these subsets, as other strategic function which are considered not

strictly correlated to the Dynamic Driving Task.

SAE also defines the concept of an "Automated Driving System" (ADS) as "the hardware and the software that are collectively capable of performing the entire dynamic driving task, regardless of whether it is limited to a specific operational design domain". Moreover, the hardware and software performance of a specific part or all of the dynamic driving task is named as "Driving Automation" (DA). Finally, the Operational Design Domain (ODD) is defined as the "operational conditions under which a given driving automation system is specifically designed to function". An example of an ODD can be a particular environment, such as a highway with some specific traffic and weather conditions: a given ADS is expected to function in this ODD, but it can have an unreliable behavior in case of a sudden weather change.

The general dynamic driving task can be fully or partially performed either by a human driver or by an automated driving system. Depending on which actor is performing determined tasks and which are the operating conditions, the SAE distinguishes six discrete and mutual exclusive levels of driving automation:

- Level 0, or "No Driving Automation". In this level, the entire dynamic driving task is performed by the human driver at any time. It is possible, anyway, that supporting active safety systems, such as the Anti-blocking System (ABS) or the Electronic Stability Program (ESP) system, are present onboard of the vehicle.

- Level 1, or "Driver Assistance". In this level, an automated driving system can take control of either the lateral or the longitudinal control sub-task of the DDT, but only when activated and under a limited operational design domain. The remaining tasks are under the control of the human driver. Examples of ADS corresponding to this level can be the Adaptive Cruise Control (ACC) or the Lane-keeping Assist (LKA).

- Level 2, or "Partial Driving Automation". In this level, an automated driving system can take control of both the lateral and the longitudinal control sub-task of the DDT, only when activated and under a limited operational design domain. The human driver has the responsibility to perform the other tasks, such as the object and events detection and response and to supervise the execution of the automated tasks. As in level 1, if a failure occurs in the automated driving system or the vehicle exits from the ODD, the driver must be reactive and immediately take control of the whole DDT. An example of level 2 ADS can be the cooperative usage of an Adaptive Cruise Control and a Lane Keeping Assist.

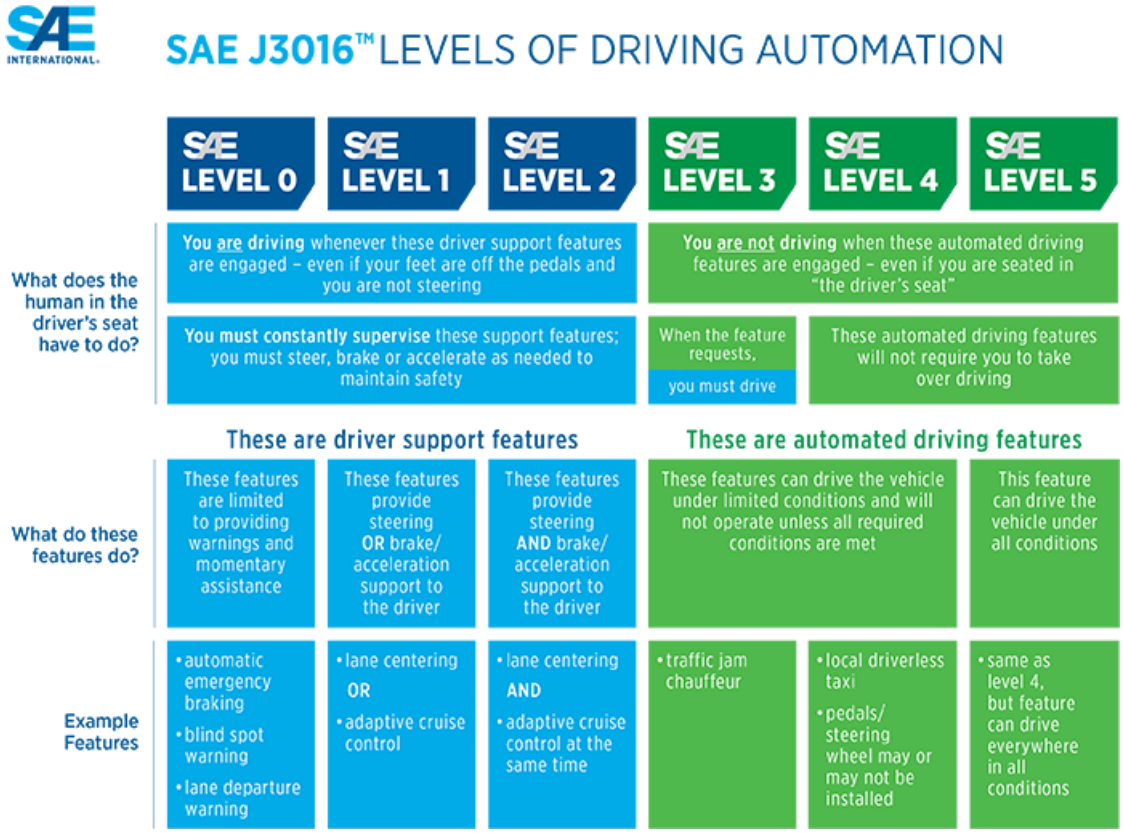

Figure 2.1. SAE - J3016: table of levels of automation

- Level 3, or "Conditional Driving Automation". From this level on, the automated driving system is capable to perform the overall dynamic driving task. In this level, the ADS takes a full control of the vehicle while engaged and only under specific operational design domains. However, the human driver is responsible to immediately take control of the vehicle in case of a system failure or in case the vehicle exits from the ODD.

- Level 4, or "High Driving Automation". In this level, the automated driving system is responsible of the overall dynamic driving task in limited operational design domains. The system is also responsible of the fallback procedure, in case of a failure or the exiting from the ODD. In this level, the human driver is not responsible of any task and it is not expected that he/she is reactive to a fallback procedure.

- Level 5, or "Full Driving Automation". In this level, the automated driving system is responsible of the overall dynamic driving task in every operational design domain, without restrictions. As in level 4, the fallback procedure is

under the system control, while the human driver has not any responsibility on the vehicle control and on its behavior.

Although autonomous driving is a topic on which automotive industry is importantly investing money, at the state of art of the vehicle production and product release, currently sold automated driving systems typically belong to the level 2 of the SAE classification. One of the most important examples of implementation in this level is the Tesla Autopilot, capable to control both the longitudinal and the lateral dynamic in limited situations, like highways. In the first half of 2019, Tesla cars have reached the amount of 1.6 billion of traveled kilometers with this system activated.

However, the majority of the standard vehicles available on the market today are restricted to the level 1 of the SAE classification. In fact, the various systems introduced permits an automated control of either the longitudinal or the lateral dynamics, by using separately different Advanced Driving Assistance Systems (ADAS). Examples of ADAS are the Adaptive Cruise Control (ACC), the Emergency Braking Assist (EBA) and the Lane Keeping Assist (LKA).

In terms of near future, many companies are currently testing and validating automated driving systems at level 3 and 4 of the SAE classification. One of the most advanced development is carried by Waymo, a company controlled by Google that has recently reached the 16 millions of traveled kilometers, while other companies like Tesla, Uber and GM – Cruise released optimistic declarations saying that the first level 3 and 4 vehicles might be ready in the first years of the 20s.

2.3 General structure of an automated driving system

The task of developing and implementing a working and reliable automated driving system is very complex. Several system architectures have been proposed among last years to efficiently approach this problem ([9]). Among the state of the art in architectural decisions, we can find two different approaches for what concern the connectivity and two different approaches for what concern the algorithmic design.

For the first topic, we can define an “ego-only” system and a “connected” system. Ego-only systems are based on the idea of having all the necessary hardware and software component needed in order to perform the dynamic driving task already onboard of the vehicle. This is the common approach used among the state of the art. The main advantage is the independence of the system from the information coming from other vehicles or infrastructure. All the inputs from the environment

needed to perform a task are coming from internal sensors, which may consider more reliable than unknown external sources. Moreover, having a self-sufficient platform can help the overall system development and validation phase. The disadvantages are linked to the fact that certain information cannot be easily derived without an external communication, causing the impossibility to obtain certain environmental inputs or implying a delay collecting them.

Connected systems, instead, have an approach based on traffic and state information sharing among peers and on communications with infrastructure elements. In order to communicate, a Vehicular Ad hoc NETwork (VANET) is used: it is a wireless broadcast network using 802.11p IEEE standard or exploiting the 4G/5G cellular network. The communication can happen between two or more vehicles, for example sharing the velocity and direction information; in this case, it takes the name of V2V (vehicle to vehicle) communication. Otherwise, it can happen between a vehicle and an infrastructural element, for instance a traffic light sharing the time to the red-light power on; in this case, the communication takes the name of V2I (vehicle to infrastructure). In general, referring to connected vehicles, it is possible to use the acronym V2X (vehicle to everything) to describe both the previously presented communication types. This kind of approach can enhance the system environment sensitivity with respect to an ego-only system, but it has not been operationally implemented yet due to complexity of the scenario where hundreds or thousands of vehicles exchange data in small city portion.

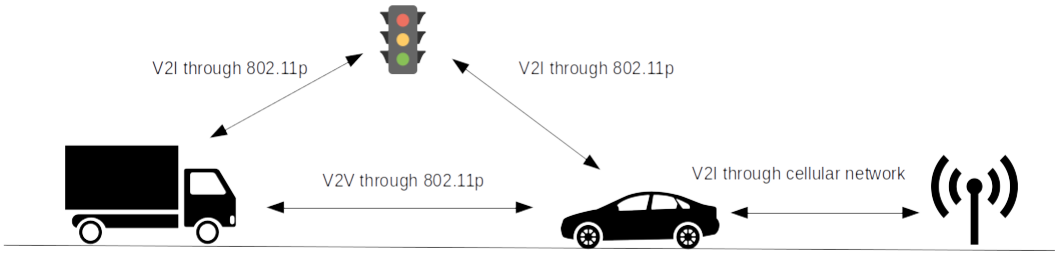

Figure 2.2. Example of a Vehicular Ad hoc NETwork (VANET)

Concerning the different architectural approaches in the algorithm design topic, there exist two possible alternatives. The first one is the “modular system” design, which consists in a structured pipeline of different components linking the environment, the internal state inputs and the actuation outputs. This is the most used choice in the state of the art. A commonly used pipeline is composed by the

following steps:

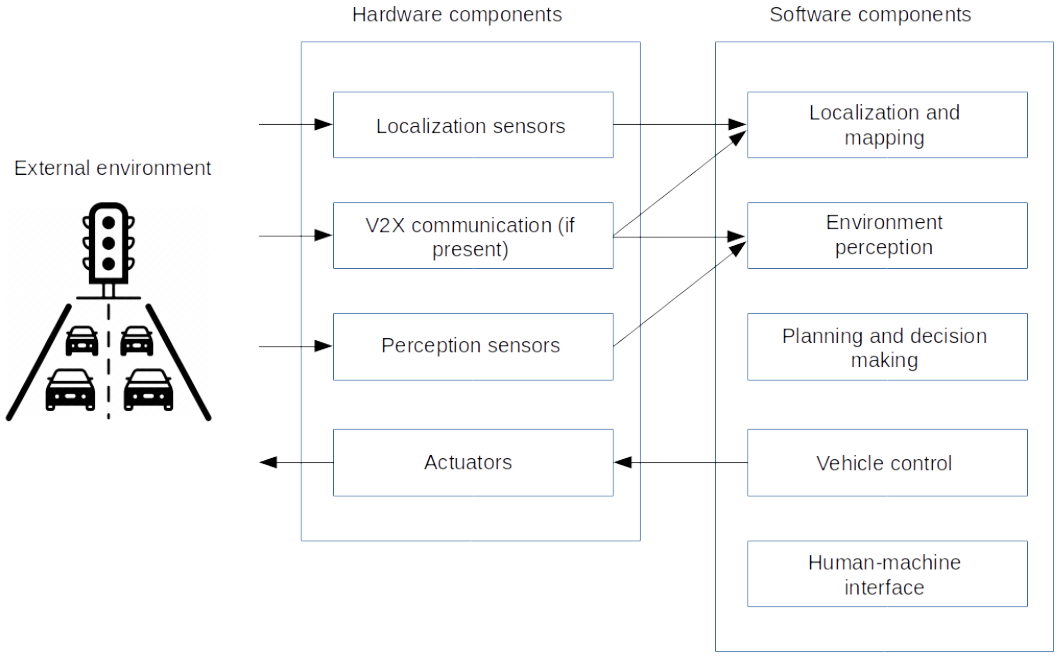

Figure 2.3. Architecture of a modular autonomous driving system

- Localization of the system and environment mapping, for example using a GPS sensor.
- External and internal inputs perception, for example using cameras and radars.
- Environment assessment, by using an artificial intelligence algorithm.
- Planning and decision making, starting from the assessment outputs.
- Vehicle control through control algorithms and actuators.
- Human-machine interface.

The main advantage of this approach is the possibility of developing and testing separately each module, dividing the dynamic driving task in several relatively easier sub-tasks, each of them having solutions already consolidated in robotics, computer vision and vehicle control system literature. Moreover, each module can be reused between different systems and permits redundant parts only in critical components. The main disadvantage regards the concept of error propagation: in case of an error in the lower modules of the pipeline, the next module will compute

its output starting from an incorrect input, causing a system failure when actuating the vehicle control.

Finally, the "end-to-end" design is based on the concept of generating motion control actions directly from the inputs, using a single module. This solution can be applied by using three approaches related to the machine learning topic:

- Direct supervised deep learning, where the system learns which actions it must take given some inputs, imitating a ground truth example (usually an expert driver). The training can be performed offline, without already using the system in a real environment. Being based on a limited supervised training, this solution has a poor generalization capacity.
- Neuro-evolution learning, similar to the preceding solution, but based on evolutionary algorithms to train the neural network. This approach does not need a backpropagation nor a supervised learning.
- Deep reinforcement learning, that has a different approach with respect to the preceding ones. In this case, the systems learns which actions it has to take given the inputs in order to choose the "best" solution, trying to minimize some reward functions. In this way, the system has not a human model to imitate, but learns on its own the optimal way to approach a scenario. This guarantees a better generalization capacity than the previous solutions, but it needs an online training in real-world cases.

The biggest problem using this kind of solutions is related to the fact that these networks has to massively interact with the environment and fail a lot of times in order to obtain a functional and working implementation. Moreover, in case of failure it is very hard to understand which errors occur and why, making difficult to fix the problems.

In terms of basic hardware components, an automated driving system needs a set of sensors in order to grab information from the environment, a set of actuators performing the decided actions as output and a computing platform implementing the previously presented system architectures. The actuation is performed on the steering system to control the lateral dynamics, on the braking system and on the power-train system to control the longitudinal dynamics in deceleration and acceleration respectively.

Sensors can be divided into two main types:

- Proprioceptive sensors. These are sensors measuring internal state quantities, for example the velocity, the lateral and longitudinal acceleration, the roll, pitch and yaw angles, etc. Some examples of this kind of sensors are encoders and Inertial Measurement Units (IMUs).

Table 2.1. End-to-end design methods

Method	Pros	Cons
Direct supervised learning	Can be trained offline	Has a poor generalization capacity
Neuro-evolution learning	No needs for backpropagation	It has not already been successfully implemented in real cases
Direct supervised learning	It is not based on human behavior	It has not already been successfully implemented in urban real cases

- Exteroceptive sensors. These are sensors collecting data coming from the external environment, such as obstacles, drivable areas like lanes, etc. Some examples are optical cameras, radars and LiDARs.

The most used sensors in automated driving systems are cameras, radars, LiDARs, GPS and IMUs. Cameras are mainly used for the objects and events detection task and can be monocular, stereo (permitting to derive the distance information) or 360° cameras. The output can be a stream of RGB images, infrared images, etc., and is differentiated by parameters like the frame produces per second (fps), the resolution of the single frame, the focal length and many others. They are passive sensors, being that they do not emit any signal, but only collect color or infrared information.

Radars and LiDARs, instead, are active sensors: both emit beams of electromagnetic waves, that bounce back to sensors, permitting to measure the distance from an object knowing the returning time and the signal speed. Radars emit radio waves, permitting to sense objects at high distances, while LiDARs emit infrared waves, obtaining more accurate results at nearby distances.

Finally, the Global Positioning System (GPS) and IMUs are cooperatively used to compute the precise position of the vehicle in the environment. The GPS is a satellite system computing the position of the receiving sensor given a set of information sent by a satellite. Being this computation effected by several physical problems (such as the signal refraction and reflection due to obstacles), the obtained measure should be fused with the internal state information coming from the proprioceptive IMU sensor.

Chapter 3

The lane detection problem

3.1 Introduction to lane detection

One of the most crucial task that an automated driving system has to deal with, as specified by the SAE definition, is the environment monitoring. This task is of pivotal importance because allows the vehicle to act on the longitudinal and lateral dynamics control in a proper way and to respond to external inputs in time. Nowadays, the main bottlenecks in development and research on this task are represented by the obstacle detection and classification and by the road and lane detection, i. e. the recognition of an area in which the vehicle can move into (named as drivable area).

These two sub-tasks of the environment monitoring task have the same main crucial problem: they must work in every environmental condition, including very variable brightness (e. g. due to shadows) and any kind of extreme weather conditions. Such a generalization ability is still a research open problem, alongside with the need of detecting a variable number of lanes in different road topologies or with different markings size and color in the case of lane detection. Moreover, this kind of tasks are used to close the loop of lateral control algorithms, so a very low error rate is required, in the order of few erroneous detection per hour in case of warning systems and even less in active control systems. This reliability requirement is very hard to obtain in computer vision-based applications.

In this thesis, the focus will be centered on the lane detection task, which is namely the process to recognize the area in the road space belonging to a lane and delimited by its lane markers. In order to solve this problem, many solutions were proposed in literature, often based on the same perceptual inputs used by human drivers: in particular, the road color and texture, the road boundaries and the lane markers. However, it should be possible to integrate other inputs, such as vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I) communications, but, especially for the

second type, it cannot be guaranteed that in every road this kind of service would be implemented and maintained, due to the huge required costs.

Outside of the context of a full automated driving, the lane detection task is the base perceptual input level of several advanced driving assistance systems. The most important are:

- The Lane Departure Warning (LDW), which simply alerts the driver when the vehicle is exiting the current lane.
- The Lane Keeping Assist (LKA), which corrects with a limited authority the lateral direction in order to maintain the current lane.
- The Automatic Lane Keeping Control (ALKC), which acts as the LKA, but having a full authority on the vehicle lateral control.

In these cases, the system must be able to recognize the current lane area within relatively short distances (between 40 and 50 meters). In more advanced systems, such as Tesla Autopilot, the lane detection module is also used to recognize adjacent lanes and perform an autonomous lane changing task. In this case, the system should be able to detect multiple lanes at long distances (about 150 meters).

The first step for every lane detection algorithm is to obtain inputs from the external environment. In order to obtain this information, several sensors have been proposed in literature, such as:

- Cameras (either single or stereo vision).
- Light Detection and Ranging (LiDAR) sensors.
- Inertial Measurement Units (IMU, sensors measuring vehicle dynamics).
- GPS systems.
- Digital maps.

The most frequently used sensing modality in literature is the camera-based one. The reasons behind this choice are mainly two: the first is the fact that lane markings and road boundaries are designed for human drivers' vision, therefore they should be visible in almost every condition also for cameras; the second reason is the low cost of such sensors that, jointly with their robustness, guarantees a good cost-effective solution. Having a single camera, the implementation cost is very low, but it is not possible to obtain a 3D description of the environment. Such target can be reached by using a stereo vision system; however, this second solution can lead to higher computational costs and to higher error probability.

The other main solution is the LiDAR, a sensor that can efficiently measure the 3D structure of the vehicle surrounding environment and which can be efficiently used alongside with cameras to handle situations where the first sensor is not able to correctly produce effective results. For example, when road markings are not visible or even absent, LiDAR can be able to detect the road boundaries and their distance with respect to the vehicle. Moreover, this solution cannot be affected by natural light issues or by low-visibility situations. LiDAR can be also used to compute incoming slopes, in order to better adapt image transformations like inverse perspective mapping (see Appendix B). The main drawback is the high cost of the LiDAR sensors, but they can lead to a more precise and accurate 3D description of the environment than the stereo vision solution.

Table 3.1. Sensors for lane detection task

Method	Pros	Cons
Mono cameras	Very simple and cheap. May perform well thanks to the visual characteristics of lane markers	Do not allow a 3D reconstruction of the scene, affected by bad weather conditions
Stereo cameras	Low cost, allow the computing of a 3D reconstruction if the environment	The distance computation can be complex and not precise
LiDARs	Very precise 3D reconstruction, do not heavily affected by bad weather conditions	Very useful when detecting road borders, but not when detecting lane markers; very high cost
GPS with IMUs and digital maps	Do not affected by visibility issues	GPS satellite communication is not reliable and IMUs error probability grows in time

Finally, it is possible to integrate GPS sensors information with IMU measures in order to obtain the current vehicle position with an accuracy of 1 meter. In this way, it is possible to guide the vehicle without considering a visual representation of the current lane markings but using the digital maps description of the environment. The main issues in this case is the low reliability, due to the need of a

continuous communication with GPS satellites and the calibration error probability.

From the same inputs, there exist two main approaches in literature to the lane detection problem:

- Traditional methods, which rely on handcrafted features and heuristic algorithms. This is the conventional approach mostly used in the industry.
- Machine learning-based methods, which are mostly based on convolutional neural networks (CNN). This is a new approach addressed by researchers.

3.2 Traditional algorithms

Traditional lane detection algorithms are methods based on a handcrafted feature selection, relying on an a priori knowledge of the problem and on its perceptible characteristics. A low-level pixel elaboration is performed on these selected features and the final results are obtained in a deterministic way. Considerably more than neural network-based methods, these algorithms highly rely on the quality of the input images, which have to contain the wanted features with a good visibility. However, due to the a priori feature selection, the image can be divided in smaller regions of interest (ROI), leading to a lighter and faster elaboration.

Almost all traditional lane detection algorithms proposed in literature follow a similar flow ([10], [11], [12]), divided in several steps, which are:

- Pre-processing (image cleaning).
- Low-level processing (feature extraction).
- Post-processing (lane model fitting).
- Temporal integration.
- Image to world correspondence.

