



**Politecnico
di Torino**



Politecnico di Torino

Mechatronic Engineering

A.a. 2025/2026

Graduation Session March 2026

Autonomous Lane Keeping: Design and Experimental Validation of an ADAS Prototype

Master's Degree in Mechatronic Engineering

Supervisors:

Massimo VIOLANTE
Angelo BORNEO

Candidate:

Giorgia LA ROSA

Abstract

Autonomous driving systems and Advanced Driver Assistance Systems (ADAS) are becoming increasingly relevant in modern society. The rapid development of autonomous machines has made the demand for safer, more efficient, and more intelligent mobility solutions a central topic in current research. Among the various ADAS functionalities, Automated Lane Keeping plays a key role in improving road safety by assisting the driver, especially in highway scenarios, and by providing a margin of error that can help prevent many accidents.

This thesis focuses on the study and implementation of an Automated Lane Keeping ADAS prototype. The work was carried out in collaboration with MCA Group, a company operating in the automotive sector and closely involved with ADAS technologies. The company provided a ROSMASTER X3, an educational robot, which was used as the experimental platform for the implementation of the lane keeping system.

The proposed system integrates computer vision and image processing techniques with trajectory generation and obstacle detection algorithms. All components were implemented within a ROS-based software architecture. Several state-of-the-art lane detection approaches were analyzed, combined, and adapted in order to achieve the best possible performance. Since many existing algorithms are designed for real vehicles and not for robotic prototypes, and considering the structural and computational limitations of the robot, a new algorithm was developed. Particular attention was paid to robustness against changes in lighting conditions and color configurations, allowing the robot to navigate scenarios with different floor and lane color combinations.

The objective of the thesis is to analyze and identify the key requirements and characteristics of an Automated Lane Keeping system. As the implementation was carried out on a prototype platform, the goal was not to develop a solution directly applicable to a real vehicle, but to study the essential features such a system must have, including robustness, reliability, adaptability, and precision.

The project also provided the opportunity to work directly with sensors commonly used in real ADAS applications, allowing an analysis of their strengths and limitations. The thesis work is mainly divided into two parallel and interdependent parts. The first focuses on computer vision, aiming to achieve accurate, reliable, and adaptive lane detection under real-time constraints, without overloading the system and causing delays that could lead to lane departure. The second part concerns motor control, which had to be tightly coupled with the perception module. Due to the limited field of view required to reduce noise and detection errors, it

was not possible to generate a continuous trajectory; instead, a control strategy was adopted that allows the robot to remain within the detected lane boundaries.

Final experimental tests demonstrated good stability, repeatability, and reliability of the system. The ROSMASTER X3 is now able to operate in different environmental conditions without significant issues related to lighting variations, and it exhibited an overall acceptable behavior with a low error margin despite the intrinsic limitations of the robotic platform.

Acknowledgements

Desidero ringraziare sinceramente MCA Group per l'opportunità che mi è stata offerta, che mi ha permesso di crescere sia dal punto di vista accademico sia professionale. In particolare, ringrazio l'ing. Angelo Borneo, supervisore di questa tesi, per la fiducia riposta in me e per la considerazione dimostrata verso il lavoro svolto; il suo contributo e la sua guida sono stati fondamentali per la buona riuscita del progetto.

Ringrazio inoltre il professor Massimo Violante, relatore accademico della tesi, per il supporto e per i preziosi consigli forniti durante lo sviluppo del progetto.

Vorrei ringraziare inoltre gli ing. Benedetto La Rosa, Halvard Besuelle e Matteo Vatteroni, per il supporto tecnico e morale che mi hanno dato durante questi mesi di duro lavoro.

Table of Contents

List of Tables	V
List of Figures	VI
Glossario	VIII
1 Introduction	1
1.1 Overview and Motivations	2
1.2 Thesis Outline	3
1.3 Requirements of the Project	4
2 State of Art	6
2.1 Automation Levels	6
2.1.1 Levels of Automation	6
2.2 ADAS System	8
2.2.1 Pedestrian Safety System	8
2.2.2 Adaptive Light Control	9
2.2.3 Park Assistant	9
2.2.4 Forward Collision Warning	9
2.2.5 Automatic Emergency Brake	10
2.2.6 Adaptive Cruise Control	10
2.2.7 Lane Keeping System	11
2.2.8 Traffic Sign and Traffic Light Recognition	11
2.3 Used Sensors	12
2.3.1 LiDAR Technology	13
2.3.2 Radar	14
2.3.3 Ultrasonic Sensors	15
2.3.4 Camera Systems	16
2.3.5 Inertial Measurement Unit (IMU)	16