The image cleaning process is the first module of the typical flow. In this module, the input image is elaborated in order to enhance the characteristics on which the feature extraction will be based. Some examples of image cleaning operations are: the obstacle detection and removal, based on 2D or 3D points tracking mechanisms; the shadows removal, based on color-space transformations; the reject of parts of the image that do not contain relevant features. This last operation is based on regions of interest selection, which will be the only parts of the input that will be elaborated by the other modules. In [16], a vanishing line is dynamically computed and then all pixel above this line are discarded. In many others works, like [13], [14] and [15], an image transformation is applied, in order to remove the perspective

effect and select only a limited remapped area.

The feature extraction process in lane detection systems is the module where the relevant features, namely the lane boundaries, need to be detected. The elaboration is performed taking into account the low-level pixel description of the image. Generally, these boundaries are represented by lane markers, which can have different physical characteristics: they can have various colors (from shades of white to yellow or orange), various widths and shapes (continuous lines, dashed lines), etc. However, being lane markers designed to be recognizable by a human being, some assumptions can be taken. The simplest assumption is the fact that the markers are brighter with respect to the road, leading many proposed algorithms (e. g. [13] and [15]) to search for dark to light to dark pixel brightness intensity bumps, eventually filtered with adaptive or fixed thresholds. In other cases, like [14], Gaussian filters are applied to the cleaned image. In [16] the assumption is that lane markers are lines that converge into the perspective focusing point.

The lane model fitting is the process in which the markers extracted features from the preceding module are fitted into a priori lane geometric models, which can usually be divided into:

- Parametric models.
- Semi-parametric models.
- Non-parametric models.

Parametric models are usually straight lines, parabolic curves or circumference arcs. These are the simplest models, based on the assumption that to obtain a good control algorithm relatively short distances are needed, where the lane boundaries can be approximated. The most used method to fit the features to a linear model description is the Hough transform, which searches for the most frequent line inclination angle. Some example are: [13], that assumes the road markings as parallel lines after an inverse-perspective mapping transformation; [17] solution uses a clothoid, a curve whose curvature is in linearly proportional with its curve length; [15] uses the RANSAC method to remove outliers and fit to a linear or hyperbolic model.

Semi-parametric models, e. g. spline curves, are more precise models, that do not take any assumption on the expected lane geometry. Moreover, differently from parametric models, a small parameters changing leads to a small curve geometry change. The drawback is the possibility of an over-fitting, with unrealistic curves as outputs. In [14] a RANSAC method is used to fit the features to a third-degrees Bezier spline. In [18], b-splines, i. e. curves with higher local parametrization control with respect to Bezier curves, are used. Finally, non-parametric models are based on a continuous description of the boundaries. This is a less common

Table 3.2. Model fitting methods

Method	Pros	Cons
Parametric models	Simple and fast, may provide good approximation at near distances	May not be precise
Semi-parametric models	More precise, small geometry variation with small parameters variaton	Can overfit the extracted features, leading to unrealistic results

geometric model.

The temporal integration process of a lane detection system is the module responsible to integrate the results of the current image elaboration with the knowledge of the previous results. This technique can lead to three main results:

- The improvement of the detection output accuracy, thanks to the possibility of smoothing the results over the time.
- The elaboration time reduction, that is obtained by computing a prediction of the future geometric model parameters. If in the next frame, the feature pixels are in the predicted position, the model fitting operation can be skipped, saving time.
- The error detection correction, obtained by filtering feature detection outputs that are too distant from the preceding ones, which are probably erroneous.

Commonly, the used methods are Kalman filters and Particle filters, which are typically applied to the image transformed into real world coordinate, for example using an inverse perspective mapping. An example can be found in [15], where a Kalman filter is applied.

The last module is the image to world correspondence, which is used alongside all the proposed common pipeline elements. In fact, the transformation from the pixel coordinates of the input image to 3D real world coordinates can be useful in several modules. Many algorithms in literature use the inverse perspective mapping transformation in order to project each pixel to the plane Z = 0 in the real world, removing the perspective effect (see Appendix B). This transform is also used to compute the coordinates of the resulting path in the center of the lane, in order to use this data as input in lateral dynamics control loops. However, these geometric

methods are usually based on the assumption of a flat road, leading to possible errors when a slope in encountered by the vehicle.

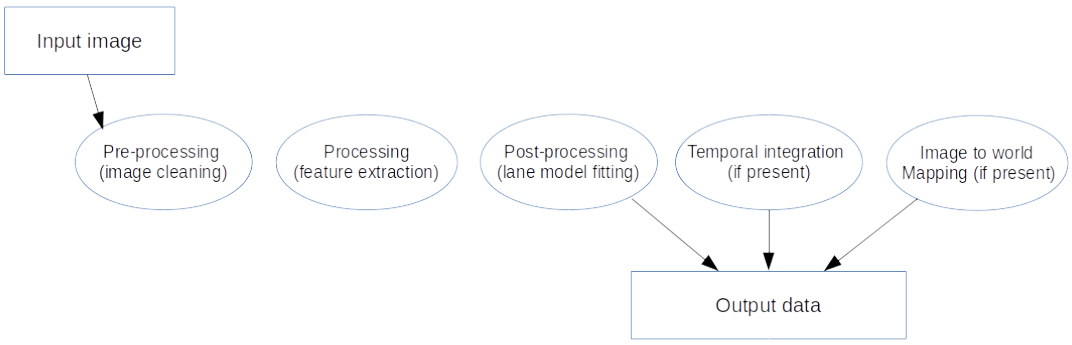

Figure 3.1. Lane detection process flow for traditional algorithms

The evolution of these traditional methods is linked to two main fields: the reliability of the system and the environment understanding improvement. The first is the problem which has the hardest level of difficulty to be solved. Being all the system proposed in literature mainly based on computer vision techniques, the robustness is a crucial point; in fact, all the possible image noises and environment condition may importantly affect the overall system performance. In order to reduce this lack of reliability, algorithms should be as generic as possible, without making strong assumptions on the scenario that the system should handle. Moreover, multiple algorithms should be executed in parallel in a weighted manner or be applied in series and, when the assumptions of the first algorithm are not met, dynamically switch to the best algorithm for every situation.

Concerning environment understanding improvement, the next steps to reach should be the capacity of detecting and distinguish multiple lanes, the capacity of detecting lane markers in far distances and the ability of understanding the road structure even in absence of clearly visible markers, like in country roads or in scenarios where road works are present.

Finally, in order to improve the algorithms ability of understanding the environment and filtering outliers at any level, different modalities than computer vision should be fused together. For example, LiDAR systems might help in lane segmentation and in road boundaries detection when lane markers are not clearly visible; GPS and IMUs data can be merged to provide additional information to the low-level

process (like signaling the current road slope for a more precise inverse perspective mapping image transformation).

3.3 Neural network-based algorithms

In the last years, an approach particularly different with respect to traditional lane detection algorithms has been proposed in literature. Instead of working at low-level with pixels and relying to an a priori knowledge of the perceptible characteristics of the lane markers and the structure of the lane, many researchers have proposed a machine learning-based solution.

As explained, the major problems that lane detection systems have to face with are represented by extremely variable environmental factors, such as daylight conditions (especially at sunrise and at sunset), artificial lights noises during the night, variable road types and structures, shadows projected onto the road surface, obstacles that can occlude the lane markers, the presence on the road of very tiny or hardly visible lane markers, bad or extremely bad weather conditions, such heavy rain, snowfalls, etc. Traditional lane detection algorithms have an approach that can be very effective in limited driving scenarios, like the highway one, if a good feature selection has been taken while designing them. However, this kind of algorithms can be importantly reduced in their effectiveness, due to their poor generalization ability with respect to the previous mentioned external condition changes.

In such a complex scenario, new approaches based on neural networks have been designed, due to the deep learning-based methods ability to perform an automatic feature selection, importantly simplifying this complex process and providing a more robust solution, able to better generalize the algorithm against extremely variable conditions coming from the environment.

The most used networks for lane detection algorithms are convolutional neural networks (CNN). This kind of networks have a design which is very effective for inputs organized into matrices, like images.

A neural network is an architecture where inputs (in the case of an image, the set of single pixels) are connected with functional blocks, called neurons. Each neuron applies an operation on its inputs (typically a weighted sum) and provides an output to other neurons (typically, the result of the operation normalized between 0 and 1). Neurons are organized in layers, divided in one input layer, one output layer and a certain number of internal layers called hidden layers. If every neuron in a layer is connected with every neuron of the preceding layer, the network is called "fully connected".

In order to train a network to provide a certain results given similar inputs, it is necessary to set properly the network parameters, i. e. the weights used in the neuron sum operation. Typically, the train operation is performed by starting from random weights and giving some inputs which expected result is known. This set of inputs is called train set. The result of the network is compared with the expected value and the weights are modified properly. This operation is called back-propagation and must be repeated until the results are acceptable. Finally, the weighted network efficiency is tested given inputs not already used during the training. The set of those inputs is called test set.

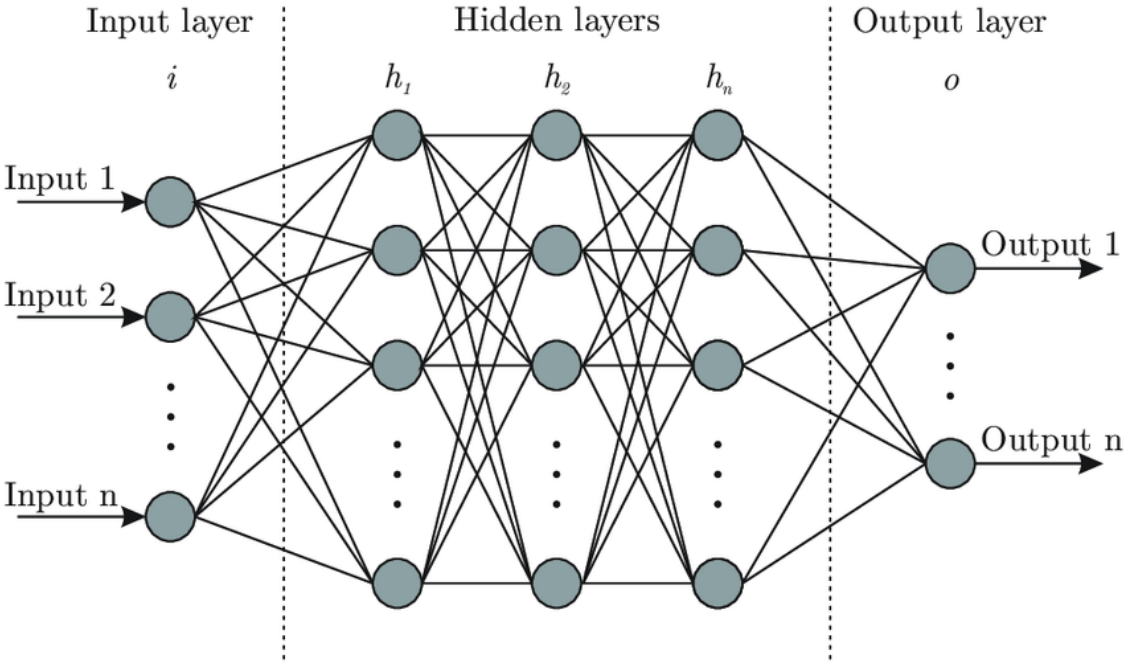

Figure 3.2. Structure of a generic neural network, from: https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051, [accessed 22 Mar, 2020]

In a convolutional neural network, there exist several types of layers, in which the most used is called convolutional layer. In a convolutional layer, neurons of each layer are organized in a 3D way, divided into width, height and depth. Instead of been connected to all the neurons of the preceding layer, each neuron is connected to a smaller portion; this allows to reduce the number of connections, thus reducing the network computation time and the number of weights needed to be set in order to train the algorithm. In case of images, each neuron provide an output applying a matrix convolution operation on a portion of the input image; neurons with the

same depth coordinate in the 3D representation work together on the whole image, looking for the same characteristic. Putting all together, a single layer takes as input a matrix (for example, an image) and provides as output a set of N values, where N is the depth of the convolutional layer, describing N characteristics of that input. Those output values are then passed to other layers (not necessarily of the convolutional type), until the final output is computed.

The convolutional neural network method particularly fits the lane detection problem, as the architecture of convolutional layer permits to check at the same time several different features of the image in order to recognize a lane structure. Moreover, the training phase permits to automatically select and set properly all these features by the assignment of the weights, optimizing the choice in complex scenarios. The general lane detection problem is thus reduced to an image binarization, recognizing if a pixel belongs to the lane or not.

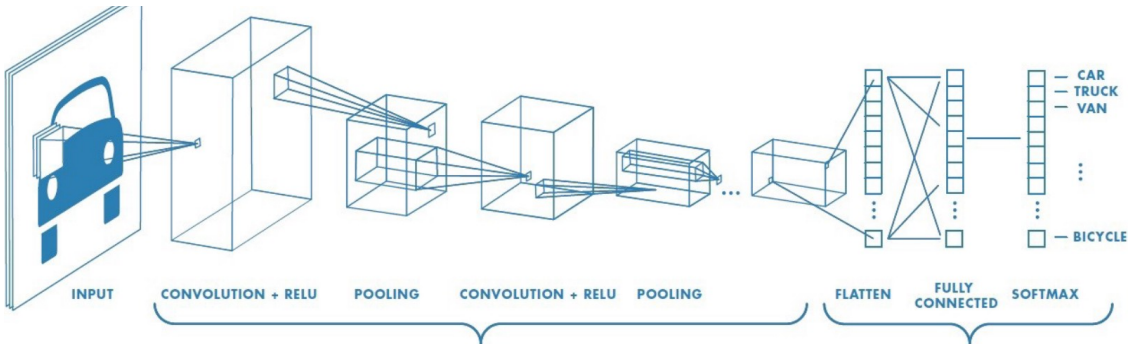

Figure 3.3. Structure of a generic convolutional neural network, from: https://miro.medium.com/max/1255/1*vkQ0hXDaQv57sALXAJquxA.jpeg, [accessed 22 Mar, 2020]

In literature, several examples of usage of convolutional neural networks to detect lanes have been presented. One of the first examples of this methods is presented in [19], where a convolutional neural network was combined with a RANSAC algorithm, starting from an edge detection operation performed on the input image. However, in this implementation, the machine learning approach is used only for image enhancement and, for performance reasons, only when the captured scene is complex, with obstacles and lane intersections.

In recent proposal, end-to-end approaches are implemented, i. e. the whole detection is performed by the convolutional neural network starting from raw images, without preliminary pre-processing operations. For examples, in [20], convolutional neural networks are applied to the highway scenario for both lane and obstacle detection. In [21] an end-to-end detection was performed both in the front view and

in a top view, in order to combine the results excluding false detections; this model is called Dual-View CNN.

Other approaches have been proposed in order to avoid specific problems encountered in the previous examples. For example, [23] proposed a Spatial CNN (SCNN) in order to avoid performance reductions due to obstacles occluding lane markings, imitating the human capacity to fill the gap in the occluded part. This goal is achieved by permitting a communication also between neurons belonging to the same convolutional layer, not only between neurons of different layers. This additional level of communication permits to achieve a smoothness in the resulting lanes, without interruptions due to obstacles.

Another problem is linked to the need of detecting multiple lanes in the same road image. The first simple solution is to design the network to perform a multi-class classification, in which each lane is mapped to a class. The main drawback of this solution is the obligation to perform the detection on a fixed number of lane, reducing the system robustness. In [24], the proposed architecture is a multi-task CNN, divided into a segmentation branch and in an embedding branch. The first performs a binary segmentation of the image into two classes: lane or background. The second, starting from the binarized image, assigns to each lane pixel a lane identification number, dividing the different lanes. In this paper, also the model fitting phase uses a deep learning approach. In fact, in order to better parameterize the resulting lane description curve, an inverse perspective mapping is applied. As mentioned, this kind of transformation relies to the fact that the road is flat, returning wrong mappings if the road has a consistent slope. To avoid this problem, a neural network is trained in order to find the sub-optimal transformation parameters, permitting to have a good perspective view in every road condition.

A similar approach to lane detection with multi-task CNNs is presented in [25], where we have: a binary segmentation branch, dividing between lane pixels and background; another binary segmentation branch, dividing between drivable areas and background; a lane point regression branch, to estimate the lane points value; a lane embedding branch, to subdivide the different lanes; a clustering branch, to process the output of the embedding branch.

Despite the advantages that deep learning-based approach may give, there exist some problematic factors that have to be handled. The main problem is due to the complexity of the convolutional neural network model which may be incompatible with real time elaboration requirements needed in control loop algorithms. In fact, the number of the neurons and the type of operations they are applying on the input data (i. e. convolution of matrices) lead to a high computational cost, that can be reduced implementing the algorithm on a specific hardware (for example, a

specific GPU) or by lightning the network. In this case, there exists a trade-off between the network performances in terms of correct detections and computational time.

Finally, as every deep learning-based method, lane detection systems based on convolutional neural networks need a huge amount of data to use in training and testing phase. Moreover, these images must be as diversified as possible, due to the fact that the system may encounter very dynamic situations and different road structures, so it must be robust in every case.

Chapter 4

The GOLD system

4.1 Bertozzi and Broggi's algorithm

GOLD (Generic Obstacle and Lane Detection system) is a stereo vision-based hardware and software architecture proposed in 1998 by Massimo Bertozzi and Alberto Broggi for lane detection and obstacle detection tasks ([13]). The two tasks are addressed at the same time thanks to a parallel hardware architecture, obtaining the input images from the same source. The lane detection algorithm is built on a pattern-matching technique, based on the assumption of the presence of bright lane markers above a dark road; the object detection algorithm is based on the determination of the free-space areas in front of the vehicle, without any 3D world reconstruction. These two algorithms are based on a common structure: an image transformation is firstly applied to inputs, then a low-level parallel processing is performed and finally the resulting data are inversely transformed to original image space. In the next sections, only the lane detection process will be treated, being the base of the thesis algorithm implementation.

4.1.1 Image transformation

Due to its structure, an image is mapped in a computing system as a matrix of pixels. This representation allows the low-level processing to be efficiently executed by single instruction multiple data (SIMD) systems, which are systems performing the same operation in a parallel way to multiple input data. This approach is very efficient in case of uncorrelated data or when applying operations considering the input as such; for example, noise reduction algorithms can be applied to images in such a way.

Conversely, more sophisticated filters and algorithms may require knowing the correlations among data. For example, in the case of lane markings recognition, the processing algorithm must be aware of the presence of a perspective effect, causing

a different lane marking width between near and far pixels. To avoid this problem, both the lane detection and the object detection algorithms presented in the GOLD system are based on the image warping approach, namely the transformation of the input image before the system elaborates it.

In particular, the technique applied in the GOLD system is the Inverse Perspective Mapping (IPM). The Inverse Perspective Mapping is a geometrical transformation which allows to remove the perspective effect from the input images, with the assumption of a flat road. This result is obtained by mapping each image pixel in a new two-dimensional space, creating a new two-dimensional pixel array.

Firstly, two Euclidean spaces are defined:

- The 3D world space $W = \{(x, y, z)\}$, representing the coordinate of each pixel in the real world, where the z axis is the vertical axis.

- The 2D image space $I = \{(u, v)\}$, representing the coordinate of each pixel on the image 2D array.

The input image coming from the camera belongs to the second space, the image obtained after the IPM belongs to the 3D world space and it is mapped to a 2D plane representation, setting the vertical component z to zero (see 4.1). The mapping

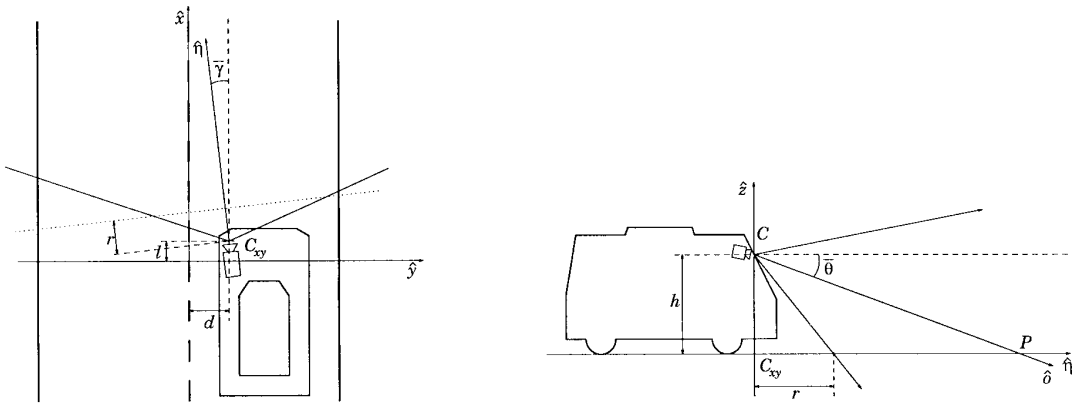

Figure 4.1. World reference in [13]

transformation can be expressed in different ways. Bertozzi and Broggi define it

as:

$$\begin{cases} x(u,v) = h \times \cot\left[(\theta-\alpha) + u\dfrac{2\alpha}{n-1}\right] \times \cos\left[(\gamma-\alpha) + v\dfrac{2\alpha}{n-1}\right] + l \\ y(u,v) = h \times \cot\left[(\theta-\alpha) + u\dfrac{2\alpha}{n-1}\right] \times \sin\left[(\gamma-\alpha) + v\dfrac{2\alpha}{n-1}\right] + d \\ z = 0 \end{cases}$$

where the camera position is defined in W space as $C = (l, h, d)$, namely the lateral distance, height and longitudinal distance of the camera with respect to the origin of the world coordinate reference; the camera direction is defined by the camera pitch angle γ and the camera yaw angle θ; the camera characteristics are defined by the camera angular aperture equal to $2 \times \alpha$ and by the image resolution $n \times n$. The dual transformation, from 3D world coordinates to 2D pixel coordinates is described as:

$$\begin{cases} u(x,y,z=0) = \dfrac{\arctan\left[\dfrac{h\sin\gamma(x,y,0)}{y-d}\right] - (\theta-\alpha)}{\dfrac{2\alpha}{n-1}} \\ v(x,y,z=0) = \dfrac{\arctan\left[\dfrac{y-d}{x-l}\right] - (\gamma-\alpha)}{\dfrac{2\alpha}{n-1}} \end{cases}$$

Another way to represent the transformation is by using matrices ([27]). In this case, the 3D world coordinates are described in a space $W = \{X_w, Y_w, Z_w\}$, where the Y_w axis is the vertical axis. The plane to which the road points belong is defined by setting $Y_w = 0$, assuming the road as flat. The image coordinates, belonging to the space $I = \{u, v\}$, are mapped from the 3D coordinate system as:

$$R = R_z(\alpha)R_y(\beta)R_x(\gamma) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma & 0 \\ 0 & \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.1)$$

$$T = \begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\dfrac{h}{\sin\beta} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

$$K = \begin{bmatrix} f_x \times ku & s & u_0 & 0 \\ 0 & f_y \times kv & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.3}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = KRT \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{4.4}$$

where the R matrix describes the overall rigid rotation of the camera with respect to the 3D world axis, obtained multiplying the rotation matrices of the yaw (α), the pitch (β) and the roll (γ) camera angles; the T matrix is the matrix describing the rigid translation of the camera position with respect to the 3D world reference point, in this case having translation offset equal to the height in which the camera is posed (h) divided by the sin of the pitch (β) angle; the K matrix represents the camera intrinsic parameters, such as the focal length (f), the optical center coordinate (u_0, v_0), the pixel skew parameter (s) and the expected ratio scaling factor of the resulting pixels (ku, kv). Having in this case the Y_w axis as vertical axis, to obtain the road mapping, Y_w is put equal to zero; the overall matrix transformation becomes:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = KRT \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{13} & p_{14} \\ p_{21} & p_{23} & p_{24} \\ p_{31} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X_w \\ Z_w \\ 1 \end{bmatrix} \tag{4.5}$$

The dual transformation, to re-map the image coordinates into the 3D world coordinates is defined as:

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = (KRT)^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{4.6}$$

For a more detailed description of the matrix representation of the IPM transformation, see Appendix B.