3	Rosmaster X3	18
3.1	Main components	19
3.1.1	Jetson Nano	19
3.1.2	ROS2 Robot Expansion Board	19
3.1.3	RPLIDAR	20
3.1.4	AstraPlus	21
3.1.5	Rosmaster Library	21
4	Computer Vision	24
4.1	Bird's Eye View Transformation	25
4.2	Mask Generation	25
4.3	Canny Method	26
4.4	Line Detection and Hough Transform	27
5	Software Implementation	29
5.1	System Architecture	29
5.2	Lane Keeping	30
5.2.1	Image Processing	30
5.2.2	Steering Angle Computation	34
5.2.3	MQTT Communication	39
5.2.4	Interface	39
5.3	Follow Road	41
6	Tests and results	46
6.1	Adaptability of the Robot	46
6.2	Path Following Performance	47
6.3	Robot limitation	48
7	Conclusion and future works	51
A	Lane_Keeping	55
B	Follow_Road	73

List of Tables

1.1	System Requirements and Satisfaction	5
2.1	Summary of common ADAS functions and their main characteristics	12

List of Figures

2.1	Current technologies for Assisted Driving	8
2.2	Principle of LiDAR technology	14
2.3	Echolocation used by ultrasonic sensor	16
3.1	ROSMaster X3	18
3.2	RPLiDAR A1	20
3.3	AstraPlus camera	21
3.4	PID controller	22
4.1	Horizontal Sobel stencil	27
4.2	Vertical Sobel stencil	27
5.1	Before BEV	31
5.2	After BEV	31
5.3	Bird's Eye View transformation effect	31
5.4	Algorithm of lane detection	34
5.5	Interface	40
5.6	Gain variation	42
5.7	LiDAR Range	44
5.8	LEDs colors	45
6.1	Full path behaviour	48
6.2	Blind Spots of the robot	49

Glossario

ADAS

Advanced Driver Assistance Systems

TTC

Time To Collision

CWS

Collision Warning Systems

FCW

Forward Collision Warning

AEB

Automatic Emergency Brake

ACC

Adaptive Cruise Control

LCA

Lane Change Assistant

LKS

Lane Keeping Systems

LiDAR

Light Detection and Ranging

FoV

Field of View

ToF

time of flight

FMCW

Frequency Modulated Continuous Wave

IMU

Inertial Measurement Unit

BEV

Bird's Eye View

IPM

Inverse Perspective Mapping

RGB

Red, Green, Blue

HSV

Hue, Saturation, Value

PHT

Probabilistic Hough Transform

Chapter 1

Introduction

Autonomous driving systems are becoming a central element of modern society. Their relevance is growing rapidly, and they are increasingly seen as indispensable, while simultaneously representing a significant field of research and technological development.

The demand for autonomous robots capable of perceiving, understanding, and interacting with their environment continues to rise. The recent integration of artificial intelligence has demonstrated how the combination of machine learning with robotics can enable advanced autonomous navigation systems.

Beyond automotive applications, AI-enhanced robotics has potential in multiple sectors. In manufacturing, for example, autonomous systems can improve the efficiency of transport, storage, and production processes. In urban mobility, they can increase safety in public and private transport, reduce traffic congestion, and encourage a lower reliance on private vehicles. In the medical field, autonomous robots can assist people with disabilities, improving mobility and accessibility.

Robotics and machine learning are now closely intertwined and evolve together. Among all domains, vehicles have been early adopters of automation due to their ubiquity and the significant safety risks associated with human error, which remains a major cause of fatalities on the road.

In 2026, the autonomous vehicle market offers vehicles belonging to the third level of assisted driving systems, which allow drivers to take their hands off the wheel and shift their attention under specific conditions, such as driving on highways below 60 km/h. Among the leading implementations are Mercedes-Benz's Drive Pilot, available in the S-Class and EQS models; BMW's Personal Pilot L3 on the Series 7; Tesla's Autopilot and Full Self-Driving (FSD) systems, continuously updated but often classified as advanced Level 2; Ford's BlueCruise; Stellantis' STLA AutoDrive systems; and, in certain geographic areas of the United States, Level 4 robotaxi services provided by Waymo and Cruise.

ADAS systems cover a very wide range of functionalities, from basic systems

such as parking sensors that alert drivers to nearby obstacles, to more complex solutions capable of object recognition as lane keeping, and fully autonomous navigation.

This thesis was developed with the aim of studying and implementing a prototype Advanced Driver Assistance System. The work was carried out in collaboration with MCA Group, a company specialized in ADAS development for the automotive sector. Within this collaboration, MCA Group provided a robotic platform that enabled the development and experimental evaluation of a prototype system, with a specific focus on lane following and automated lane keeping functionalities. The activities presented in this thesis concentrate on the design, implementation, and testing of these functionalities on the ROSMASTER X3 robotic platform.

It is important to underline that the ROSMASTER X3 was not originally conceived for automotive ADAS applications and therefore presents both structural and computational limitations. These constraints prevented the complete and optimal realization of an automated lane keeping system comparable to those deployed in real vehicles. Such limitations are inherent to the use of a robotic prototype and are analyzed in detail in the final testing and future work chapter, where their impact on the overall system performance is discussed.

Finally, the prototype was tested in different environments and under various conditions, including multiple trajectories and the presence of obstacles. These tests were designed to evaluate the robustness, reliability, and control performance of the implemented system, highlighting both its capabilities and its limitations in realistic operating scenarios.

1.1 Overview and Motivations

The motivations for this project are multiple. Primarily, developing a prototype ADAS allows for a deeper understanding of the requirements for deploying such systems in vehicles, including sensor types, sensor placement, and system specifications. ADAS must meet stringent requirements: they need to be robust, insensitive to changing lighting conditions, and adaptable to diverse environmental scenarios such as rain, fog, or low light. Furthermore, ADAS systems must operate with minimal error, as even small mistakes can have serious consequences.

In real-world applications, an Automated Lane Keeping system cannot simply drift into an adjacent lane and attempt to correct itself later, as this behavior could result in collisions or other hazardous situations. Unlike a controlled simulation, actual traffic conditions do not allow for trial-and-error corrections; therefore, ADAS systems must operate with a high degree of precision and reliability from the outset. The primary objective of this thesis is to identify and analyze the fundamental requirements for achieving accurate, robust, and error-free lane keeping.

This involves understanding the necessary sensor capabilities, control strategies, and software implementations that ensure safe operation under a wide range of conditions. Through careful design, implementation, and testing on a controlled robotic prototype, this project seeks to illustrate how these requirements can be systematically addressed, providing a foundation for the development of reliable lane keeping systems in real automotive applications.

1.2 Thesis Outline

The thesis is structured to provide a clear and logical progression from theoretical foundations to practical implementation and testing. Each chapter is described in the following:

- **Chapter 1 - Introduction:** Presents an overview of autonomous driving systems, the motivations behind ADAS development, and the objectives of this thesis project, including the Automated Lane Keeping prototype on the ROSMASTER X3.
- **Chapter 2 - State of the Art:** Reviews automation levels and current ADAS systems, highlighting their functionalities, including pedestrian safety, adaptive lights, park assist, forward collision warning, automatic emergency braking, adaptive cruise control, lane keeping, and traffic sign/light recognition.
- **Chapter 2.3 - Used Sensors:** Describes the main sensors used in ADAS, including LiDAR, radar, ultrasonic sensors, camera systems, and the inertial measurement unit (IMU), with their operational principles and application contexts.
- **Chapter 3 - ROSMASTER X3:** Provides a detailed description of the robotic platform, including its main components such as the Jetson Nano, ROS2 expansion board, RPLidar, and AstraPlus camera, explaining how they enable the implementation of the Automated Lane Keeping system.
- **Chapter 4 - Computer Vision:** Explains the computer vision techniques used for lane detection, including bird's eye view transformation, mask generation, Canny edge detection, region of interest selection, and line detection with the Hough transform.
- **Chapter 5 - Software Implementation:** Presents the system architecture and the implementation of the control algorithms in Python, including line following, road following, and AI-based lane tracking strategies.

- **Chapter 6 - Tests and Results:** Discusses the experimental setup, testing procedures, challenges encountered, and the evaluation of the lane keeping prototype's performance.
- **Chapter 7 - Conclusion and Future Works:** Summarizes the outcomes of the thesis, highlights key findings, and suggests directions for future research and improvements in ADAS lane keeping systems.

1.3 Requirements of the Project

The first fundamental step in any engineering project is to clearly identify the required system requirements and specifications. In this case, the company that commissioned the project provided a significant degree of freedom in the design approach. Therefore, the initial step consisted of defining a comprehensive list of requirements that the robot was expected to fulfill upon completion of the project.

At the conclusion of this thesis, the initial requirements will be reviewed and analyzed to assess which ones have been satisfied, which have not, and the reasons behind any unmet requirements. The first fundamental step in any engineering project is to clearly identify the required system requirements and specifications. In this case, the company that commissioned the project provided a significant degree of freedom in the design approach. Therefore, the initial step consisted of defining a comprehensive list of requirements that the robot was expected to fulfill upon completion of the project.

At the conclusion of this thesis, the initial requirements will be reviewed and analyzed to assess which ones have been satisfied, which have not, and the reasons behind any unmet requirements.

ID	Description	Satisfied
FR-1	The system shall acquire sensor data (LiDAR, camera, battery voltage) in real time.	Yes
FR-2	The system shall process perception data to estimate the robot state and its position relative to the environment.	Yes
FR-3	The system shall detect lane markings and extract a reference trajectory.	Yes
FR-4	The robot shall follow the detected lane trajectory.	Yes
FR-5	The system shall detect obstacles along the robot path.	Yes
FR-6	The robot shall stop safely when an obstacle is detected.	Yes
FR-7	The robot shall start moving only if a lane is correctly detected.	Yes
FR-8	The system shall be compatible with the mechanical and electronic structure of the robot.	Yes
NFR-1	The system shall be robust to variations in lighting conditions.	Yes
NFR-2	The system shall be stable during continuous operation.	Yes
NFR-3	The system shall ensure smooth and continuous robot motion.	No
NFR-4	The system shall be adaptable to different environments.	Yes
NFR-5	The system shall be adaptable to different floor and lane colors.	Yes
NFR-6	The system shall be user-friendly and easily usable by non-developer users.	Yes
NFR-7	The system shall be for all the path inside the line	No

FR = Functional Requirement, NFR = Non-Functional Requirement

Table 1.1: System Requirements and Satisfaction

Chapter 2

State of Art

2.1 Automation Levels

The concept of automation levels is fundamental for understanding the transition from manually operated vehicles to fully autonomous systems. According to the definition of *Satchell* (1998), automation is defined as:

“Automation is the replacement of human activity by machine activities.”