4.1.2 Processing algorithm

The lane markers detection algorithm takes as input a stream of images on which the inverse perspective mapping transform has already been applied. It is based on an a priori assumption: in the 3D space $W = \{(x, y, z = 0)\}$, the lane markers are represented as almost-straight lines composed by bright pixels surrounded by darker pixels representing the road. Moreover, another assumption is the fact that the road is considered flat.

Thus, the first phase of the algorithm is focused on detecting every bright-dark and dark-bright variation of horizontally contiguous pixels value. Every pixel brightness quantity ($b(u, v)$) is compared with the left and the right pixel at distance m, and then it is remapped to a 2D array R, such that:

$$r(u,v) = \begin{cases} d_{m+}(u,v) + d_{m-}(u,v) & \text{if } (d_{m+}(u,v) > 0) \wedge (d_{m-}(u,v) > 0) \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

where

$$\begin{aligned} d_{m+}(u,v) &= b(u,v) - b(u,v+m) \\ d_{m-}(u,v) &= b(u,v) - b(u,v-m) \end{aligned} \tag{4.8}$$

In this way, the values of the R matrix are relative to the brightness difference between pixels and not to an absolute threshold value, in order to be more robust to possible homogeneous noise, for example shadows. However, shadows and obstacles in the image view can cause small brightness variation between neighbor pixels; to avoid erroneous situations, the R matrix is mapped into a control matrix applying a certain number of iterations of a geodesic morphological dilatation. This kind of filter gives to a pixel the maximum value of all the pixels near to it in vertical and horizontal directions. In this way, the value of the pixels belonging to the lane marker increases, except in the direction in which the pixel values are zero.

Finally, the resulting image is obtained through a two-level binarization, given an adaptive threshold:

$$t(u,v) = \begin{cases} 1 & \text{if } e(u,v) \geq \dfrac{m(u,v)}{k} \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

where $e(u, v)$ is the image enhanced by the geodesic morphological dilatation, $m(u, v)$ is the maximum value among a given neighborhood $c \times c$ of the given (u, v) point and k is a constant. After this phase of the algorithm, the result is a

binary image in which pixels belonging to the road are set to zero and the remaining pixels belong to the lane markers.

The next phase consists into the determination of the geometry of the road. Firstly, the image is scanned row-by-row and every two nonzero non-contiguous pixels are considered in pairs. For each pair, a c_i value, representing the coordinate of the road medial-axis, and a w_i value, representing the lane width, are computed. These data are then filtered, and a histogram is built from the remaining pairs. The value of road width w that has a peak in the histogram is considered as the actual lane width value and, subsequently, all the pairs with a road width close to w are considered. The center line of the road is finally reconstructed using the c_i values of the selected pairs.

The last step of the algorithm consists into apply the dual inverse perspective mapping transformation to the image with the detected lane markers and superimpose it to the input original image.

4.2 Results and conclusions on GOLD algorithm

The hardware architecture of the original GOLD system was divided in two main devices:

- The PAPRICA system, a low-cost parallel processing system implemented on a FPGA, composed by 256 processing elements working by using the SIMD approach.
- The Host workstation, a SPARC embedded processor connected with the previous system.

The first section of the system is responsible to interface with the input cameras through an acquisition system and with the output screen through a frame buffer. It is also composed by a main memory device to store the inputs and the elaborated images (up to 8 MB), an image re-mapper module, a 16×16 processor matrix and a control unit. The PAPRICA system executes the low-level part of both the lane detection and the obstacle detection tasks, obtaining some intermediate data that will be sent to the host computer to continue the elaboration. In the case of the lane detection algorithm, the result is represented by the image matrix in which pixels belonging to the road are set to zero and the remaining pixels belong to the lane markers.

As explained, the first part of the algorithm takes an important advantage in terms on performance thanks to the parallel execution of the same instruction on all the

128×128-byte image matrix, having each processing element able to apply the elaboration on 64 bytes a time. The inverse perspective mapping is also implemented on the PAPRICA system in a dedicated device, which is able to map 512×256 image matrices coming from cameras to 128×128 matrices to give as input to the system in 3 milliseconds.

The host workstation, instead, is responsible to take as input the matrices coming from the obstacle and lane detection low-level algorithms and to implement the medium-level part. In the case of the lane detection, in this level the road geometry determination is implemented. This computation is executed in a different hardware device with respect the PAPRICA system, with a different architecture: in fact, being the elaboration of the medium-level based on highly connected data, the advantage represented by a parallel execution on several low-cost processors is not satisfying as it was in the low-level section. The elaboration on this device is based on software, while in the previous case a hardware solution was used.

The performances in terms of timing on the original system were:

- The data acquisition from the two cameras and the output results took 20 milliseconds.
- The remapping of the two 512×256 images to two 128×128 matrices took 6 milliseconds in total.
- The obstacle detection pre-processing on PAPRICA, taking as input two 128×128 matrices and giving as output a 128×128 matrix, took 25 milliseconds.
- The lane detection pre-processing on PAPRICA, taking as input a 128×128 matrix and giving as output a 128×128 matrix, took 34 milliseconds.
- The obstacle detection part on the host workstation took from 20 to 30 milliseconds
- The lane detection part on the host workstation took about 30 milliseconds.

Considering the delays due to the data exchange between the PAPRICA system and the host workstation, the total amount of latency from the moment when the images are grabbed by cameras to the moment when the output is sent to the screen frame buffer is about 100 milliseconds. Thus, the overall GOLD system runs at 10 Hz.

In terms of results accurateness, the GOLD system reached the 95% of successful lane detection in 1998, among a total of 3000 km traveled with the system activated. The tests were taken in extra-urban roads and freeways under different traffic and illumination condition, with a speed up to 80 km/h.

Part II

The GOLD-based lane detector implementation

Chapter 5

The software implementation

5.1 Introduction to the system

The main content of this thesis is the implementation of a Lane Detection algorithm based on the GOLD system previously presented. This application receives a stream of images from a camera device, from a file or from a network stream. The output is composed by a stream of images that will be displayed into a screen by a HDMI connection and by geometrical information provided through a shared portion of the RAM memory. Eventually, the output stream can be saved into an AVI video file, for testing or results storage purposes.

This GOLD-based application is a traditional lane detection algorithm, following the architectural scheme presented in chapter 3 (see fig. 3.1). It is logically composed by:

- A pre-processing block, responsible to reduce the input image into a smaller pixel matrix, transforming the input through an inverse-perspective mapping.
- A low-level processing block, extracting pixels belonging to lane markers.
- A post-processing block, computing the pixels belonging to the center of the lane starting from the extracted lane markers.
- An image to world correspondence block, computing and applying the transformation between original image space and inverse-perspective mapping space.

The implementation is based on the GOLD system lane detection algorithm. In fact, an inverse-perspective mapping is performed before elaborating the input images and the reverse transformation is applied to provide visual results. Moreover,

the low-level elaboration approach is similar, looking for dark-to-bright ad bright-to-dark bumps in the pixel brightness. However, the filtering and the model fitting processes are handled differently.

The algorithm runs on a custom distribution of an Embedded Linux operating system. The system is based on the Poky distribution and it was built using the Yocto project tools (see Appendix A). The distribution has been developed including the necessary software modules to run the final application and to permit the debugging and testing phase. In particular, the Qt5 library has been added to implement the graphical user interface (GUI), the OpenCV library was used for low-level computations and OpenSSH module was used to remotely deploy, debug and test the application.

The algorithm has been written using the C++ language and was compiled in order to exploits the ARM extension for single instruction multiple data (SIMD) called NEON. Thanks to this hardware component it has been possible to considerably increment the overall system performances. It takes as inputs Full HD frames, thus having a resolution of 1920x1080 pixels at rates of either 30 or 24 frames per second.

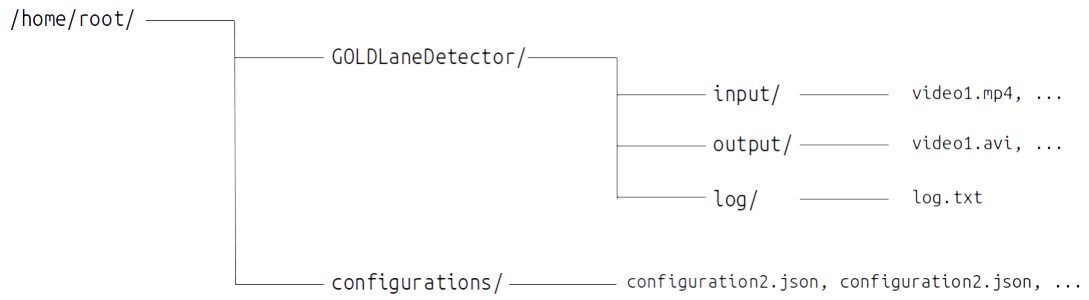

Figure 5.1. Directories structure of the target system

In order to properly work, the system needs a specific directories configuration inside the file system. When started, the application checks the existence of the folder `/home/root/GOLDLaneDetector` and its three sub-folders `input`, `output` and `log`. If the folders do not exist, the program creates them. Under the `input` folder, the input video files used for testing are contained; the `output` folder is where the optional output video files are saved; the `log` folder is where a textual log file is created and updated with every useful information. Finally, a folder named

`configurations` has been created and placed under the `/home/root/` directory, to contain the configuration JSON files used during the testing phase. This folder is not needed by the system to correctly run but is useful to organize files.

The application has been developed using the QtCreator IDE for the graphical user interface part and for the final modules assembling, while the lane detection part has been developed and debugged using the Eclipse IDE and then trasposed into the QtCreator project.

5.2 The application

The thesis application is composed by two logical levels: a graphical user interface and a lane detector processing module. Both the two software entities interact with the operating system through the file system, while the processing module can also interact with camera devices, shared memory streams and network streams.

The system is linked with two main peripherals, which are used during the standard execution of the program:

- The USB peripheral, from which is theoretically connected a camera device providing the image stream.
- The HDMI peripheral, providing the graphical output produced by the GUI architectural level containing the final results overlaid to the original frames.

Moreover, the system exploits a shared memory allocation to provide lane geometric information extracted from the image stream, which are useful for control loop algorithms such as Lane Keeping Assist.

5.2.1 Graphical user interface module

The graphical user interface software module is developed using the Qt5 framework. Qt is an open source toolkit for graphical user interfaces developing, based on C++ language. This framework provides an additional level of abstraction with respect to native applications: in fact, its code provides transparent interfaces which can be reusable within different operating system platforms such as Linux, Windows, Android and embedded distributions of Linux. This mechanism can be particularly useful when developing applications like the one proposed in this thesis, abstracting the low-level description of the hardware and the low-level operating system interfaces and moving all the complexity to the library itself.

This module is responsible to retrieve the configuration information, to start the

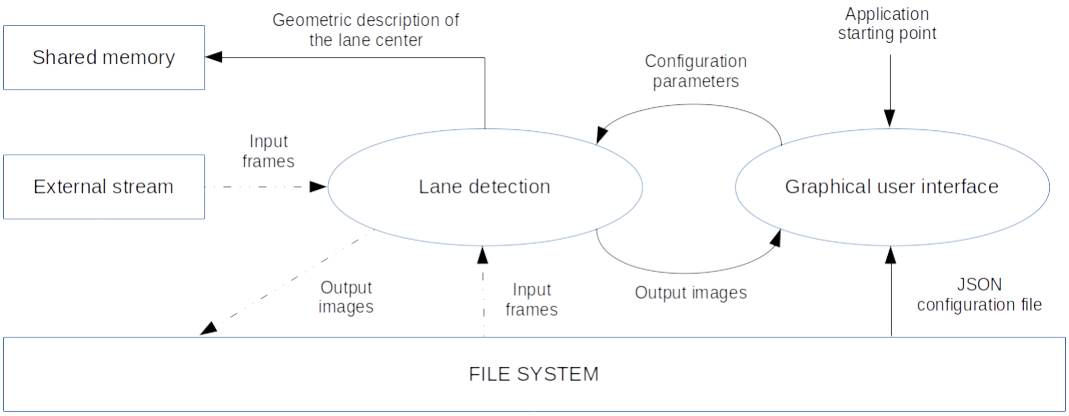

Figure 5.2. Application architecture: communication scheme between GUI and Lane Detection modules

lane detection process and to visualize the output stream. The configurations parameters are loaded into the application through a JSON file located in the file system, which is parsed by exploiting the Qt JSON parsing module (QJsonDocument). The configuration file contains necessary information to correctly set up the application, as:

- The input file name, if the system is configured to retrieve the image stream from a file.
- The input USB stream device name, if the input stream comes from a camera device.
- The input GStreamer pipeline, if the stream is provided by other sources as the network or a shared memory;
- The output file name, if the system is configured to save the output stream into a file.
- The camera intrinsic parameters, as the vertical and horizontal focal lengths and the image center coordinates in pixels, necessary for inverse-perspective mapping.
- The camera extrinsic parameters, as the rotation angles and the translation distances where the camera is posed with respect to the road 3D reference, necessary for inverse-perspective mapping.

- The low-level algorithm parameters needed during the processing phase.
- Other parameters, like the target frame per seconds value, used for validation tests, a scale factor, introduced for optimization purposes and the degree of the polynomial geometrically describing the lane center.

When those configuration parameters are gathered, the graphical module starts the lane detection process. Practically, the worker paradigm is exploited: in this paradigm, the user interface creates a new thread and loads a "worker" object in it; the worker is an object which is previously initialized, providing an entry-point function, run inside the secondary thread. In this way, the user interface part of the application can proceed its execution without blocking and eventually obtain data from the worker, updating the graphical aspect. Meanwhile, the worker object is running, executing a processing operation and periodically providing data to the user interface.

In this application, the worker is initialized with the input parameters obtained from the JSON file. Then, it executes the lane detection computation, periodically sending the output frame to the user interface. When this happens, the frame is displayed on the screen. When the working object execution ends or an error occurs, the application receives this information, deletes the secondary thread and exits, eventually signaling the error.

The graphical interface, instead, simply provides an image visualizer, in which the output frames are displayed whenever set available by the worker. Those outputs are composed by an image with lane markers and center of lane pixels colored in green, superimposed onto the original input frame.

5.2.2 Lane detection processing module

In parallel to the previous presented module, the lane detection process is executed. It is run inside the secondary thread allocated by the user interface level and it is contained into the worker object. This is the part of the application where the GOLD-based algorithm is actually performed.

Apart from the output stream display operation, which is carried by the graphical user interface, the application interfaces with the system are handled in this module. In fact, depending on the configuration parameters set in initialization phase, the input stream is captured from the file system, from a camera device or from another source, like the network or a shared memory. Moreover, the geometric output data are sent to whichever process that needs them through a shared memory and, if defined, the output stream can be saved as an AVI file by this module.

The input data stream is handled by using the OpenCV library, that is an open-source project for computer vision application development. The library permits to use the same interface for extracting frames from the stream, independently from the source. OpenCV itself exploits other libraries as back-end for interfacing with videos in file system, USB devices and other types of streaming sources. The most used are GStreamer and FFmpeg, software platforms for the audio and video management.

Logically, the lane detection module is subdivided into three procedural steps:

- The pre-processing: in this step, every frame coming from the image stream is transformed by using the inverse-perspective mapping. The result is a smaller image, containing the road pixels only, in which the perspective effect is removed, and lane markers are mapped to parallel lines, in the assumption of a flat straight road.

- The processing: this is the step in which, starting from the perspective-free image, the actual algorithm is performed, obtaining an image where pixels belonging to lane markers and to the lane center are set to specific values (255, in decimal representation), while all other pixels are set to 0. The geometric description of the lane center is also computed.

- The post-processing: in this step, the processed image is re-mapped to the original coordinates system using the reverse transformation of the one applied in the first step. Then, the result is superimposed onto the original frame and provided externally.

This module should be as fast as possible in terms of computing time, being that the application should work in real-time. When a Full HD frame is retrieved as input, the result must be computed before the arrival of the next frame. In case of a stream with rate equal to 30 frames per second, the time interval between each frame is approximately 33.33 milliseconds, with a stream at 24 frames per seconds, the interval is 41.66 milliseconds.

However, it is important to underline that the embedded Linux distribution on which the application is executed is not a real-time operating system, therefore there is not any hardware/software assurance that a frame, whatever is the mean time to complete the processing operation, can be elaborated before the exceeding of a given threshold time. With the "real-time" term it is meant that the avarage time needed by the application to process a frame is less than the stream inter-frame time interval.

The first implementation of the module was structured to execute the three computing steps sequentially. However, the overall execution time considerably exceeded the time limit to guarantee the end of the computation of a frame before the next is given as input. To importantly improve the frame computing rate, the module has been split into five different threads, exploiting the multiprocessor hardware architecture.

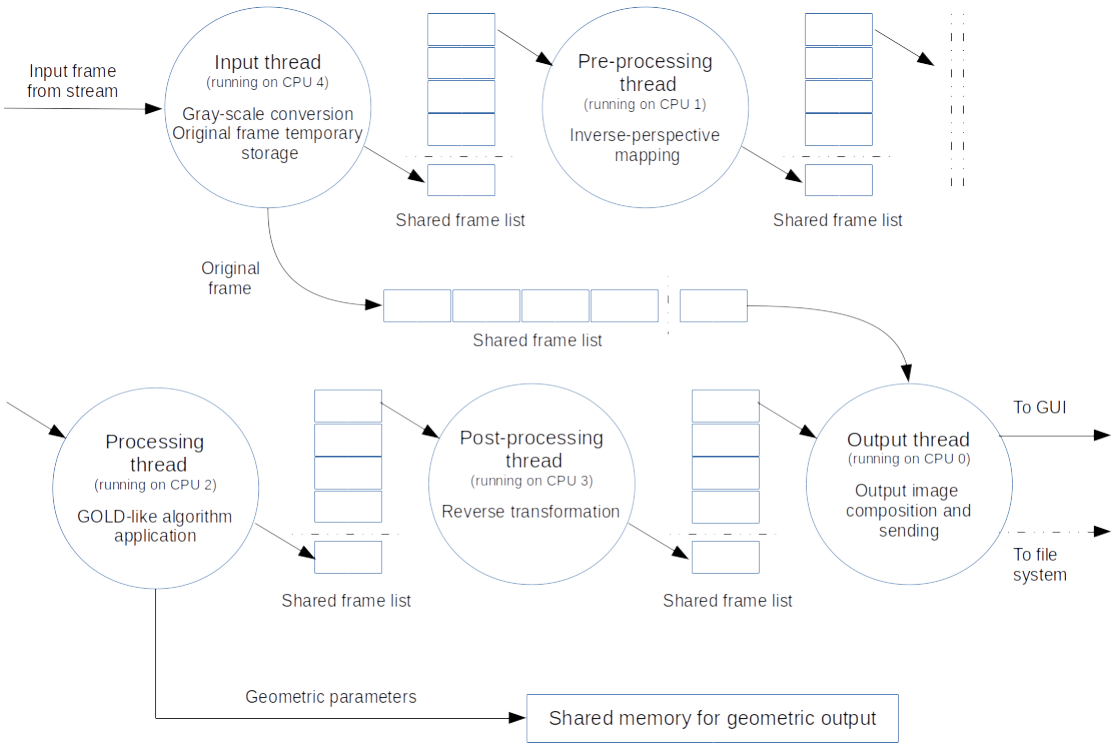

Figure 5.3. Application architecture: Lane Detection module's threads structure

For this purpose, the C pthread library has been used. Moreover, to improve the overall system performances, the CPU affinity mechanism has been exploited: through the pthread library interface, in fact, it is possible to define in which CPU cores the thread will be executed. In this implementation, each thread has been assigned to one CPU core only, in order to reduce the time lost when the thread is waiting to be scheduled by the operating system. Moreover, the scheduling policy of all the application threads has been set to "Round Robin", that is a scheduling algorithm capable to avoid the starvation effect between threads.

The module architecture has been then structured as a pipeline: in a pipeline,

the elaboration is split into a sequence of a given number of sub-modules, each of them interacting with the previous and the next one in the sequence by retrieving and providing inputs and outputs, respectively. At any time, every sub-module elaborates a different input in a parallel way. The time to elaborate a single input is the same as it would be if all the operations were executed sequentially, but the throughput, i. e. the rate of completed elaboration, is ideally the overall execution time divided by the number of sub-modules. Actually, if the time needed by each sub-module to complete the operation is different, the overall throughput depends on the elaboration time of the slowest one.

Using the pipeline structure, the lane detector module has become able to reach a throughput compatible with the real-time requirement. The actual delay from the time of the input frame retrieving to the time when the output is provided to the user interface has remained the same.