The concept of automation has expanded significantly and now encompasses computer-based systems capable of interpreting data, making decisions, and visualizing information across multiple domains, including the automotive sector.

A fundamental aspect of automation design lies in understanding the respective strengths of human and machine performance. This distinction guides the allocation of functions between humans and automated systems, enabling the development of transportation systems that are both robust and flexible.

2.1.1 Levels of Automation

The concept of *Level of Automation* (LoA) has been extensively investigated across different domains, including aviation, robotics, and industrial process control. Due to the diversity of application fields, definitions and classification criteria vary significantly. While some approaches describe automation as a continuous spectrum ranging from manual to fully automated operation, others define discrete levels to characterize the degree of human involvement. [1]

One of the earliest and most influential classifications was proposed by Sheridan [2], who introduced a ten-level taxonomy describing the interaction between human operators and automated systems:

1. The human considers alternatives, makes decisions, and implements actions.
2. The computer offers a set of alternatives, which the human may ignore.
3. The computer offers a restricted set of alternatives, and the human selects one.
4. The computer suggests one alternative, while the human makes the final decision.
5. The computer suggests an alternative and executes it if the human approves.
6. The computer makes decisions, allowing the human to veto before execution.
7. The computer makes and implements decisions and informs the human afterward.
8. The computer makes and implements decisions, informing the human only upon request.
9. The computer makes and implements decisions and informs the human only when deemed necessary.
10. Full automation, where the computer makes and executes decisions autonomously.

In the specific context of autonomous driving systems, the automotive industry has converged toward a simplified five-level classification, which more effectively reflects practical vehicle functionalities:

1. **Driver only:** The driver maintains full control of the vehicle, with no assistance or warning systems.
2. **Driver assisted:** The driver remains in control while the vehicle provides visual or acoustic assistance, for example during lane change maneuvers.
3. **Semi-automated:** Partial vehicle control is automated under specific conditions, such as Adaptive Cruise Control or parking assistance systems.
4. **Highly automated:** Most driving tasks are performed by the system, while the driver remains available to take control when required.
5. **Fully automated:** The vehicle operates autonomously in all driving scenarios without human intervention.

Currently, most commercial systems are classified as highly automated, and the driver retains the ultimate decision-making authority.

2.2 ADAS System

Advanced Driver Assistance Systems (ADAS) have emerged as critical technologies in modern vehicles, aiming to improve road safety by mitigating human error, which accounts for approximately 90 of traffic accidents. These systems monitor various parameters associated with the vehicle and its surroundings to detect potentially dangerous situations at an early stage, thereby assisting drivers in making safer decisions or, in some cases, taking control of specific vehicle functions. The primary objective of the development of ADAS is to reduce the frequency and severity of traffic accidents by supplementing or replacing human input, which is often compromised by distraction, fatigue, or incorrect judgment. [3]

The evolution of ADAS technologies represents a significant step toward fully autonomous vehicles. Some researchers are focused on developing human-centric intelligent driver assistance systems based on cognitive knowledge, using databases of information from expert drivers to compare with current driver behavior. This approach allows for sophisticated systems that can adapt to different driving scenarios and provide appropriate assistance when needed.

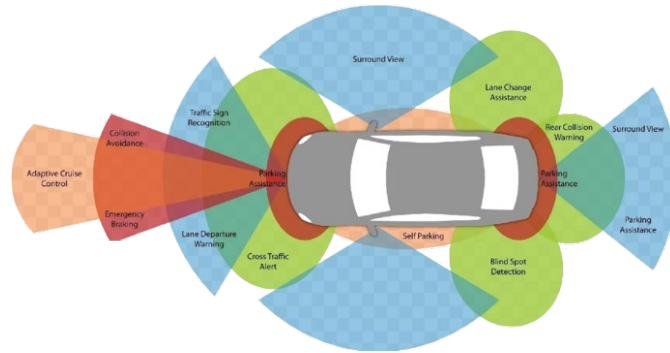


Figure 2.1: Current technologies for Assisted Driving

The figure 2.2 shows the sensors required for a fully autonomous vehicle and the specific aspects of the environment that each of them is designed to monitor.

2.2.1 Pedestrian Safety System

Pedestrian detection systems are predominantly deployed in urban environments where pedestrians considered vulnerable road users due to their lack of protection and potential unawareness of dangerous situations are common. The initial pedestrian detection systems utilized stereo-vision cameras, employing techniques such as motion-based object detection and pedestrian recognition to provide suitable measurements of the time to collision (TTC). These systems are classified as Collision Warning Systems (CWS); however, considering the relatively slow human

reaction time (approximately 2 seconds), they typically have access to the vehicle's brake system to facilitate prompt response to avoid hazards.

Modern implementations often rely on deep learning based detection and tracking algorithms, which improve robustness in complex environments and varying lighting conditions. These systems can also incorporate sensor fusion (e.g., camera and radar) to reduce false positives and enhance detection accuracy.

2.2.2 Adaptive Light Control

Adaptive light control systems have been increasingly incorporated into commercial vehicles. Their primary advantage lies in the ability to adjust illumination patterns according to road conditions (straight or curved). For instance, Adaptive High Beam technology can rotate headlights in the direction of curves, anticipating potential obstacles such as bicycles or pedestrians. Map-supported frontal lighting systems dynamically adapt headlamps by adjusting reflector positions based on environmental conditions and GPS information.

Such systems contribute to improved night-time visibility and safety, especially in scenarios where road geometry changes frequently. The adaptive lighting logic can also consider vehicle speed and steering angle to optimize illumination and reduce glare for oncoming traffic. [4]

2.2.3 Park Assistant

Park Assistant technology represents one of the most widely utilized and demanded ADAS features in contemporary vehicles. Ultrasonic park assist systems have rapidly transitioned from high-end to mainstream vehicles. These systems facilitate parking maneuvers in confined spaces by alerting drivers to rear obstacles and their proximity to the vehicle. Intelligent park assist functionality identifies suitable parking spaces and automatically steers the vehicle into position. These systems operate under driver supervision, allowing manual override through accelerator or brake pedal intervention.

More advanced versions use cameras and image processing to provide a surround view, enabling accurate obstacle detection and precise trajectory planning. These systems usually rely on path planning algorithms to generate the steering trajectory required to park the vehicle safely.[4]

2.2.4 Forward Collision Warning

Forward Collision Warning systems assist drivers in avoiding or mitigating rear-end collisions by presenting audible, visual, and/or haptic alerts. FCW systems

incorporate forward-looking vehicle detection capabilities facilitated by technologies such as radar, lidar, or cameras.

FCW logic typically involves threat assessment, where the system evaluates the relative speed and distance of the leading vehicle. If the risk of collision exceeds a defined threshold, the system triggers a warning to the driver. Modern FCW systems may also incorporate predictive models to estimate future vehicle trajectories. [4]

2.2.5 Automatic Emergency Brake

Automatic Emergency Brake systems (AEB), sometimes referred to as Collision Warning with Auto Brake or Automatic Emergency Brake, represent sophisticated ADAS technology capable of applying braking force autonomously without driver intervention. AEB systems continuously monitor the area ahead of the vehicle and, upon identifying a significant collision risk, alert the driver to initiate braking while simultaneously pre-activating the brakes.

AEB logic typically includes both risk assessment and intervention thresholds. The system calculates the probability of collision based on relative speed, distance, and time-to-collision (TTC), and it decides whether to perform partial or full braking depending on the urgency.

These systems employ various sensor technologies:

- Short-range lidar (light detecting and ranging), which offers a cost-effective solution effective at speeds up to 50 km/h.
- Long-range radar (radio detecting and ranging), which is essential for adaptive cruise control functionality and remains effective in critical situations at speeds up to 200 km/h.

2.2.6 Adaptive Cruise Control

Adaptive Cruise Control allows drivers to adjust their speed and distance from preceding vehicles without manual throttle or brake pedal operation. Traditional Cruise Control systems maintained vehicle speed automatically by controlling the accelerator pedal (longitudinal control). Adaptive Cruise Control represents a more advanced implementation that manages longitudinal vehicle control, enabling it to follow a lead vehicle, autonomously operating throttle and brake pedals, while maintaining a predefined headway.

Further evolution of this technology involves inter-vehicle cooperation to reduce following distances and mitigate the accordion effect in traffic congestion. Initial ACC systems utilized laser technology, which proved unreliable in adverse weather conditions. Consequently, contemporary ACC systems predominantly employ

Long Range Radar, which functions effectively across a broader range of weather conditions and can detect non-reflective vehicles. Various ACC implementations exist, including more economical systems without stop-and-go capability (meaning the vehicle cannot automatically come to a complete stop and subsequently resume motion), which typically operate within specific speed ranges (e.g., 50–200 km/h).

More advanced ACC versions incorporate stop-and-go functionality and may be integrated with lane keeping to support highway driving automation.

2.2.7 Lane Keeping System

Lane change maneuvers represent one of the most hazardous driving scenarios. Lane Change Assistant (LCA) systems warn drivers about dangerous lane change situations when turn indicators are activated, utilizing visual elements in side mirrors, haptic warnings through the steering wheel, or acoustic signals.