Every thread is responsible of a specific operation, taking different time amounts to complete. The first thread, namely the input thread, is responsible to retrieve each frame from the stream, to convert it from BGR color-space (Blue-Green-Red) to gray-scale (passing from a pixel representation based on three bytes to one byte only), to provide the gray-scale matrix to the next thread and to temporally store the original colored image, which will be necessary when composing the final output. The transformation of the input from a BGR image to a one-byte per pixels matrix permits the reduce the computation time of the next blocks and remove redundant information, being that the core algorithm is only based on pixel brightness values, not on color-based information.

Figure 5.4. Example of the gray-scale conversion handled by the input thread

The second thread, namely the pre-processing thread, is responsible of the pre-processing logical step of the lane detection algorithm, transforming the gray-scale pixels matrix by applying an inverse-perspective mapping (see Appendix B). In this

thesis, the transformation is applied by means of the matrix operation described in chapter 4.1.1. The transformation matrix is computed starting from the extrinsic matrix, describing the position and orientation of the camera in the space, and from the intrinsic matrix, describing the camera characteristics. The necessary parameters are provided by the graphical user interface module at the initialization. The overall transformation matrix is computed once at the beginning of the application execution and then is applied to every frame of size 1920×1080, in order to obtain a mapped representation of the road of size 150×746 bytes.

Figure 5.5. Example of the inverse-perspective mapping handled by the pre-processing thread

The transformation was originally applied using the OpenCV library functions, but the results were not acceptable in terms of computation time, exceeding the time limit to afford a real-time elaboration. The solution was to exploit a tool contained into the MATLAB environment, called MATLAB Coder. This tool permits to automatically translate a MATLAB script into a generated code written in C, enabling important optimizations. Thus, one script computing the transformation matrix and one script applying the mapping were written using the functions provided by a special software module for computer vision, called Computer Vision

Toolbox. Then, the automatic code generation was performed, obtaining two sets of C functions able to obtain the same results of the scripts. Using this solution, the execution time of this step considerably decreased.

The third thread, namely the processing thread, is the one performing the actual low-level processing for feature extraction. The algorithm is based on the GOLD lane detection process in the first part of the elaboration, but there exist considerable differences in the remaining parts. As the original algorithm, the extracted features are the lane markers, which are assumed to be described as bright elements over a darker background. Moreover, being that in the previous thread the perspective effect has been removed, it is also possible to take the assumption of a parallel linear description. Therefore, in this step, the algorithm looks for dark-to-bright and bright-to-dark bumps inside the mapped image. A temporary pixel matrix R_0, with the same size of the mapped image, is created, such that:

$$R_0(i,j) = \begin{cases} d_{m+}(i,j) + d_{m-}(i,j) & \text{if } (d_{m+}(i,j) > th) \wedge (d_{m-}(i,j) > th) \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

where

$$\begin{aligned} d_{m+}(i,j) &= image(i,j) - image(i,j+m) \\ d_{m-}(i,j) &= image(i,j) - image(i,j-m) \end{aligned} \tag{5.2}$$

and where m is the distance in pixel from the current pixel where to look for a darker value and th is a threshold given as parameter. More the current pixel $image(i,j)$ has a brightness value higher than the value to the left and the value to the right, higher would be the value of R_0 in that point.

The next step after having obtained the temporary R_0 matrix is to filter possible outliers, thus pixels having higher bright values with respect to the neighborhood but not actually belonging to the lane markers. Firstly, a median blur filter is applied to delete "salt and pepper noise", i. e. isolated pixels scoring high brightness values. Practically, this operation substitutes every pixel with the median value among a squared kernel of size N pixels, centered in the current elaborated pixel. In this implementation, a special OpenCV library function is used to perform this filtering, setting the N value to 3.

Then, every pixel in the resulting matrix is compared with a minimum threshold value (th_{min}): if the pixel value is less than the threshold, then its value is set to 0. Finally, as last filtering step, the algorithm must recognize pixels belonging to lane markers only. Thanks to the assumption about the markers' representation

as straight parallel lines, it is possible to define a pixel as part of a lane marker if, in accordance to the first part of the elaboration, pixels on top or on bottom still have a similar value. Practically, the resulting matrix is defined as:

$$R_1(i,j) = \begin{cases} 150 & \text{if } R_0(i,j)! = 0 \vee (R_0(i - m_{filter}, j)! = 0 \wedge R_0(i + m_{filter}, j)! = 0) \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

where m_{filter} is the distance in pixel from the current pixel where to look for a darker value.

Then, a last filtering operations is applied. Firstly, each matrix column is scanned, creating a histogram having the column index as x-axis and the total amount of non-zero pixels belonging to the given column as y-axis. Then, starting from the center of the histogram, the left and the right peaks are detected. Being the assumption of a straight-line representation of the lane markers, the indices corresponding to peaks should describe the position of the markers in the matrix. Every non-zero pixel belonging to the columns described by the two peaks and to the contiguous columns are maintained. A column is considered contiguous to the peak if there is not a zero in the histogram between the given column and the peak.

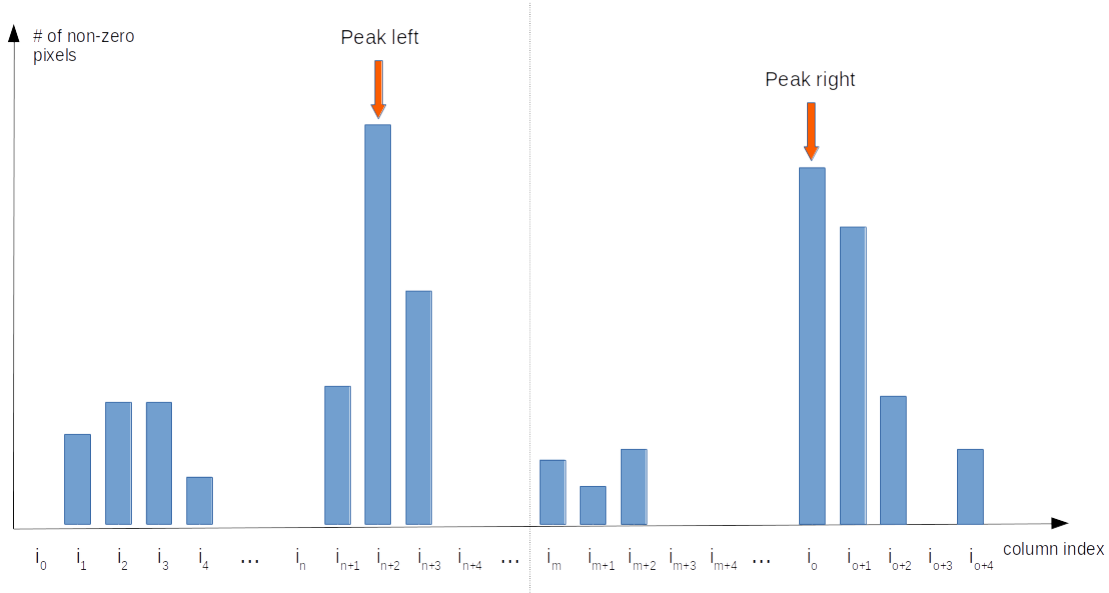

Figure 5.6. Graphical representation of the histogram describing the number of non-zero pixels for each column of the R_1

At the end of the elaboration, the result is a binary matrix where pixel belonging to lane markers are set to a defined value greater than 0 (namely, 255) and road

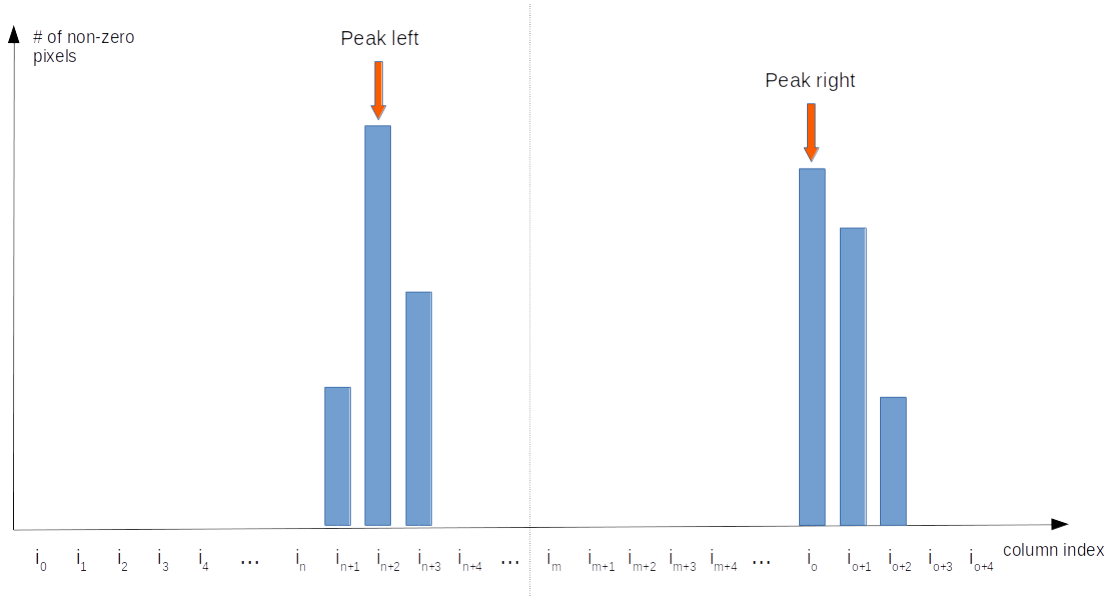

Figure 5.7. Graphical representation of the filtering operation performed on the histogram

pixels are set to 0.

In order to improve the image processing, alongside with the lane boundaries detection a lane center detection process is performed. Firstly, the filtered matrix is scanned row-by-row in order to find the pixel corresponding to the center of the lane for each row: at any row, starting from its center, the algorithm searches for the first non-zero pixel both in left and right directions. If a non-zero pixel is found in both directions, the medium value between the two pixels column indexes is computed and added to the lane center points list.

At the end of the elaboration, a list of points belonging to the lane center is obtained, having the image space as coordinates reference. The next step consists in translating those points into a world reference description. Coherently with the preprocessing phase, the MATLAB Computer Vision Toolbox is exploited. Starting from a MATLAB script, a proper function has been generated in order to transform the list of points from image coordinates to real world coordinates.

Then, the *polyfit* MATLAB function is used to fit the real-world points into a geometric description of a polynomial of degree n. The n geometric parameters of the polynomial are then provided externally through a shared memory. Finally, the obtained polynomial points are translated back to the image reference and highlighted into the processed matrix, to provide a visual feedback of the model fitting

results.

This fitting operation is based on a very simplified model, in which the lane is considered straight, and it could be fitted into a linear model description (the polynomial degree is set to 1 by default). This data has to be intended as an additional information alongside the lane markers description, which is considerably more accurate.

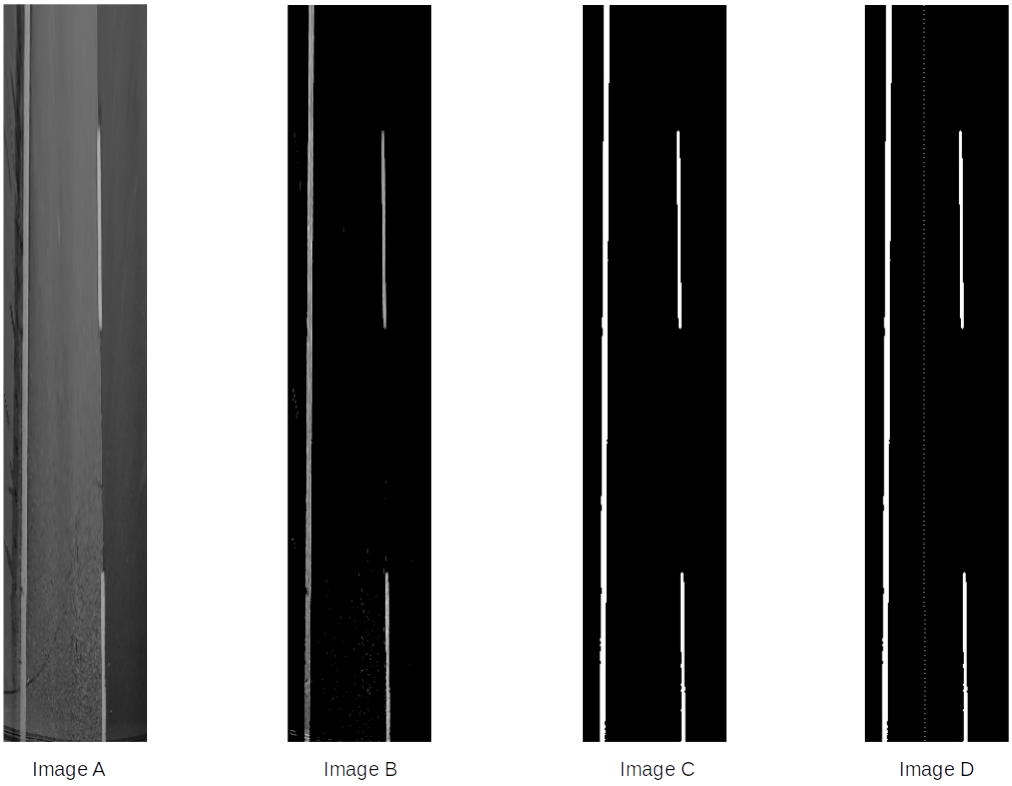

Figure 5.8. Example of the GOLD-like algorithm application handled by the processing thread. Given Image A as the processing input, Image B represents the first elaboration step, Image C is the filtering result, Image D is the output of the model fitting operation.

The final result elaborated by the processing thread is a 150×746 matrix with pixels belonging to the center of the lane and to each lane marker set to 255 and all other pixels set to 0. The whole processing is performed exploiting the OpenCV interfaces, that guarantee a fast pixel-per-pixel elaboration.

The fourth thread, namely the post-processing thread, is responsible to re-map the

image elaborated by the processing thread to the original coordinate references.

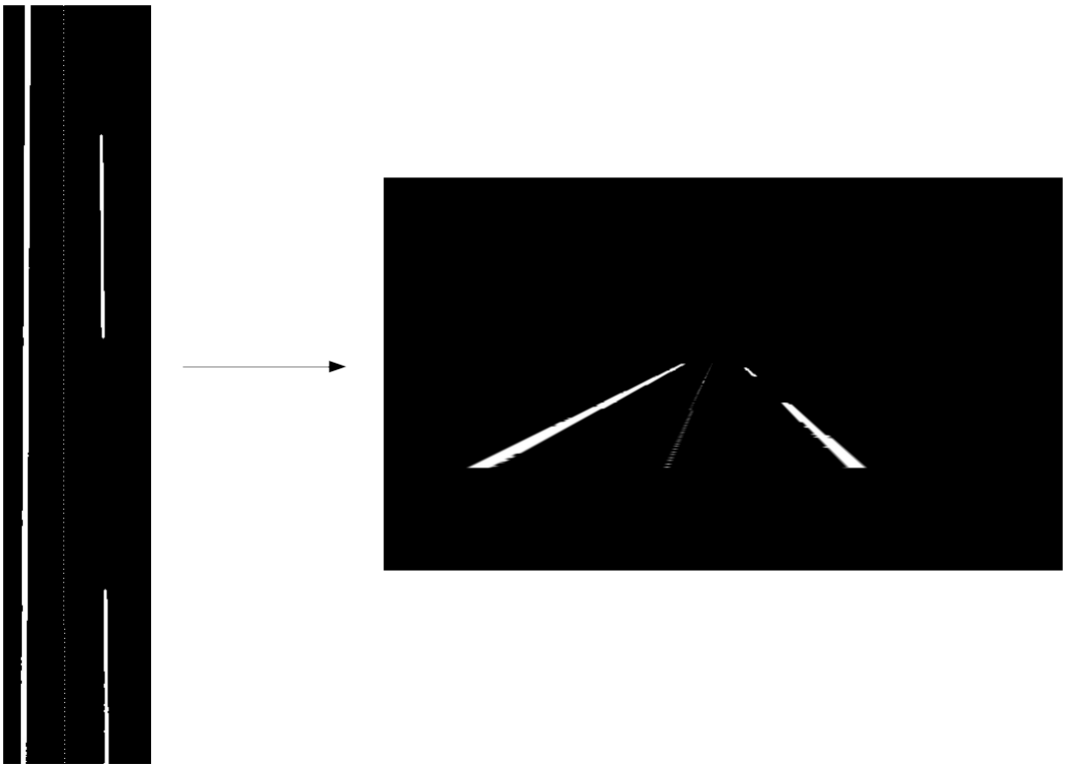

Figure 5.9. Example of the reverse transformation handled by the post-processing thread

In a similar way to the pre-processing and the processing threads, in this block functions generated from MATLAB are used, being that OpenCV library was not fast enough to handle this kind of elaboration. Practically, the transformation matrix used during the inverse-perspective mapping is inverted and then applied to the processed image. However, the reverse process to re-map the 150×746 image to the original 1920×1080 size took more time with respect to the pre-processing one, exceeding the time limit for the real-time computation. The adopted solution was to perform the re-mapping operation to an image scaled by a given factor. For example, setting the scaling factor to 3, given the input image of size 150×746, the result is an image of size 640×360, which leads to an important reduction of

the computing time. The matrix transformation must be modified such that:

$$T_{inv} = T^{-1} \begin{bmatrix} \frac{1}{s} & 0 & 0 \\ 0 & \frac{1}{s} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

where T is the transformation matrix used during the pre-processing and s is the scaling factor.

The re-mapping transformation matrix is computed once during the initialization phase and then applied to every image during post-processing. Then, in order to bring the image back to the original frame size, the OpenCV resize function is used.

Figure 5.10. Example of the output frame composition handled by the output thread

The last thread, namely the output thread, has been introduced to handle the resulting frames. It is responsible to superimpose the post-processed image onto the original input frame to provide visual feedback of the lane detection process, to send the output image to the graphical user interface module and, if requested by configuration parameters, to save the elaborated frames into a AVI video on the file system.

The first operation is performed though the Qt signal/slot mechanism. This mechanism is based on signals, that can be considered as events triggered by a Qt object, and slots, which are a sort of callback functions. Through the Qt framework it is possible to connect signals to slots, in order to obtain the execution of the slot function whenever the signal event is raised. In this case, the signal (containing the output image) is emitted by the worker object whenever the output thread is ready to send the result. In the graphical user interface module, the correspondent slot is responsible to grab the sent image and visualize it onto the screen. The

eventual file saving operation, instead, is performed through the OpenCV interface.

The last operation handled by the post-processing thread is the final output image composition: the processed image (re-mapped to the original coordinates reference and scaled to the original size) is superimposed to the original input frame. Being that the original image is represented with the BGR color representation, the post-processed frame is firstly converted into a BGR image, setting the green component equal to the gray-scale pixel value and the other two components to 0.

Every thread, in order to properly work, needs to exchange data with the previous and the next thread in the pipeline. The communication is handled through an implementation of the producer-consumer pattern. This pattern is composed by two processes sharing a memory buffer, to which the first process writes and from which the second process reads. In order to avoid problems due to the concurrent usage of the same memory space, the buffer access must be regulated by locks: when a process wants to obtain the access to the buffer, firstly it must obtain the lock; if the lock is free, the operation can be executed, if the lock is engaged, the process must wait until it is released.

In this implementation, every thread that have to exchange images with the next thread in the pipeline shares an unbounded list with it. The list must be accessed by specific functions, using locks to guarantee the mutual access to the resource. Every thread, apart from the input thread, firstly executes a check on its input list; whenever the preceding thread writes an image in it, the list is mutually accessed, and the input is elaborated. Finally, the result is written to the output list shared with the next thread in the pipeline, apart from the output thread.

Chapter 6

The hardware setup

6.1 Hardware description

The thesis application has been deployed and tested on a development kit called iW-RainboW-G27M. This kit is used as demonstration for applications to be implemented on the NXP's iMX8 system on chip. In fact, the final application of the GOLD-based lane detector should run on a custom board based on such architecture.

The system on chip used for this thesis, in particular, is the iMX8 QuadMax version. It is composed by eight processors, namely:

- Two ARM Cortex A72 cores, which are processors based on a 64-bit architecture, equipped with the specific hardware components needed in order to run a Linux distribution, such as a memory management unit (MMU) to handle memory paging. The nominal working frequency is 1.6 GHz, and they are equipped with a L1 48 KB instruction cache and a 32 KB data cache. The two processors are part of the same CPU platform, sharing a 1 MB L2 cache.

- Four ARM Cortex A53 cores, which are 64-bit architecture-based processors too. As the A72 cores, they are equipped with the necessary hardware components to run a Linux-based operating system. The nominal working frequency is 1.2 GHz and the L1 cache is subdivided into a 32 KB instruction cache and a 32 KB data cache. As before, the four processors are part of the same CPU platform, sharing a 1 MB L2 cache.

- Two ARM Cortex M4F cores, which are 32-bit architecture-based processors, used for real-time control purposes. These processors may not run a Linux distribution, but can be interfaced with the previously presented CPU platforms and other components through the serial bus. The nominal working

frequency is 266 MHz and it is equipped with a 256 KB embedded tightly coupled memory (TCM).