Lane Keeping Systems are designed to reduce high-speed collisions, particularly on highways, and to prevent run-off-road accidents. These systems utilize optical recognition of road markings, typically white lines. Due to their reliance on optical recognition, these systems are susceptible to road marking quality and weather conditions, potentially resulting in reduced reliability during heavy rain, snow, or excessive solar glare. LKS represents a proactive system that not only warns drivers when vehicle drift is detected but can also adjust steering to control vehicle direction, thereby reducing accident risk when driver attention is compromised.

The Automated lane keeping system, is an autonomous driving system that doesn't require driver supervision on motorways. [5]

The lane keeping process typically follows these steps:

1. Main lane recognition: identification of primary lane boundaries.
2. Neighbor lane recognition: detection of adjacent lanes to assist with lane keeping in critical traffic conditions.
3. Lane-marking classification: identification of road-marking types demarcating the current lane.

2.2.8 Traffic Sign and Traffic Light Recognition

Traffic sign and traffic light recognition systems process outdoor imagery using various image processing and segmentation techniques (e.g., color analysis, shape analysis) to identify traffic control devices under daylight conditions in real-world scenarios, contributing to environmental perception capabilities.

Modern implementations often employ deep learning-based classification methods to improve robustness under variable lighting and occlusion conditions. These

systems can also integrate temporal filtering to stabilize detections across multiple frames.

ADAS Function	Main Purpose	Typical Sensors / Technologies
Pedestrian Safety System	Detect pedestrians and estimate TTC to prevent collisions	Stereo-vision cameras, radar, deep learning algorithms
Adaptive Light Control	Adapt illumination to road geometry and visibility conditions	Headlight actuators, GPS, map data, vehicle dynamics
Park Assistant	Support parking maneuvers and detect obstacles	Ultrasonic sensors, cameras, path planning algorithms
Forward Collision Warning	Warn driver of imminent frontal collisions	Radar, lidar, cameras, threat assessment logic
Automatic Emergency Braking	Autonomously apply braking in high-risk situations	Radar, lidar, braking control system
Adaptive Cruise Control	Maintain speed and safe distance from lead vehicle	Long-range radar, vehicle-to-vehicle communication (advanced versions)
Lane Keeping System	Prevent lane departure and support lane following	Cameras, lane detection algorithms, steering control
Traffic Sign and Light Recognition	Detect and classify traffic signs and signals	Cameras, image processing, deep learning classification

Table 2.1: Summary of common ADAS functions and their main characteristics

2.3 Used Sensors

This section presents the main sensors used in ADAS. It should be noted that, since 2014, regulations have made certain ADAS systems mandatory for all newly registered vehicles, so understanding which sensors are fundamental in a new car and which are the difference is essential to understand ADAS systems and their development.

The required systems include Advanced Emergency Braking Systems (AEB), Lane Keeping (LK), and Intelligent Speed Assistance (ISA), a system that reads speed limits through cameras and provide warnings or assisting the driver in maintaining the correct speed.

As a result, all new vehicles must be equipped with specific sensors, making it essential to understand how they operate and why they are used for different functions. The following sections will describe lidar, radar, ultrasonic sensors, IMU, and cameras.

2.3.1 LiDAR Technology

Light Detection and Ranging (LiDAR) is a scanning technology that, using a transmitter to emit laser pulses and a receiver to capture them, can calculate not only the distance to objects but also reconstruct a three-dimensional representation of the surrounding environment. [6]

As the distance to the target changes, the reflected laser spot shifts on the detector. By measuring this displacement, the distance can be calculated using trigonometric relationships, since the incident and reflected rays form a triangle. This method allows precise distance measurement over short ranges and is commonly used in applications requiring high accuracy, such as object profiling, industrial automation, and small-scale robotics.

The Field of View (FoV) is defined as the angular range over which LiDAR signals are emitted. For a comprehensive three-dimensional representation of a vehicle's surroundings, LiDAR sensors must provide both vertical and horizontal FoV.

LiDAR is a highly precise sensor capable of long-range operation. Depending on the emission angle of the laser signal, the Field of View can be wider or narrower. It is suitable for understanding the three-dimensional environment around a vehicle, detecting objects and people, estimating their distance, and, in combination with cameras, creating a 3D map closely resembling reality.

LiDAR sensor performance is characterized by:

- Precision: The repeatability of measurements
- Accuracy: The geographical or positional correctness of measurements
- Resolution: The minimum angular or linear separation between two points that the sensor can distinguish

The two main techniques used by LiDAR to achieve this are the Time of Flight method and the Triangulation method.

Time of Flight Method This method measures distances by calculating the round-trip delay of light signals emitted from a laser toward a target. The range (R) to the target can be determined using the equation:

$$R = \frac{1}{2}c\tau$$

where c is the speed of light and τ is the round-trip delay of the emitted signal. This delay, known as the *time of flight* (ToF), can be obtained by modulating the frequency, phase, intensity, or other characteristics of the transmitted light and measuring the time required for that modulation pattern to be detected by the receiver. The detection range refers to the maximum distance at which an object can be identified, which largely depends on the sensor's transmitted power. Range is typically expressed as minimum and maximum values.

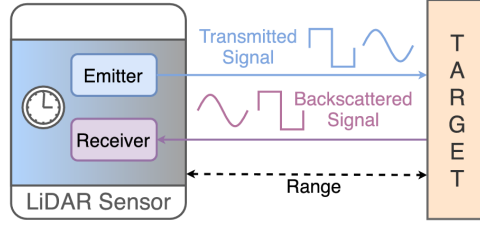


Figure 2.2: Principle of LiDAR technology

Triangulation Method Another common principle, particularly in short-range single-line LiDAR systems, is the laser triangulation method. In this approach, a laser beam is projected at a known angle onto a target surface. The reflected light is captured by a lens and focused onto a position-sensitive detector, typically a CCD sensor.

2.3.2 Radar

Initially, radar sensors were mainly used for crash avoidance applications. Nowadays, they are employed in a much wider range of ADAS functions, such as Adaptive Cruise Control (ACC), Automatic Emergency Braking (AEB), Blind Spot Detection (BSD), and Lane Change Assist (LCA).

Radar sensors can operate at short, medium, or long range, with a detection range typically between 30 and 250 m, and they use electromagnetic waves in the radio-frequency band, commonly at 24, 77, or 79 GHz. The transmitted wave is reflected by surrounding objects and received by the sensor; similarly to lidar, the distance to the object is estimated through time-of-flight measurements.

A key strength of radar sensors lies in their ability to measure not only distance but also the relative velocity and direction of motion of detected objects, thanks to the frequency shift caused by the Doppler effect.

This characteristic makes both lidar and radar indispensable for autonomous vehicles. Although these sensors may appear similar or interchangeable, they provide complementary information: lidar is capable of reconstructing a detailed 3D representation of the environment and is therefore essential for highly or fully automated vehicles, while radar remains crucial even in semi-automated vehicles, despite lidar offering higher spatial resolution.

Moreover, radar sensors are less affected by adverse environmental conditions such as fog, rain, or low-light scenarios compared to optical sensors. For this reason, radar is considered an essential technology for the reliable detection of vehicles, pedestrians, and cyclists.

Modern automotive radar systems often employ Frequency Modulated Continuous Wave (FMCW) technology, which transmits a continuous signal with varying frequency. FMCW radar enables high resolution and accurate measurement of both distance and velocity, even in complex driving scenarios.

2.3.3 Ultrasonic Sensors

Like the other sensors previously discussed, ultrasonic sensors are capable of detecting objects and obstacles without physical contact. Depending on the sensor and the properties of the target object, their effective range extends from a few centimeters up to several meters, making it much shorter than the range of lidar or radar. Ultrasonic sensors, or transducers, generate and emit ultrasonic pulses that are reflected back when they encounter an obstacle.

Ultrasonic sensors are piezoelectric transducers, meaning they can convert an electrical signal into mechanical vibrations. They operate both as a microphone and as a speaker at a single frequency, following a monostatic approach. The sensor measures the time difference between the transmitted pulse and its echo, allowing it to calculate the distance to the object.

Note that the velocity of sound through air varies with temperature. In dry air at 20°C (68°F), the speed of sound is 343 m/s, which corresponds to approximately 1 kilometer in 2.91 seconds.

Ultrasonic sensors are therefore not used for long-range detection but are suited for short-range applications. They are commonly employed in systems such as Park Assist and, in some cases, Blind Spot Detection. The figure 2.3 below demonstrates the working principle of ultrasonic technology

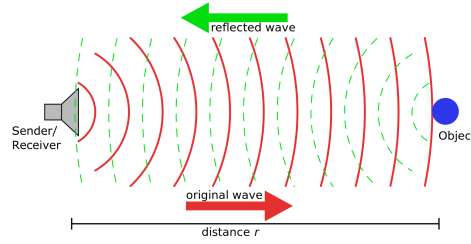


Figure 2.3: Echolocation used by ultrasonic sensor

2.3.4 Camera Systems

Camera systems are fundamental components of ADAS, providing essential information for object recognition. The most commonly used cameras in ADAS are **monocular cameras**, which offer a single viewpoint and are mainly employed for traffic sign recognition, lane keeping, and pedestrian detection.

Stereo cameras, equipped with two lenses, also provide depth perception and are therefore used for distance estimation, supporting functions such as Forward Collision Warning (FCW) and Automatic Emergency Braking (AEB).

Surround-view systems integrate multiple cameras to achieve a 360° view, assisting in parking and low-speed maneuvers.

Cameras operate by capturing high-resolution images that are processed using deep learning algorithms to detect and classify objects. Some advanced systems also incorporate **infrared imaging**, enabling effective performance in low-light conditions.

2.3.5 Inertial Measurement Unit (IMU)

An Inertial Measurement Unit (IMU) is a sensor that measures a vehicle's linear acceleration and angular velocity using accelerometers and gyroscopes. IMUs provide essential information on the vehicle's motion, orientation, and dynamics, even in conditions where GPS or visual sensors are unavailable or unreliable.