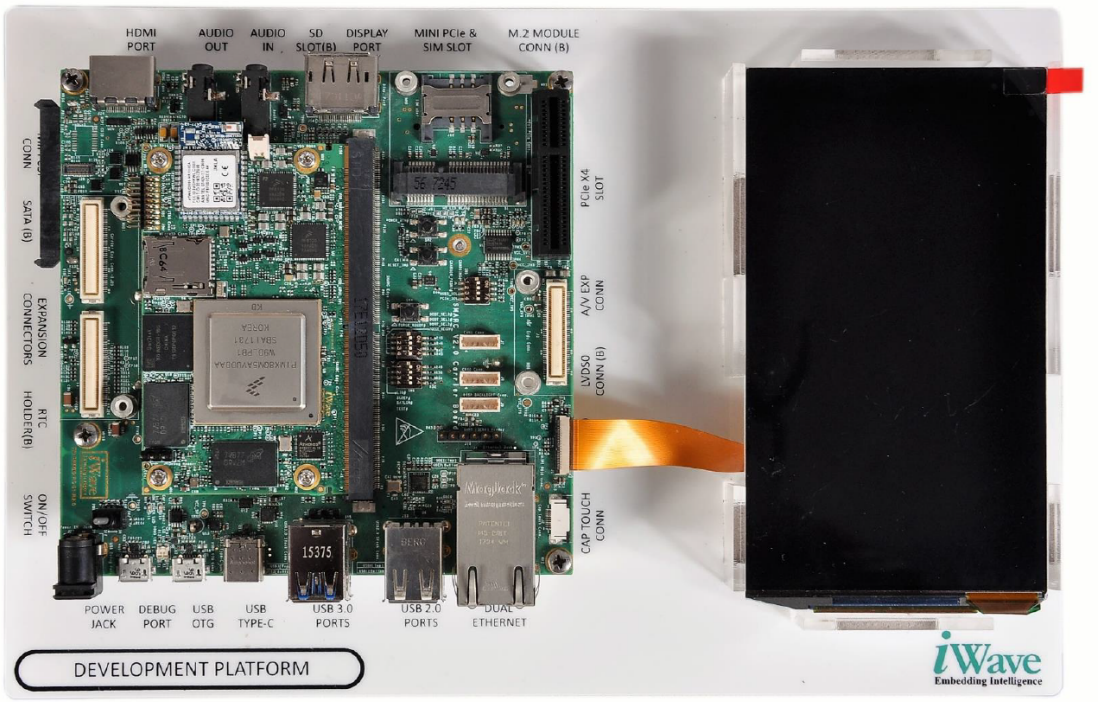

Figure 6.1. The iW-RainboW-G27M development board

The GOLD-based lane detector is executed on a Linux distribution running on the six 64-bit architecture-based processors. The two ARM Cortex M4F cores can be, instead, used to implement a control loop algorithm that exploits the geometrical information about the lane center computed by the lane detection algorithm.

The iMX8 QuadMax is also equipped with a hardware video processing unit (VPU). This component is able to accelerate the decoding and the encoding operations for many video formats, including the H264 format used in the presented application. The embedded RAM on the chip is a DDR4 memory of 4 GB, which was big enough to handle the frames lists shared among threads for intra-thread communication.

The iMX8 QuadMax system on chip is connected to the iW-RainboW-G27M module peripherals through a 314-pin edge connector. In particular, in this thesis three main peripherals were used:

- The Debug UART port, that was exploited during the programming phase and testing phase.

- The 1 Gb Ethernet ports, that were used to remotely connect to the system and test the application.
- The HDMI 2.0 out port, from which the graphical output of the application is provided externally.

In a real-case implementation of the GOLD-based lane detector, a USB port should also be used to obtain the image stream as input.

Both the ARM Cortex A72 cores and the A53 cores are equipped with a hardware component called NEON, which is a 128-bit extension for single instruction multiple data (SIMD) operations. This module is composed by 16 registers of 128 bits, that can be treated as 32 registers 64 bits-long also and allows to perform the same operation on all the registers at the same time. For example, two vectors made by 16 unsigned integer of 8 bits can be summed at the same time using only two NEON registers for the two vectors and one register to store the results.

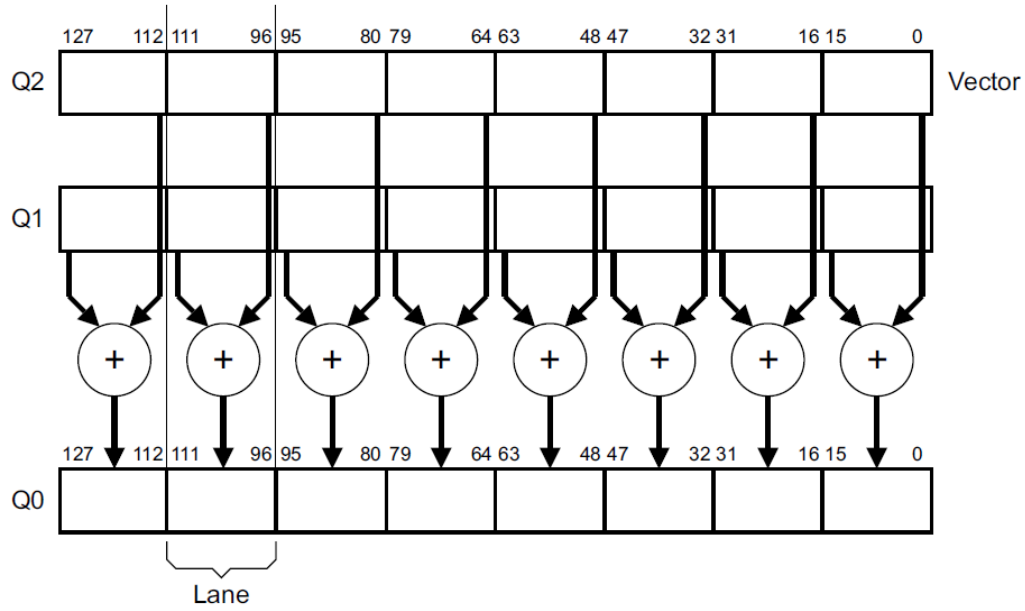

Figure 6.2. Example of a sum of two vectors of 16-bit elements using NEON registers

This architecture is extremely useful in a matrix-based application like the one presented in this thesis and may lead to an important performances increasing. The C/C++ code was compiled in a compatible way by enabling the "-O3" optimization option provided by the g++ compiler for 64-bit ARM architectures.

In order to obtain test video to introduce into the system to verify the output results, a GoPro Hero 7 Black camera was used. The videos was recorded at a resolution of 1080p (1920x1080 pixels) with a framerate equal to 30 frames per second. The camera has a sensor resolution of 12 MP and a shooting angle of 120 degrees. A useful feature is the HyperSmooth stabilization system: the camera is able to process each frame before recording it into the video to reduce disturbances due to sudden camera movements. In this way, for example, the recorded videos do not suddenly change the pitch angle with respect to the road when the vehicle encounters a pothole, reducing the possibility of a wrong inverse perspective mapping.

6.2 Simulation and final setups

The real-case implementation of the final system should be structured into the following components:

- The two 64-bit architecture-based CPU platforms of the iMX8 QuadMax system on chip, where the Linux distribution runs, and the GOLD-based lane detector is executed.
- The 32-bit architecture-based CPU platform composed by the two Cortex M4F cores, where a control loop algorithm using the output from the lane detector should be implemented.
- A camera device, connected through USB to the system, providing real-time streams of images to give as input to the lane detector.
- A display device, connected through HDMI to the system, visualizing the graphical output of the lane detector.
- A communication protocol interface, to communicate externally the actuation commands computed by the control algorithm.

As the thesis is focused on the development and on the testing phase of the lane detector module only, the final setup used to simulate and to test the system is composed by:

- The two 64-bit architecture-based CPU platforms of the iMX8 QuadMax system on chip, where the Linux distribution runs, and the GOLD-based lane detector is executed.
- MP4 file videos from file system, to be given as system input and simulating the camera device. The filesystem is stored into an external SD memory card.
- A display device, connected through HDMI to the system, visualizing the graphical output of the lane detector.

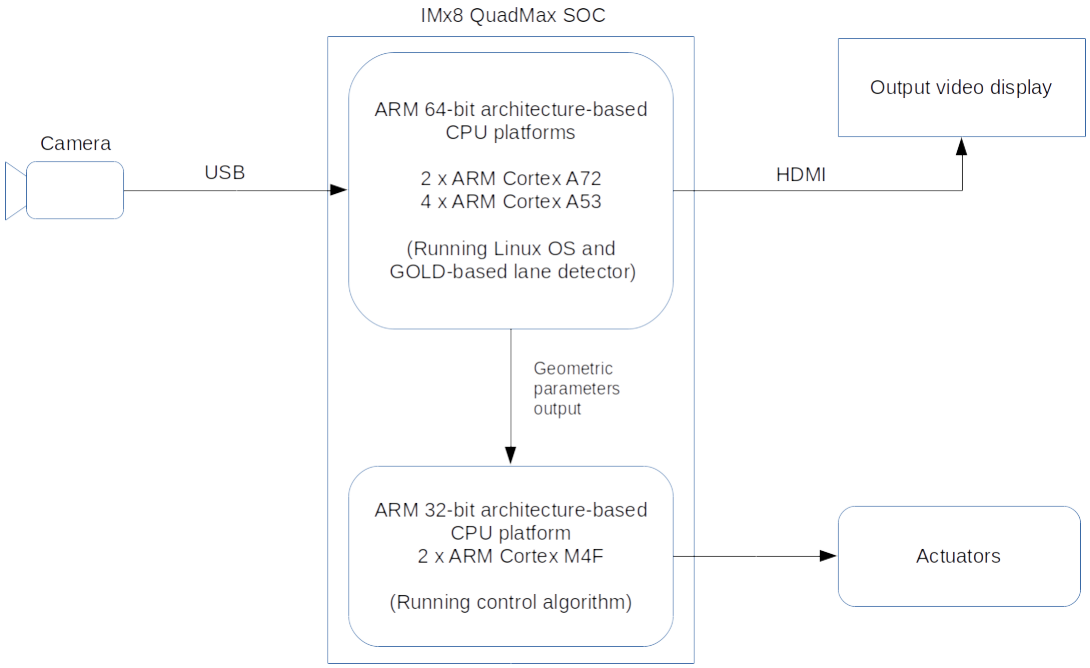

Figure 6.3. Graphical scheme of a real-case system setup

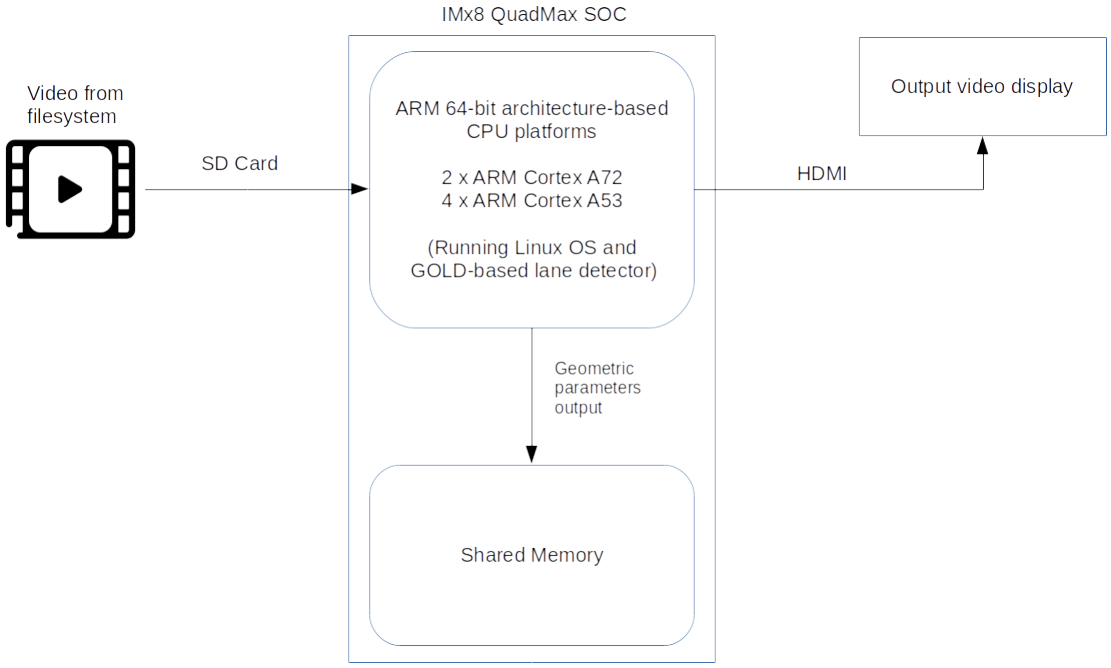

Figure 6.4. Graphical scheme of the simulation system setup

Part III

Test results and conclusions

Chapter 7

Results

The testing phase was performed using the simulation hardware configuration explained in the previous chapter. As said, a GoPro Hero 7 Black camera was used. It has been placed inside a car on top of the windshield, at a height of 1.220 meters above the ground and pointing ahead with a pitch angle equal to 2.5 degrees.

Fifteen videos have been captured in several different scenarios, such as urban and semi-urban roads and extra-urban highways, mainly in the Turin ring road, in north Turin urban roads and in semi-urban roads near the city (in the Canavese area). Moreover, the videos were taken at different hours, with daylight and at night, and with different environmental scenarios, e. g. with the sun in front of the camera or in a rainy condition.

As the testing phase was based on previously recorded videos given as file inputs to the system, a problem regarding the real-time execution condition occurred: in fact, the time needed by the OpenCV API to grab a BGR FullHD frame (1920x1080 pixels for every of the three channels) from the file system and decode it, passing from the H264 format used inside the MP4 file to a matrix of pixels, took around 50 milliseconds. Being the nominal working rates of the system either 30 frames per seconds (one frame computed every 33.33 milliseconds) or 24 frames per seconds (one frame every 41.66 milliseconds), a simulation with a video file as input cannot reach the output real-time goal, which can be guaranteed by a video stream from a USB camera.

To avoid this problem and to perform a relevant simulation, an intermediate solutions has been taken: the application has been modified for the testing purpose in order to firstly read and store all the frames of a video and then giving as input to the system every frame with a delay which guarantees the correct input rate. Obviously, this approach can be exploited for short video only, being that every frame must be load in a list stored in memory and it is likely that a memory overflow

occurs if the number of frames is high.

Exploiting this method, the testing phase have been performed simulating both a real-time video stream at 30 frames per second and 24 frames per second. For each frame rate, all the fifteen video samples have been given as inputs, repeating the simulation three times.

All the elaborations have been performed with the following set of elaboration parameters:

- $m = 6$
- $th = 20$
- $N = 3$, the kernel size of the median filter
- $th_{min} = 20$
- $m_{filter} = 20$
- $post\text{-}processing\ scaling\ factor = 6$
- $polynomial\ degree = 1$

7.1 Timing results

The first metric used to evaluate the system results is the capability to reach the output real-time goal, namely the capability of provide the output frames at the same rate in which they entered into the system as inputs.

Being the application divided in several threads, for each of them the execution time for every frame has been measured, without considering the time when the thread was polling the input list, waiting for the next frame. At the end of the stream elaboration, every thread computed the mean time spent for a single frame and the percentage of elaborations which did not exceed the time limit (33.33 milliseconds when the rate is set to 30 frames per second, 41.66 milliseconds when the rate is 24 frames per second).

Table 7.1. System performance at 30 FPS

Target FPS	Input FPS	Output FPS
30	29.97	29.895

Moreover, the overall mean execution time for each thread has been computed by measuring the total amount of time between the moment when the first input entered the system until the moment when each thread completed its operations. This amount of time has been divided by the number of elaborated frames in every stream, obtaining the actual output rate, which also considers the time spent by threads waiting the new inputs from the input shared list, the time spent waiting for the lock to access shared objects and the delay between the moment when the first frame was given as input and the moment when its elaboration was concluded.

Table 7.2. Threads performance at 30 FPS - 1

Pre-processing mean time (s)	Pre-processing percentage under time limit	Processing mean time (s)	Processing percentage under time limit
0.0299	99.595	0.01406	99.991

Table 7.3. Threads performance at 30 FPS - 2

Post-processing mean time (s)	Post-processing percentage under time limit	Output mean time (s)	Output percentage under time limit
0.01575	99.991	0.01554	99.91

Giving frames as input with a rate equal to 30 frames per second, the gathered numeric results showed that:

- The pre-processing thread, responsible of performing the inverse-perspective mapping, had a mean elaboration time per frame equal to 29.90 milliseconds, exceeding the time limit of 33.33 milliseconds only with the 0.405% of the overall number of elaborated frames.
- The processing thread, responsible of the low-level feature extraction and model fitting, had a mean elaboration time per frame equal to 14.06 milliseconds, exceeding the time limit with only the 0.009% of the overall elaborated frames.
- The post-processing thread, responsible of the reverse transformation to return in the original image reference space, had a mean elaboration time equal to 15.75 milliseconds, exceeding the time limit with only the 0.009% of the overall elaborated frames.

- The output thread, responsible of the output frame composition and of its sending to the graphical user interface, had a mean elaboration time per frame equal to 15.54 milliseconds, exceeding the time limit with only the 0.09% of the overall elaborated frames.

In an ideal condition, with the input stream coming from a real-time source, the input thread mean time per frame should directly depend on the rate of the incoming inputs. In this simulation, the stream was coming from a video in the file system, therefore the input thread was forced to send a frame almost every 33,33 milliseconds. The actual input rate was 33.37 milliseconds, with an actual frame rate equal to 29.97 frames per second.

The overall output frame rate computed was 29.895 frames per second, with a mean overall throughput of one frame every 33.45 milliseconds. The delay with respect to the mean input frame rate, equal to 0.075 milliseconds, is in minimal part due to the inter-thread communication mechanism and partially due to the initial delay of the first frame, which is provided as output with a mean overall elaboration time of 71.05 milliseconds. More the elaborated frames of the input stream are numerous, more this initial delay can be approximated to 0, ensuring the reach of the real-time elaboration goal.

Table 7.4. System performance at 24 FPS

Target FPS	Input FPS	Output FPS
24	23.981	23.935

Table 7.5. Threads performance at 24 FPS - 1

Pre-processing mean time (s)	Pre-processing percentage under time limit	Processing mean time (s)	Processing percentage under time limit
0.02797	99.925	0.01305	100

The same simulation has been repeated adapting the sample videos, filmed at a rate of 30 frames per second, to a 24 frames per second stream, exploiting the GStreamer tool. The results showed that:

- The pre-processing thread had a mean elaboration time per frame equal to 27.97 milliseconds, exceeding the time limit of 41,66 milliseconds only with 0.075% of the overall number of elaborated frames.

Table 7.6. Threads performance at 24 FPS - 2

Post-processing mean time (s)	Post-processing percentage under time limit	Output mean time (s)	Output percentage under time limit
0.01501	99.988	0.01345	99.996

- The processing thread had a mean elaboration time per frame equal to 13.05 milliseconds, never exceeding the time limit.
- The post-processing thread had a mean elaboration time equal to 15.01 milliseconds, exceeding the time limit with only the 0.012% of the overall elaborated frames.
- The output thread had a mean elaboration time per frame equal to 13.45 milliseconds, exceeding the time limit with only the 0.004% of the overall elaborated frames.

The actual input frame rate was equal to 23.981 frames per second; therefore, a frame was given as input once every 41.70 milliseconds. The output stream reaches the throughput value of one frame every 41.78 milliseconds, with an output frame rate equal to 23.935 frames per second. The results are in line with the elaboration with the frame rate set to 30 frames per second. In this case also, the real-time elaboration goal has been reached considering the offset between the input and the output frame rates approximated to 0 with larger number of frames.

It is important to underline that these results have been obtained without saving the output stream to a file. In fact, with this feature activated, the output thread mean elaboration time for one frame becomes equal to 62.03 milliseconds due to the time needed to interface with the file system, exceeding the time limits for both the 30 and 24 frames per second rates. Therefore, the output saving should be considered a diagnostic and testing feature only, without using it during a real-time session.

However, it must be highlighted that the geometrical parameters describing the center of the lane are computed inside the processing thread. Therefore, the mean total amount of accumulated delay from the time when the frame is given as input to the moment when the output parameters are written into the shared memory is the sum of the pre-processing and the processing threads mean time only, namely about 43.96 milliseconds at 30 frames per second and 41.02 milliseconds at 24 frames per second.

7.2 Elaboration results

As explained, the tests have been performed starting from fifteen videos registered in various scenarios and with different environmental conditions, including weather conditions, different illumination and hour of the day. In the presented output images, the lane markers are highlighted in green. These colored pixels are the result of the processing phase of the algorithm. Moreover, in order to verify the correctness of the geometric parametrization of the center of the lane, the real-world points described by the computed linear equation are translated back to the image coordinates, generating a visual representation in green as well.

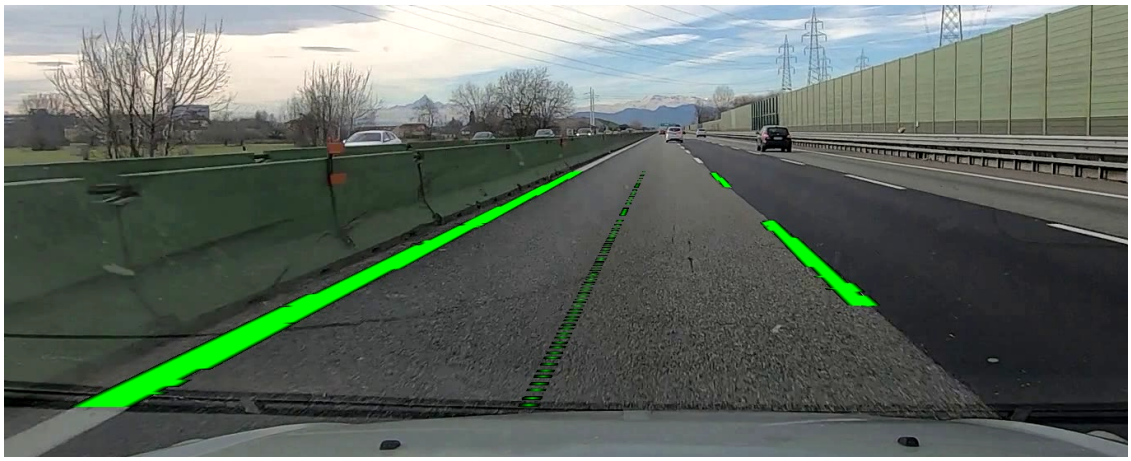

Figure 7.1. Good result obtained in a highway scenario, Turin ring road

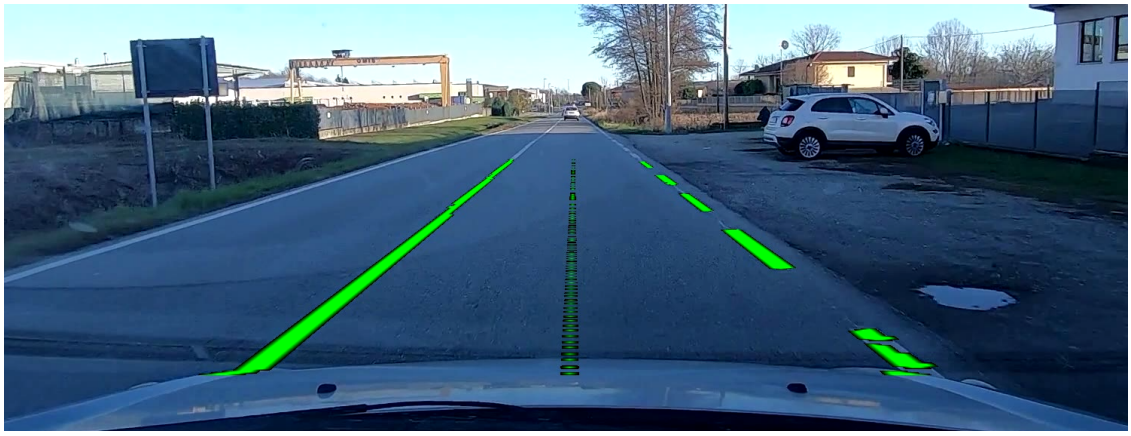

Figure 7.2. Good result obtained in a semi-urban scenario, Rivara (TO)

Good results have been achieved in the main scenario where lane detection algorithms are currently used and where they perform better, which is the highway scenario. In fact, the lane markers are often clearly visible and brighter than the asphalt, although some exceptions may occur, as road works. Moreover, the road is well maintained due to the high speed of cars traveling on it and the lanes can be successfully approximated as straight.

Figure 7.3. Good result obtained in a highway scenario at night, Turin ring road

Figure 7.1 describes the result obtained in the Turin ring road, at daylight and without traffic. In Figure 7.2, it is possible to notice that, wherever the road conditions are comparable with the highway scenario, the results are similar.

Moreover, if a good illumination is guaranteed by car headlights or by infrastructure lamps, good results are obtained also at night. Figure 7.3 describes the result obtained in the Turin ring road at night, without traffic.

The presented performances are maintained in traffic condition also, if the road is good condition and the illumination is good as well. An example is presented in Figure 7.4, where the Turin ring road presents traffic conditions.

As said, the results obtained are valid as long some conditions are encountered, and the good state of the road is one of these. In fact, if the road has potholes or lane markers cannot be easily distinguished by the system, the results can be extremely far from an acceptable level. This situation can be partially avoided through a well-designed filtering phase, which can be able to remove wrong markers detection if the road is not too much ruined. Figure 7.5 is an example of a good performance even in presence of a road in bad condition.

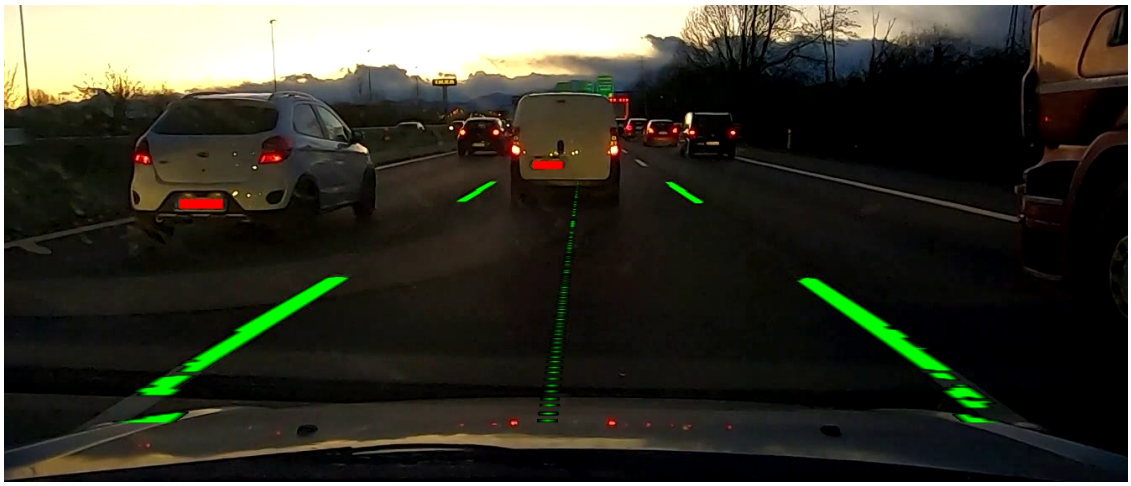

Figure 7.4. Good result obtained in a highway scenario with traffic, Turin ring road

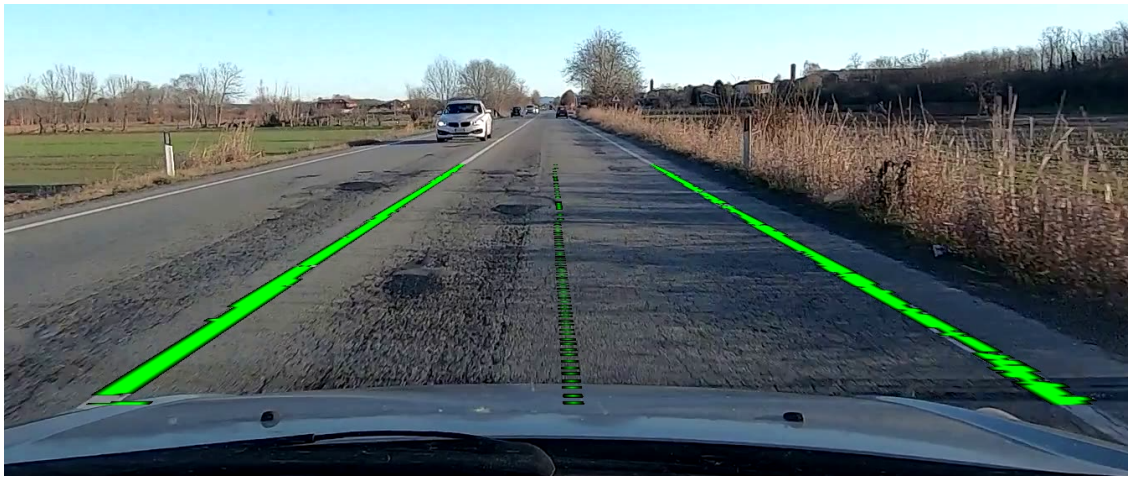

Figure 7.5. Good result obtained in a road in bad conditions, Salassa (TO)

Another possible factor able to reduce the system performances is the illumination, in particular the sun illumination that importantly varies during the day.

In Figure 7.6, it is possible to notice a loss of detection performance due to the sun light illumination on the right part of the road, reducing the brightness difference between the asphalt and the lane marker.

In Figure 7.7 and Figure 7.8 is presented a comparison between two images taken

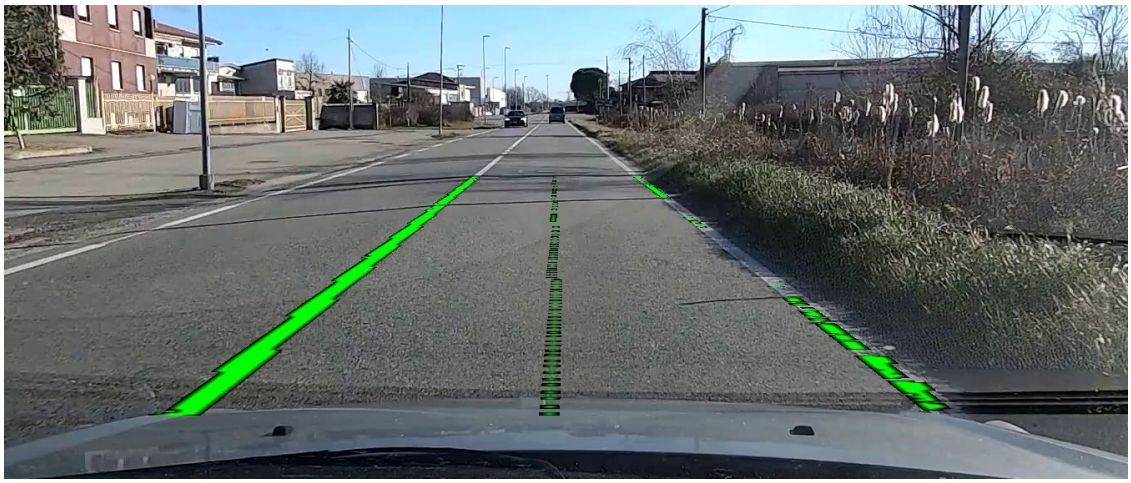

Figure 7.6. Example of problems caused by sun light - 1, Rivara (TO)

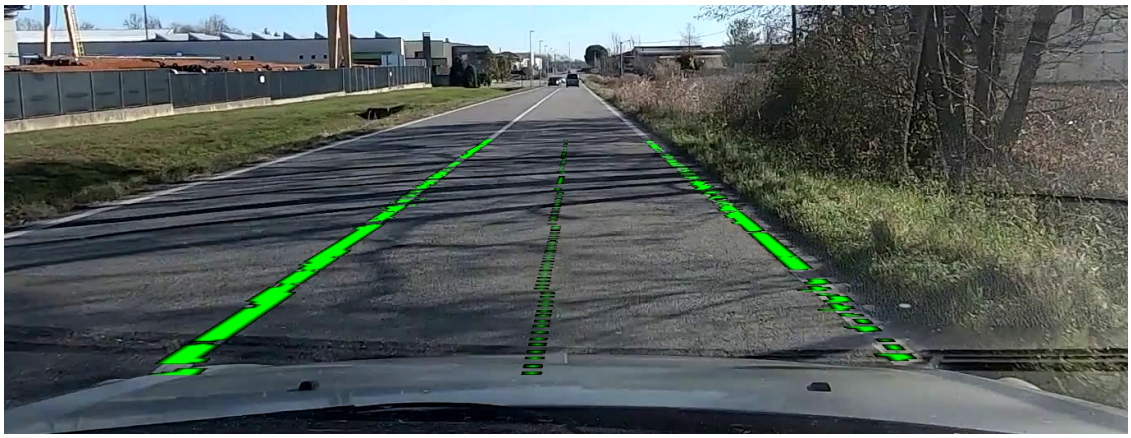

Figure 7.7. Example of problems caused by sun light - 2, Rivara (TO)

in different hours of the same road segment. In the first case, the sun position projected the shadows of the trees on the lane, although the filtering phase correctly removed possible outliers. In the second, the particular illumination caused the system fail to correctly detect all the right lane marker.

Another issue related to the sun light is the presence of the sun directly in front of the camera. In this case, the captured frame could present brightness noise which can reduce the system performance. Some examples of this effect can be found in Figure 7.9 and Figure 7.10, captured in a highway scenario with good road conditions and in a semi-urban scenario with bad road conditions. In both cases, the lane in direction of the sun light present a loss of performances due to the minor

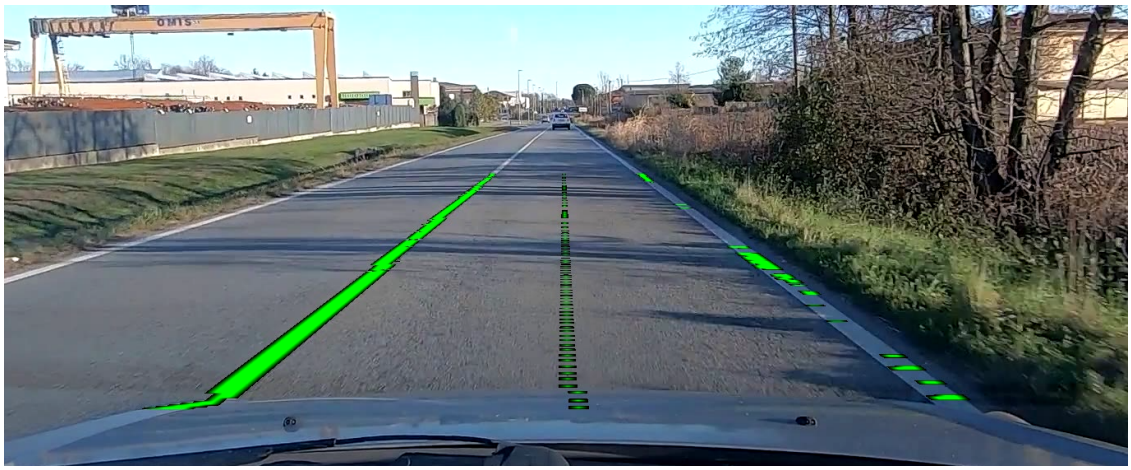

Figure 7.8. Example of problems caused by sun light - 3, Rivara (TO)

Figure 7.9. Example of problems caused by sun light in front of the camera - 1, Turin ring road

brightness difference between the asphalt and the white lane marker. This problem can be partially solved adequately adjusting the camera parameters to reduce the light noise.

Moreover, paintings and road works can affect the final system results. In Figure 7.11, a painted arrow in a highway scenario caused a bad lane detection. In this cases, more advanced filtering techniques are needed to proper recognize the arrow as not part of a lane marker and discard the related pixels.

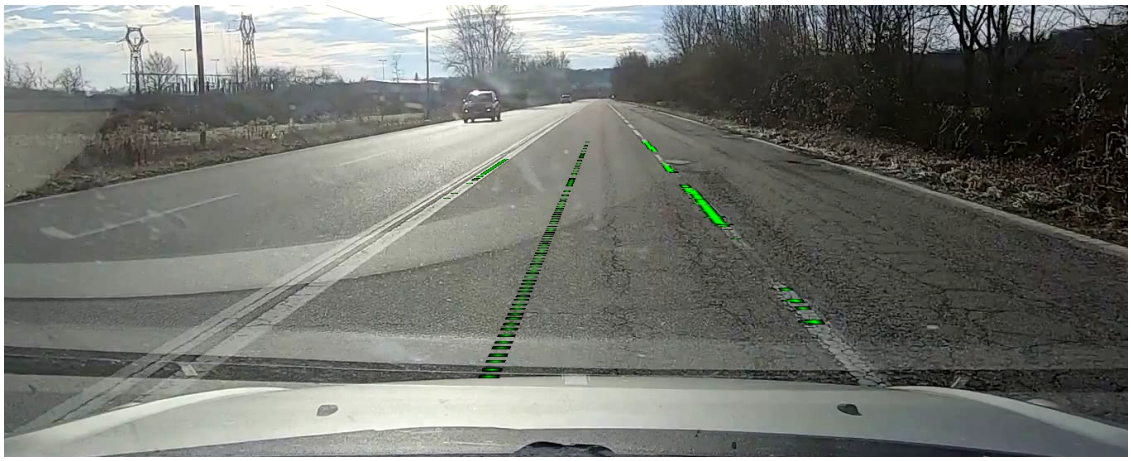

Figure 7.10. Example of problems caused by sun light in front of the camera - 2, Busano (TO)

Figure 7.11. Example of problems caused by road paintings, Turin ring road

In Figure 7.12 and Figure 7.13, some road works were present. In this case the system performed well, thanks to the histogram-based filter proposed, avoiding bad detections.

Another possible issue is the weather. In Figure 7.14 and Figure 7.15 the problems caused by the reflective effect of the water on the ground are clearly shown. The cars' headlights produce an important noise onto the frame.

However, the system performed well even in case of rain in the Turin ring road, as described by Figure 7.16, thanks to the better asphalt quality and to the absence

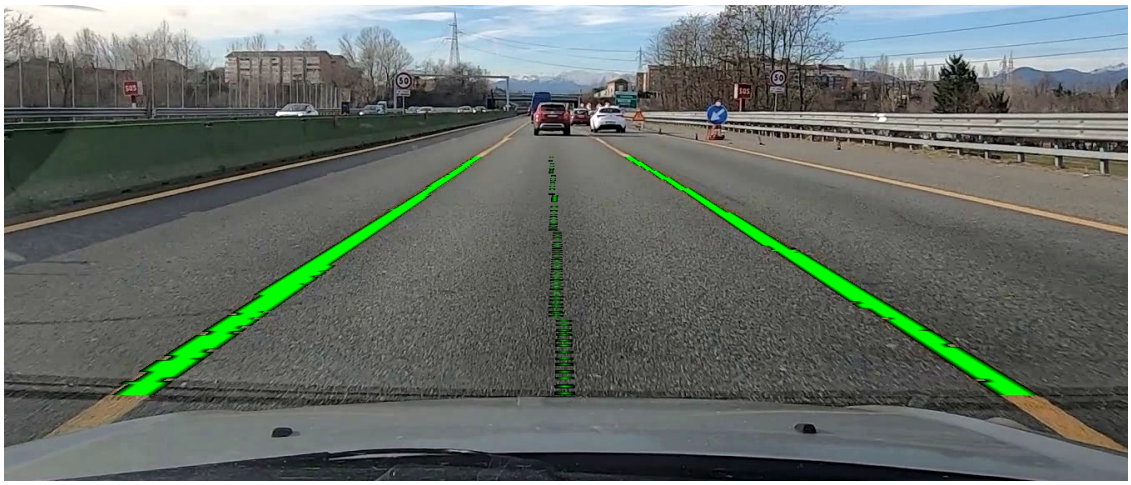

Figure 7.12. Example of detections with road works - 1, Turin ring road

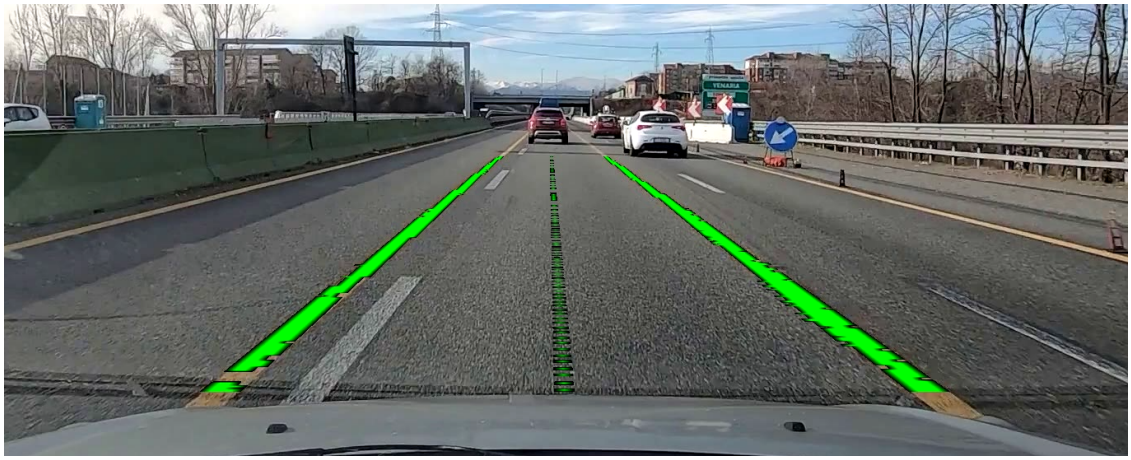

Figure 7.13. Example of detections with road works - 2, Turin ring road

of cars incoming from the front able to produce light reflection on the road.

The last scenario in which tests have been taken is the urban one. As it is possible to see in Figures 7.17, 7.18, 7.19 and 7.20, traditional lane detection system have difficulty to properly work in such scenario: in fact, even if in some cases the specific conditions met in highways are present, as the approximation to a straight road and the clear visibility of the markers in any situation, the high variance of the environment scene, the presence of road paintings and road crossings without markers do not allow an acceptable system reliability.

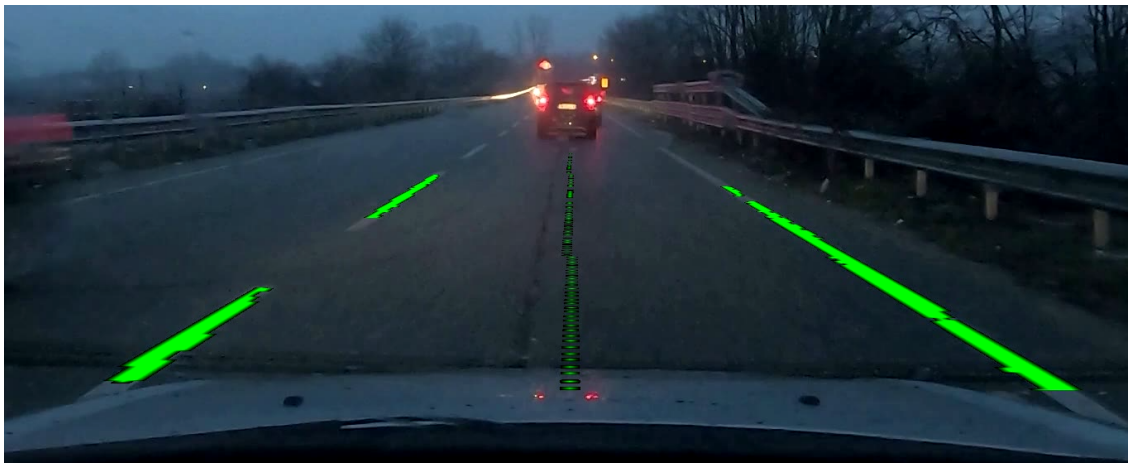

Figure 7.14. Example of problems caused by rain - good detection in absence of incoming cars, Orbassano (TO)

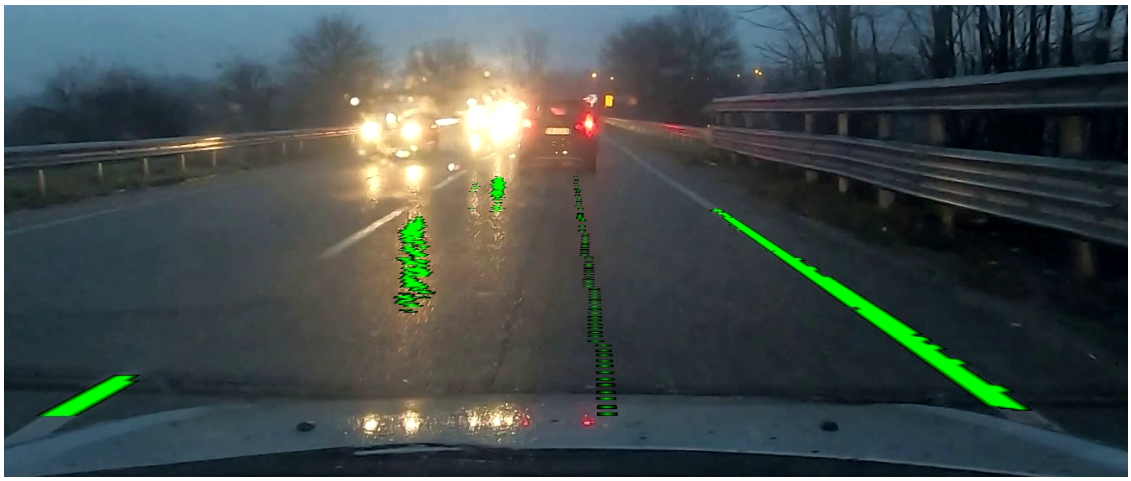

Figure 7.15. Example of problems caused by rain - bad detection due to light reflection on the road, Orbassano (TO)

Finally, due to the simple structure of the model fitting module implemented in the presented system, the result are not acceptable in presence of a road which cannot be approximated as straight. An example can be found in Figure 7.21.

However, being that the markers are correctly detected, the problem can be avoided by implementing a semi-parametric model fitting module, based on Bezier curves or splines.

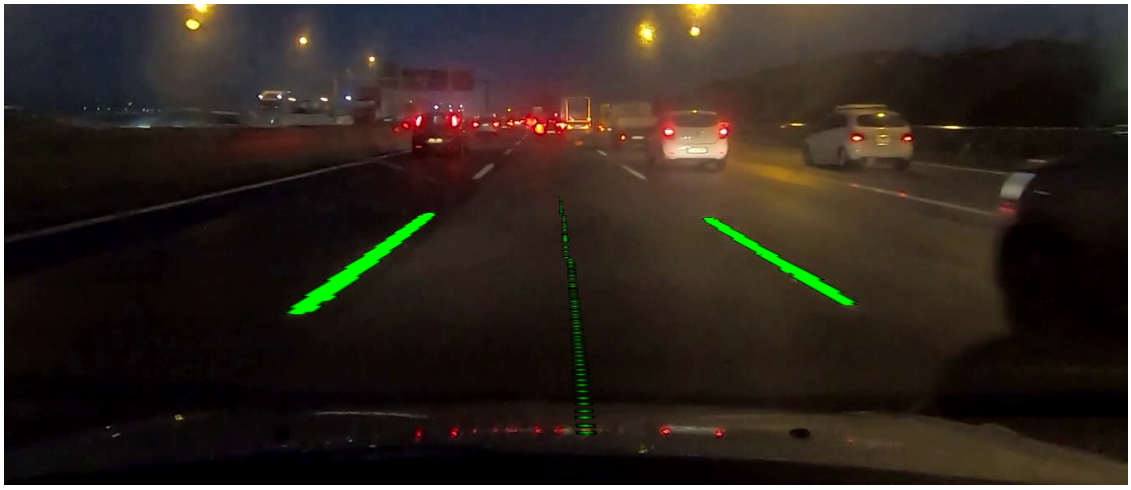

Figure 7.16. Good result obtained in a highway scenario in presence of rain, Turin ring road

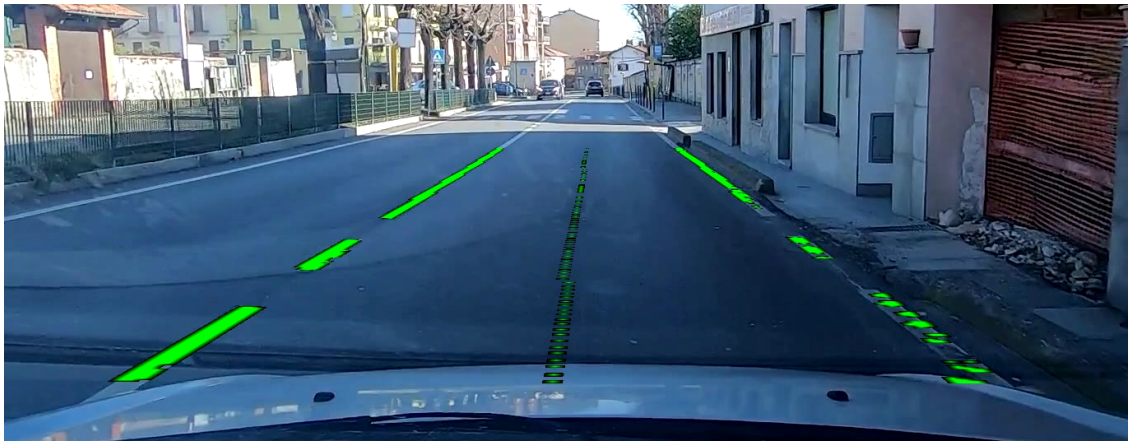

Figure 7.17. Example of elaboration in an urban environment - 1, Rivara (TO)

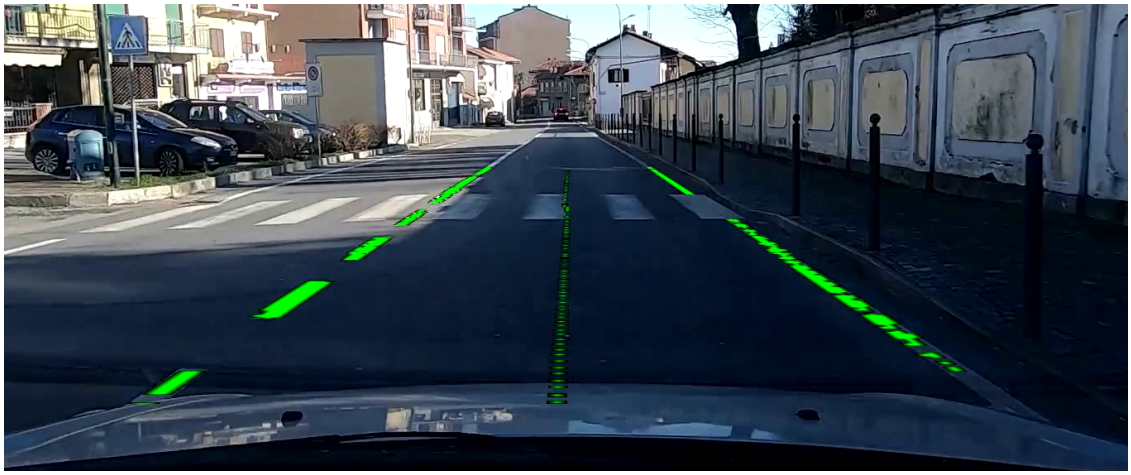

Figure 7.18. Example of elaboration in an urban environment - 2, Rivara (TO)

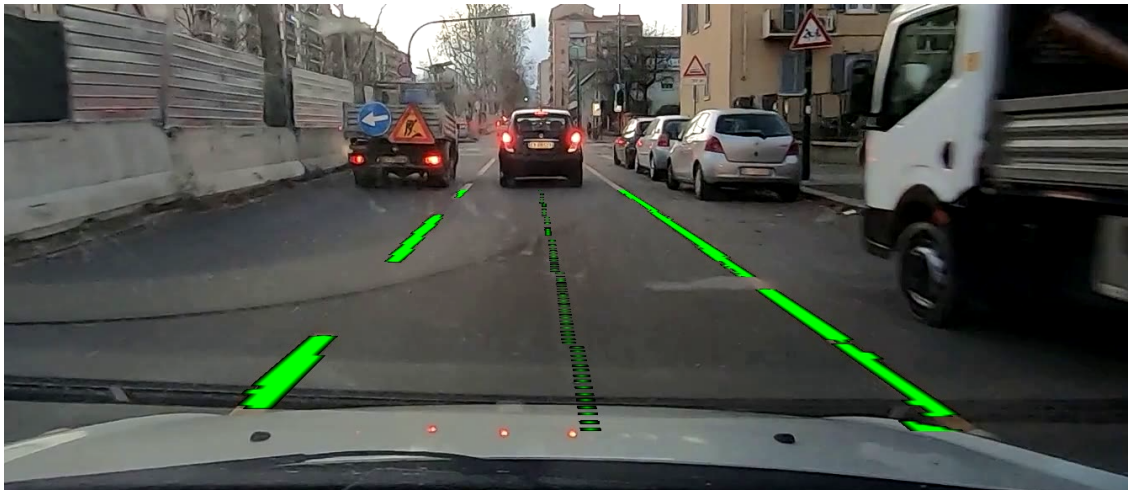

Figure 7.19. Example of elaboration in an urban environment - 3, Turin

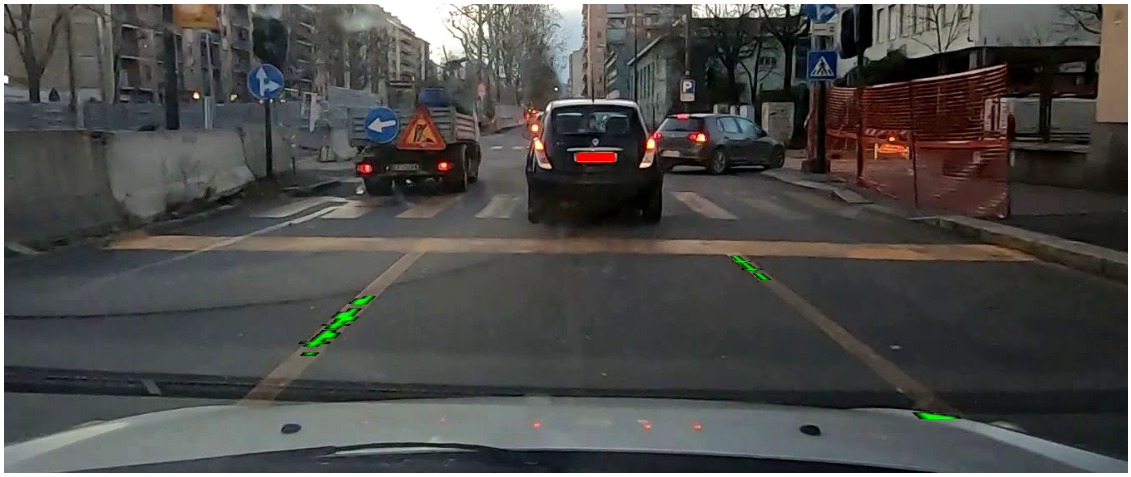

Figure 7.20. Example of elaboration in an urban environment - 4, Turin

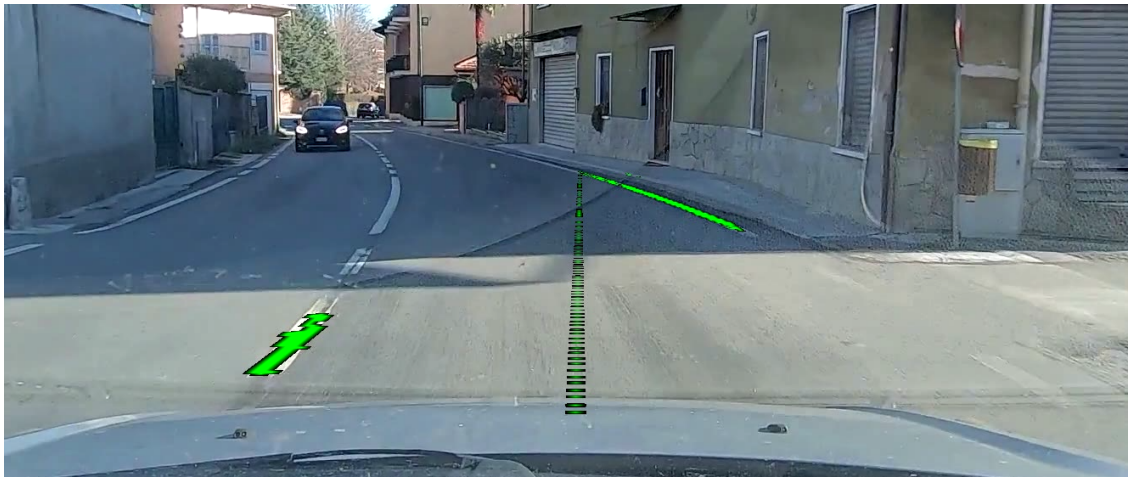

Figure 7.21. Example of erroneous road model fitting, Rivara (TO)

Chapter 8

Conclusions

In this thesis, an implementation of lane detection algorithm based on the GOLD system has been presented. The goals of this work were mainly related to two important aspects: the system should be able to perform a real-time elaboration of an incoming frame stream; it should be also able to correctly recognize the current lane markers in different environmental situations at least in one restricted scenario.

For what concern the first goal, the timing results showed that the real-time elaboration has been reached for incoming frame streams having rates equal to 30 frames per second and 24 frame per seconds, as long as a small delay has been registered due to threads communication mechanism, to the initial delay, intrinsic of the pipeline, and to eventual drops of performances, caused by the operating system scheduling policy.

In fact, the operating system and the scheduler architectures do not guarantee that the operations always take the same amount of time. The processors may not execute the application threads only, but it is possible that other system applications or kernel routines occupy a given core for a considerable amount of time, causing a delay on the thread operation on the current frame. Therefore, the application can be considered as real-time taking into account the mean elaboration times only.

However, the timing results have been evaluated simulating a real-time stream providing a new frame every 33.33 and every 41.66 milliseconds; this was implemented by previously reading all frames and then forcing to give them as system input, imposing the desired inter-frame time interval. This configuration was necessary due to the delay introduced by the frame reading operation directly from the file system, which introduced a delay of 50 milliseconds per frame.

In the final system, with the frame stream directly coming from a USB camera, the encountered delay due to the video file reading and decoding should not be present

and the input frames should be provided to the system with the correct frame rate.

The second goal to reach was the correct lane markers detection. The provided visual outputs showed the good results obtained by the algorithm when the road was not in bad conditions (with a good markers visibility), when the lane was approximately straight and the light illuminating the scene did not encountered obstacles, projecting shadows on the road.

In particular, the system performed well in the highway scenario, where the markers have a particularly good visibility against the asphalt. In this case, the algorithm correctly recognized lane markers also in presence of rainy conditions and at night. In other scenarios, although in some cases the results were valid, small environment changes caused performances drops.

In this implementation, a model fitting mechanism was introduced alongside the markers' recognition. This operation, however, is based on a quite simple linear model, leading to precise results only in specific conditions, i. e. when the road is approximately straight.

Given the reliability obtained by the markers detection in the highway scenario, the road model fitting operation could be improved in the future by implementing more accurate algorithms to obtain a curve road description, starting from the found lane markers pixels. Moreover, the markers detection itself can be further improved. In fact, in some situations outliers can be present, such as shadows on the road, light reflections due to the rain or disturbances due to the sun light in front of the camera, reducing the quality of the final results. To better handle this problems, more accurate filtering methods can be introduced and, to better integrate results in time, a Kalman filter can be implemented.

The final system is a real-time lane markers and lane center detector, able to work with input stream up to 30 frames per second and able to obtain reliable performances in restricted highway scenarios with good road conditions, visible bright markers and approximate straight shape. Other scenarios, as semi-urban and urban roads, cannot guarantee the minimal conditions in terms of environmental noise to permit a good system performance in term of detection results.

Finally, this kind of system could be introduced in vehicles belonging to level 1 or level 2 of the SAE classification only, due to the absence of a reliable automatic fallback procedure in case of wrong detections.

Part IV

Appendices and Bibliography

Appendix A

Embedded Linux and the YOCTO project

Nowadays, in the market of embedded system operating systems, it is possible to find two main system families:

- Real-time operating systems, like FreeRTOS or MicroC OS, mainly used in dedicated control systems.
- General purpose operating systems, like Android and the Embedded Linux distributions.

In the last years, technological improvements in the ambit of computing power and memory storage have allowed many embedded system architectures to run complex operating systems, not only able to access low-level interfaces, but also able to handle file systems, complex memory and scheduling management and user-level system interfaces. Many examples of applications using this kind of operating systems come from the automotive infotainment topic, complex internet of things (IoT) applications and complex industrial human-machine interfaces.

Due to their lightness, efficiency and robustness, embedded distributions of Linux have become more and more popular solutions. A Linux-based embedded system is composed by four main parts that allow it to work properly on different hardware architectures:

- The Bootloader, that is the basic software component, written for a particular hardware architecture, that is responsible to initialize every system peripheral and to start the Linux kernel process.
- The Device Tree, that is a data structure describing the list of the physical devices of the system and providing to the Linux kernel all the information

to properly activate and initialize the needed device drivers. In this way, the same Linux kernel, if the proper drivers are linked in it, can be able to run in different hardware architectures. Moreover, if a hardware change is needed in the system architecture, only this component must be modified, while the kernel should not be recompiled.

- The Linux kernel, that is the core component of the operating system. It provides the device drivers abstracting the hardware, the system call interface between the kernel itself and user applications and consists in four management modules: process management, memory management, network management and virtual file system management.

- The Root File System, that is the container of every user application and every system program. This is the last component initialized by the bootstrap process before the root user process starts to run.

Thanks to this architecture, in order to use a Linux distribution as the operating system of a proper hardware device, the only components that have to be changed are: the device drivers module of the kernel, by adding new drivers to interact with the new hardware; the device tree data structure, by adding the description of the new hardware devices to use; eventually, adapt the bootloader component to the new architecture. The other components of the kernel and the user applications inside the root file system may not be changed with respect to other Linux-based systems.

This approach makes possible to easily create custom Linux distributions, i. e. systems based on the Linux kernel designed to work only with specific hardware components and only with the user-space applications needed to work in a restricted application domain.

Another reason of the popularity of Embedded Linux as operating system is the possibility to load the device tree, the kernel and the root file system components from different memory sources. In fact, the bootloader can load those components from a flash memory embedded in the system on chip, from an external mass memory or even from a network-connected external device, making possible to design several system configurations depending on the system needs. The root file system component can be loaded on RAM, depending on the willingness to make its contents volatile or not: in fact, if changes happen while the file system is placed into the RAM, at power off they will be lost; if the root file system is loaded on a persistent memory, all the changes will be maintained.

In order to create an Embedded Linux custom distribution that fits the needs of a specific application scenario, the bootloader, the device tree, the kernel and

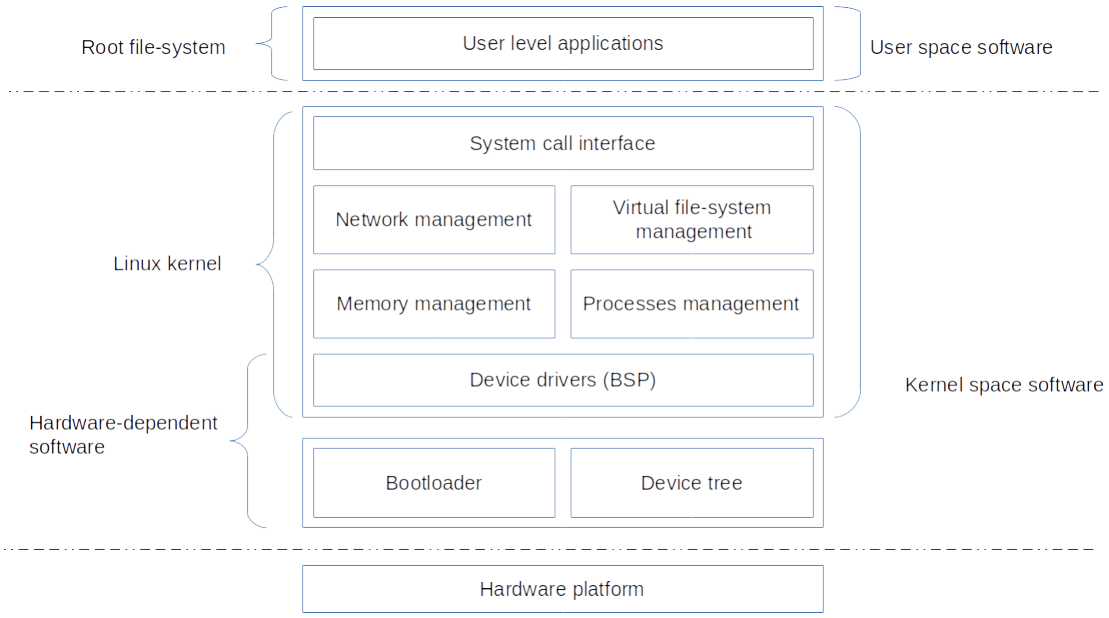

Figure A.1. The architecture of an Embedded Linux system

the root file system must be compiled starting from the existing source code and in a dependent way with respect to the hardware-specific configurations of the system where the operating system will be deployed to and with respect to the application-specific configurations, for what concerns the user level. This kind of operations is complex to manage and to execute manually, therefore an automated build system is fundamental to perform a custom distribution build.

One of the most used and standardized build systems is an open-source project called Yocto. Practically, a Yocto distribution is composed by collections of configuration files describing how every module of the operating system must be built and deployed and how the final system image must be composed.

The configuration files are usually in the form of "recipes", special files that define where to achieve the source code, how to build and where to deploy a given software component. Other configuration files can be related to user-specific configuration (for example, which libraries to include in the user level) or to hardware-specific configuration (for example, the list of the peripherals and their characteristics).

Several configuration files are collected into layers, that are containers which permit to customize set of software builds, to override determined compiling or deploying behaviors and to define for which architecture their software components may be

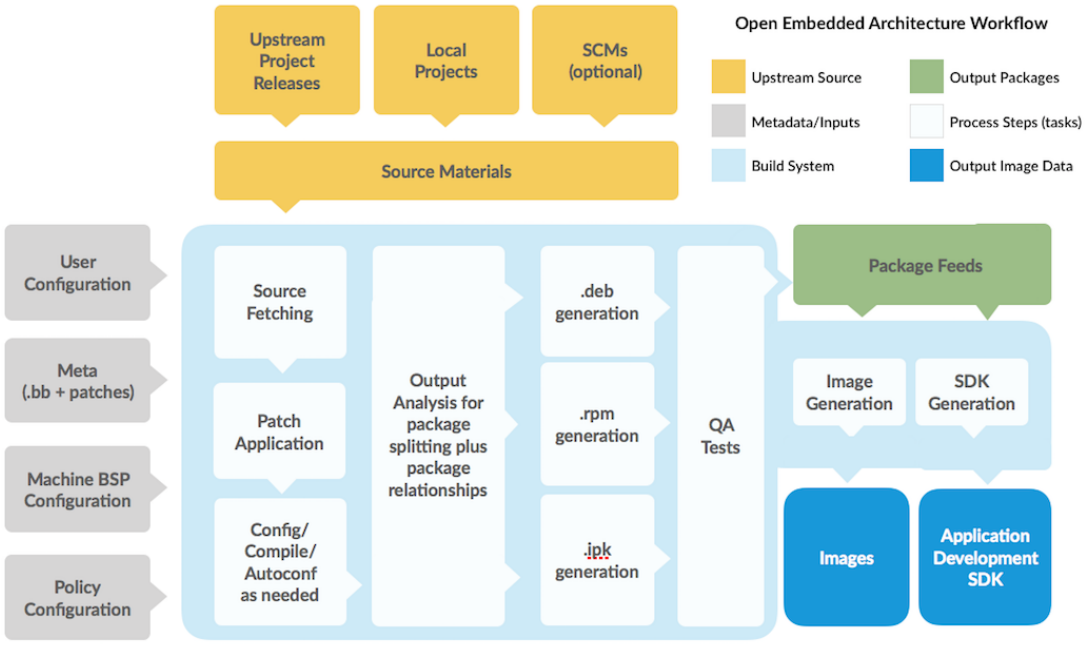

Figure A.2. The OpenEmbedded work flow, from https://www.yoctoproject.org/wp-content/uploads/2017/07/yp-how-it-works-new-diagram.png [Accessed 25 March, 2020]

added to.

The tool that permits to start from recipes and other configuration files and to finally obtain a built custom Linux distribution is called OpenEmbedded. This tool, through the usage of a task executor tool called Bitbake, is able to parse all the recipes linked to a given distribution and to execute the actions described, which normally are:

- The source code fetching, from a local or remote source.
- The patch application in case of patch updates.
- The compilation process.

The results of these operations are called packages, that are compiled software components of standard formats (for example, deb or rpm package formats). Once the different packages are generated, the build tool collects them together obtaining the system image, composed by the bootloader, the kernel image, the device tree and the root file system. Eventually, a software development kit can be generated, which is namely a set of tools and libraries to allow the developing of applications

based on the specific Linux distribution.

For example, in the work done in this thesis, the Embedded Linux distribution has been generated from an existing repository called "iwg27-release-bsp", developed by the board producer company starting from the default distribution template called "poky". This template is a base Embedded Linux distribution, maintained by the Yocto project foundation, that is possible to improve and modify, in order to obtain more specific or complex results. To the "poky" distribution, several new layers were added by the board producer, of which the most important is the Freescale BSP layer, containing the drivers and the hardware description of the board peripherals devices.

In order to adapt this base distribution to the needs of the thesis work, some configuration files have been modified. In particular, the file named "local.conf" in the folder `build_imx8qm/conf` of the distribution defines which software modules must be added to the already defined image configuration (see A.3). The modules added in this files are added for every configuration defined inside the distribution. In this thesis, there were added:

- The QT base library module, necessary to the application run and used in the Graphical User Interface (GUI).
- The OpenCV library, used in the low-level management of the images.
- The QT Creator debug server alongside with the OpenSSH server, necessary to remotely debug the overall application.
- The Eclipse debug server, used to debug the C++ application generated to test the low-level part of the application only.
- The OpenSSH SSH File Transfer Protocol (SFTP) server, necessary to upload the application from a remote host (namely, the development computer).

Another important file that needed to be modified to adapt the base distribution to the application requirements is the file named "bblayers.conf" in the folder `build_imx8qm/conf`. Inside this file the list of configuration layers added to the final distribution is defined. In order to make the QT5 library work on the generated Linux image, the entire "meta-qt5" layer has been downloaded, added to the distribution folder and defined to be used inside the "bblayers.conf" file (see A.4).

When the configuration files were properly set and the "meta-qt5" layer was added to the distribution, the Bitbake tool was used to compile the entire operating system image for the specified machine, namely "imx8qm_iwg27m". The result was composed by:

```
MACHINE ??= 'imx8qm_iwg27m'
DISTRO ?= 'fsl-imx-x11'
PACKAGE_CLASSES ?= "package_rpm"

PACKAGECONFIG_append_pn-qtbase = " fontconfig examples"

EXTRA_IMAGE_FEATURES ?= "debug-tweaks"

USER_CLASSES ?= "buildstats image-mklibs image-prelink"
PATCHRESOLVE = "noop"
BB_DISKMON_DIRS = "\
    STOPTASKS,${TMPDIR},1G,100K \
    STOPTASKS,${DL_DIR},1G,100K \
    STOPTASKS,${SSTATE_DIR},1G,100K \
    STOPTASKS,/tmp,100M,100K \
    ABORT,${TMPDIR},100M,1K \
    ABORT,${DL_DIR},100M,1K \
    ABORT,${SSTATE_DIR},100M,1K \
    ABORT,/tmp,10M,1K"
PACKAGECONFIG_append_pn-qemu-native = " sdl"
PACKAGECONFIG_append_pn-nativesdk-qemu = " sdl"
CONF_VERSION = "1"

DL_DIR ?= "${BSPDIR}/downloads/"
ACCEPT_FSL_EULA = "1"

CONNECTIVITY_CHECK_URIS = "https://www.google.com/"

IMAGE_INSTALL_append = " qtbase-examples qtdeclarative-plugins"
CORE_IMAGE_EXTRA_INSTALL += "opencv libopencv-core libopencv-imgproc"
EXTRA_IMAGE_FEATURES += "qtcreator-debug ssh-server-openssh "
DISTRO_FEATURES_remove = "wayland"
IMAGE_INSTALL_append = " openssh-sftp-server"
EXTRA_IMAGE_FEATURES += "eclipse-debug"

DEBUG_BUILD="0"
```

Figure A.3. The content of the "local.conf" file of the custom distribution

- A binary file renamed to flash.bin, containing the U-Boot bootloader code.
- A binary file renamed to Image, containing the Linux kernel image.
- The device tree file, containing the hardware description of the system.
- A tar formatted archive containing the root filesystem.

These files were deployed to the SD card from which the board boots. The SD card was previously dived in three partitions:

- A raw unformatted partition of 7 MB, containing the bootloader binaries and the boot environment variables space.
- A FAT16 formatted area of 504.7 MB, containing both the device tree file and the kernel image.

```
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BSPDIR := "${@os.path.abspath(os.path.dirname(d.getVar('FILE', True)) + '/../..')}"

BBFILES ?= ""
BBLAYERS = " \
  ${BSPDIR}/sources/poky/meta \
  ${BSPDIR}/sources/poky/meta-poky \
  \
  ${BSPDIR}/sources/meta-openembedded/meta-oe \
  ${BSPDIR}/sources/meta-openembedded/meta-multimedia \
  \
  ${BSPDIR}/sources/meta-freescale \
  ${BSPDIR}/sources/meta-freescale-3rdparty \
  ${BSPDIR}/sources/meta-freescale-distro \
"

# Freescale Yocto Project Release layers
BBLAYERS += " ${BSPDIR}/sources/meta-fsl-bsp-release/imx/meta-bsp "
BBLAYERS += " ${BSPDIR}/sources/meta-fsl-bsp-release/imx/meta-sdk "

BBLAYERS += " ${BSPDIR}/sources/meta-browser "
BBLAYERS += " ${BSPDIR}/sources/meta-openembedded/meta-gnome "
BBLAYERS += " ${BSPDIR}/sources/meta-openembedded/meta-networking "
BBLAYERS += " ${BSPDIR}/sources/meta-openembedded/meta-python "
BBLAYERS += " ${BSPDIR}/sources/meta-openembedded/meta-filesystems "
BBLAYERS += " ${BSPDIR}/sources/meta-qt5 "
```

Figure A.4. The content of the "layers.conf" file defining the layers included into the custom distribution

- An ext4 formatted area of 14 GB, containing the uncompressed root filesystem.

Finally, the boot settings arguments were defined in order to instruct the bootloader about the name and the location of the kernel image, the device tree and the root filesystem to load. Other environment settings may be defined, e. g. for the evaluation board, the primary display output device could be selected among the board screen and HDMI interface.

Appendix B

The pinhole camera model and 3D to 2D transformations

Every object can be described in the real world by assigning to every discrete point belonging to it three numerical values corresponding to the three coordinates in a 3D system. When taking a picture of that object, we are projecting the light produced by these points onto a film (in analog cameras) or onto a digital sensor.

The simplest model that can be used to describe the structure of a camera is the pinhole camera model. A pinhole camera is a system in which the camera aperture can be described as a single point, without using particular lenses to focus all the light coming from the environment in determined points of the sensor. This model is a simplified version of the structure of a camera, being that we can ignore the non-linear geometric distortion and the blurring effect introduced by the usage of lenses with finite sized apertures, obtaining a first order approximation of the overall model. These effects, however, should be subsequently handled in the computation of computer vision transformation, in order to obtain more precise results.

In this model, every light ray passes through the single-point camera aperture and reaches the sensor, posed at a distance f to the aperture point called focal length. The resulting image is rotated of 180 degrees.

Starting from this model, when can geometrically describe the obtained image also as a projection of the real-world object on a 2D plane, called image plane, posed to a distance f (equal to the focal length of the camera) from the aperture point. From now on, the aperture point will be described as the origin O and the axis perpendicular to the image plane passing by O will be named as optical axis.

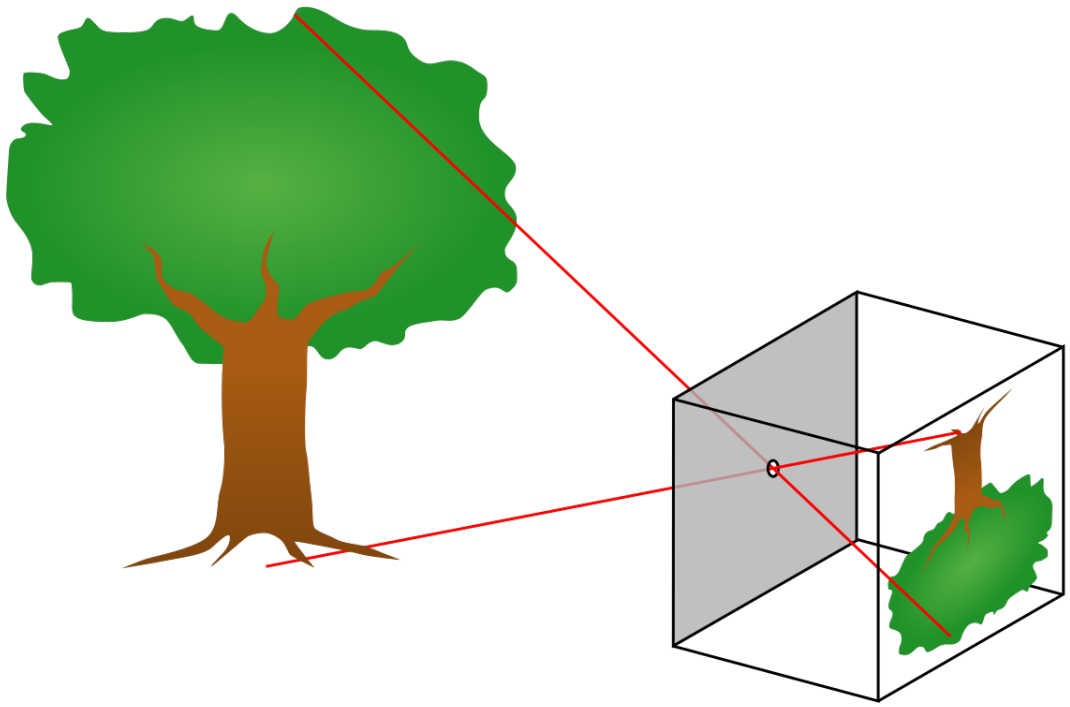

Figure B.1. A graphical representation of a pinhole camera

Now, we can define four coordinate systems:

- The 3D world coordinates system $World = \{(X, Y, Z)\}$, which is the description of the position of the real object in a predetermined 3D system.
- The 3D camera coordinates system $Camera = \{(X_c, Y_c, Z_c)\}$, which has the origin O in the camera aperture point and the Z_c axis corresponding to the optical axis.
- The 2D film coordinates system $Film = \{(x, y)\}$, which is the 2D projection of the real object points into the image plane, physically corresponding to the plane of the camera sensor in the pinhole model; the origin of this system is the center of the image plane.
- The 2D pixel coordinates system $Image = \{(u, v)\}$, which is the 2D affine transformation to pose the origin of the system to the first pixel of the pixel array, used to describe the image into a digital computation system.

Firstly, the points described in the world reference system must be transported into the 3D camera reference system. This can be done by applying rigid transformation

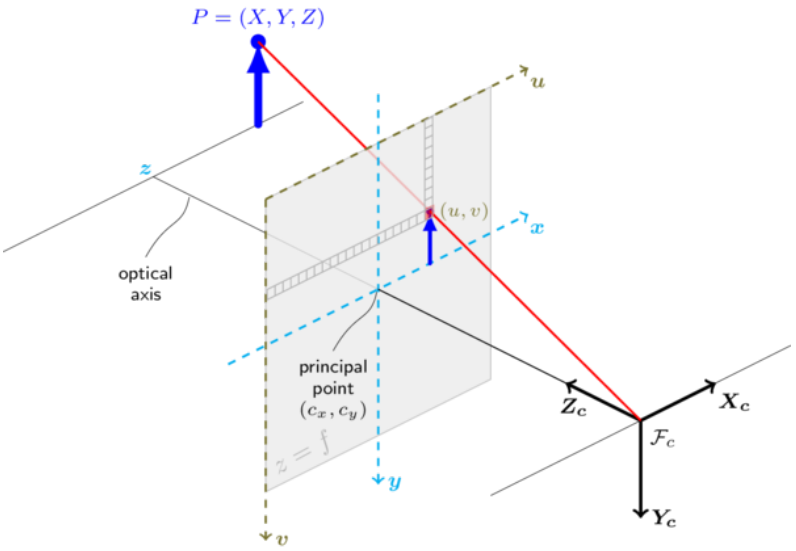

Figure B.2. The pinhole camera model, from https://docs.opencv.org/2.4/_images/pinhole_camera_model.png, [Accessed 27 March, 2020]

composed by a rotation and a translation. The rotation matrix R is derived by knowing the pitch (α), roll (β) and yaw (γ) angles between the world reference axis and the camera reference axis. The translation vector is derived by knowing the distances between the world reference origin and the camera reference origin O.

$$R = R_z(\alpha)R_y(\beta)R_x(\gamma) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{B.1}$$

$$T = \begin{bmatrix} -t_x \\ -t_y \\ -t_z \end{bmatrix} \tag{B.2}$$

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \begin{bmatrix} R \mid T \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{B.3}$$

When the rigid transformation has been applied, the 3D coordinates must be reported to the 2D image plane film coordinates. This operation can be done by

applying a projection onto the image plane. Geometrically, the perspective projection equation can be derived considering the triangles obtained between the optical axis and the X and Y axis. The obtained equations are:

- $x = f \cdot X_c/Z_c$
- $y = f \cdot Y_c/Z_c$

where f is the focal length (distance between the origin O and the image plane), Z_c is the optical axis distance of the given point in the camera reference coordinates and X_c and Y_c are the other two 3D coordinates in camera reference.

If the focal length of the x axis and the focal length of the y axis are not equal, we can rewrite the equations as:

- $x = f_x \cdot X_c/Z_c$
- $y = f_y \cdot Y_c/Z_c$

The obtained matrix representation of the projection is:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{B.4}$$

Finally, an affine transformation is needed to transform the 2D film coordinates into 2D pixel coordinates to represent the image in an array-based representation. The origin must be rigidly transpose to the top-left corner of the pixel image by a distance equal to the image height divided by two in the y axis and equal to the image width divided by two in the x axis. Moreover, the it is possible to introduce scaling factors to resize the pixel dimensions. The overall projection matrix becomes:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{f_x}{s_x} & 0 & o_x \\ 0 & \frac{f_y}{s_y} & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{B.5}$$

This matrix is called intrinsic matrix, because the transformation described only depends on internal camera parameters, such as the focal lengths, the image size and the pixel scaling factors. In the opposite way, the matrix describing the rigid transformation between the world and camera reference mapping is called extrinsic matrix, because its parameters only depends on external factors, i. e. the relative position between the camera and the world reference. The overall transformation

is thus defined as:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{f_x}{s_x} & 0 & o_x \\ 0 & \frac{f_y}{s_y} & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R \mid T \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \frac{f_x}{s_x} & 0 & o_x \\ 0 & \frac{f_y}{s_y} & o_y \\ 0 & 0 & 1 \end{bmatrix} \left(R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \right) \tag{B.6}$$

In a computer vision application, as the implementation of the algorithm described in this thesis, the inverse of the described 3D to 2D transformation may be extremely useful. In fact, knowing the characteristics of the camera (its intrinsic matrix) and the position in the space of the camera with respect to the world reference system and to the object (the extrinsic matrix) it is possible to map real-wolrd points to an image 2D plane.

In particular, given the pixel coordinate [u, v], if we pose $Z = 0$, it is possible the mapping of the points belonging to the plane $Z = 0$ in real world coordinates into a 2D image. Applying this concept to the lane detection scope, we can compute the inverse-perspective mapping of a captured road image, i. e. compute the mapping of the lane points in a space where the perspective effect is removed.

It is important to underline that this procedure to obtain the planar projection of the filmed scene into the $W = 0$ 3D plane is possible only when the input image is directed to a planar object. For example, in the case of this thesis the road must have a slope near to zero, or the transformation will not be effective. In case of a road slope, the extrinsic matrix parameters should be properly modified.

Bibliography

[1] S. Singh, “Critical reasons for crashes investigated in the national motor vehicle crash causation survey,” Tech. Rep., 2015.

[2] Istituto Nazionale di Statistica (ISTAT), “Incidenti stradali in Italia“, july 2019, (https://www.istat.it/it/files//2019/07/Incidenti_stradali_2018.pdf).

[3] P. Suresh, P. V. Manivannan, “Reduction of vehicular pollution through fuel economy improvement with the use of autonomous self-driving passenger cars“, *Journal of Environmental Research And Development*, Vol. 8 No. 3A, January-March 2014.

[4] M. Maurer, J. C. Gerdes, B. Lenz, H. Winner, "Autonomous Driving, Technical, Legal and Social Aspects", *Springer*, Chap. 16.

[5] T. Litman, “Autonomous Vehicle Implementation Predictions: Implications for Transport Planning“, Victoria Transport Policy Institute, January 2020.

[6] T. B. Lee. “Autopilot was active when a tesla crashed into a truck, killing driver“, (https://arstechnica.com/cars/2019/05/feds-autopilot-was-active-during-deadly-march-tesla-crash/).

[7] Wikipedia, https://en.wikipedia.org/wiki/Death_of_Elaine_Herzberg.

[8] SAE, “Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles,” SAE J3016, 2018, Tech. Rep.

[9] E. Yurtsever, J. Lambert, A. Carballo, K. Takeda, “A Survey of Autonomous Driving: Common Practices and Emerging Technologies“, *arXiv preprint arXiv*: 1906.05113v2, 2020.

[10] C. Bila, F. Sivrikaya, M. A. Khan, S. Albayrak, “Vehicles of the Future: A Survey of Research on Safety Issues“, *IEEE Trans. Intelligent Transportation Systems*, vol. 18, no. 5, pp. 1046-1065, may 2017.

[11] J. C. McCall, M. M. Trivedi, “Video-based lane estimation and tracking for driver assistance: survey, system, and evaluation.“, *IEEE Trans. Intelligent Transportation Systems*, vol. 7, no. 1, pp. 20-37, March 2006.

[12] A. Bar-Hillel, R. Lerner, D. Levi, G. Raz, “Recent progress in road and lane detection: a survey.“, *Mach. Vis. Appl.*, vol 25, no. 3, pp.727-745, 2014.

[13] M. Bertozzi, A. Broggi, “GOLD: A parallel real-time stereo vision system for generic obstacle and lane detection.“, *IEEE Trans. Image Process*, vol. 7, no. 1, pp. 62-81, Jan 1998.

[14] M. Aly, "Real time detection of lane markers in urban streets.", *IEEE Intelligent Vehicles Symposium*, pp. 7-12, June 2008.

[15] A. Borkar, M. Hayes, M. T. Smith, "A novel lane detection system with efficient ground truth generation.", *IEEE Trans. Intelligent Transportation Systems*, vol. 13, no. 1, pp. 365-374, March 2012.

[16] Z. Nan, P. Wei, L. Xu, N. Zheng, "Efficient Lane Boundary Detection with Spatial-Temporal Knowledge Filtering.", *Sensors*, 2016, 16, 1276.

[17] E. D. Dickmanns, B. D. Mysliwetz, "Recursive 3-D road and relative ego-state recognition", *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, no. 2, pp.199-213, Feb. 1992.

[18] Y. Wang, E. Teoh. D. Shen, "Lane detection and tracking using B-Snake", *Image Vis. Comput.*, vol. 22, no. 4, pp. 269-280, April 2004.

[19] J. Kim, M. Lee, "Robust Lane Detection Based On Convolutional Neural Network and Random Sample Consensus", *ICONIP*, pp. 454-461, 2014.

[20] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. Mujica, A. Coates, A. Y. Ng, "An empirical evaluation of deep-learning on highway driving", *arXiv preprint arXiv*: 1504.01716v3, 2015.

[21] B. He, R. Ai, Y. Yan, X. Lang, "Accurate and robust lane detection based on Dual-View Convolutional Neutral Network", *Intelligent Vehicles Symposium*, pp. 1041-1046, 2016.

[22] J. Koushik, "Understanding convolutional neural networks", *arXiv preprint arXiv*: 1605.09081, 2016.

[23] X. Pan, J. Shi, P. Luo, X. Wang, "Spatial as deep: Spatial CNN for traffic scene understanding", *arXiv preprint arXiv*: 1712.06080, 2017.

[24] D. Neven, B. De Brabandere, S. Georgoulis, M. Proesmans, L. Van Gool, "Towards end-to-end lane detection: an instance segmentation approach.", *arXiv preprint arXiv*: 1802.05591, 2018.

[25] Y. Hou, "Agnostic lane detection", *arXiv preprint arXiv*: 1905.03704, 2019.

[26] Y. Hou, Z. Ma, C. Liu, C. Loy, "Learning lightweight lane detection CNNs by self attention distillation", *arXiv preprint arXiv*: 1908.00821, 2019.

[27] S.Tuohy, D.O'Cualain, E. Jones, M.Glavin, "Distance Determination for an Automobile Environment using Inverse Perspective Mapping in OpenCV", *ISSC* 2010, UCC, June 23-24.