IMU sensors are essential for the following ADAS system:

- **Lane Keeping and Lane Departure Warning:** by providing accurate vehicle motion data to detect unintended lane drift.
- **Adaptive Cruise Control and Forward Collision Warning:** by supplying vehicle acceleration and deceleration information to improve distance estimation and control accuracy.
- **Dead Reckoning Navigation:** in GPS-denied environments, IMUs estimate the vehicle's position based on motion data.

IMU's outputs are typically combined with GPS, cameras, lidar, or radar measurements through sensor fusion algorithms to achieve robust and reliable vehicle perception and localization.

Chapter 3

Rosmaster X3

The ROSMASTER X3 is an educational robotic platform designed for learning and experimentation with autonomous systems. Being commercially available, it allows users to program and control it in various ways, either via keyboard commands, a wireless controller, or through autonomous or semi-autonomous navigation algorithms, as implemented in this thesis.



Figure 3.1: ROSMASTER X3

The robot is based on the Robot Operating System (ROS) and features 360° omnidirectional Mecanum wheels. It is compatible with multiple processing units, including Jetson NANO, Jetson ORIN NANO SUPER, Jetson ORIN NX SUPER, and Raspberry Pi 5. The platform is equipped with high-performance modules, such as a LiDAR sensor, a depth camera, and a voice interaction module, enabling a wide range of functionalities including mapping and navigation, obstacle avoidance,

object following, autopilot, and human posture detection. Python programming allows full customization of its behavior.

At the core of the system is the Jetson Nano 4GB, which serves as the primary processing unit. It is connected to a dedicated ROS expansion board that acts as a central hub, interfacing with all peripheral components. The LiDAR is strategically mounted to provide complete environmental scanning for 3D mapping and obstacle detection, while the depth camera delivers real-time depth information essential for navigation and object recognition.

The ROSMASTER X3 also incorporates a voice interaction module, enabling control of the robot's movement and execution of specific functions via voice commands. Its omnidirectional Mecanum wheels, each powered by independent DC motors, grant exceptional maneuverability in any direction.

Finally, the platform supports multiple remote control methods, including mobile applications, handheld controllers, ROS-based PC control, and computer keyboards. Extensive educational resources, including 103 video courses and a large repository of sample code, make it an ideal tool for learning artificial intelligence programming and ROS-based robotics.

3.1 Main components

3.1.1 Jetson Nano

The Jetson Nano 4GB SUB is the main board of ROSMASTER X3 and it is configured with the image provided by Yahboom, which provides a pre-configured environment and useful material. Developed by NVIDIA, the board is an embedded computing platform designed for edge AI applications, providing a balance between performance and power efficiency. [7]

3.1.2 ROS2 Robot Expansion Board

The ROS V3.0 expansion board is the main interface between Jetson Nano and the various hardware components of ROSMASTER X3, acting as both a communication bridge and a power management hub. Communication between Jetson Nano and the expansion board takes place via a USB serial connection, where Jetson Nano transmits serial data to the integrated microcontroller (MCU). The microcontroller, which is an STM32, is responsible for executing and interpreting commands.

The main role of the expansion board is power management. It is connected to a battery that supplies 12 V and ensures a constant 5 V to the Jetson Nano. The board must also supply voltage to the four encoder motors, enabling precise closed-loop control. [8]

3.1.3 RPLIDAR

The RPLIDAR, integrated in the robot as the A1 model, is a single-line LiDAR that primarily uses the Time-of-Flight (ToF) principle to measure distances. [9]

A single-line LiDAR emits a single laser beam and is commonly used in robotics due to its high scanning speed, reliable performance, and compact design. It operates with high angular resolution and sensitivity, making it particularly accurate for measuring distances and detecting obstacles along a planar sweep.



Figure 3.2: RPLiDAR A1

The RPLIDAR A1 is a 360-degree single-line scanning LiDAR specifically designed for cost-effective robotic applications. Unlike high-end LiDARs, it offers a scanning range of up to 12 meters in typical indoor environments and operates at a rotational speed of up to 5.5 Hz (330 RPM), providing a good balance between resolution and update rate. Its compact and lightweight design makes it ideal for educational robots and small mobile platforms, while its accuracy, approximately $\pm 2\%$ within the effective range, is sufficient for reliable obstacle detection and SLAM-based mapping.

1. The Laser is the source of LiDAR emission, which pulses and projects a laser beam into the surrounding environment.
2. When the laser beam strikes an obstacle, the receiver captures the reflected light through a lens system, preparing it for further processing.
3. The Signal processing unit monitors the laser emission and analyzes the received signals to calculate the distance to the target object based on the time-of-flight principle.
4. To enable continuous 360-degree scanning, these components are mounted on a mechanism that spins at a constant speed. This rotation allows the LiDAR to generate a 2D map of the surrounding environment in real time.

3.1.4 AstraPlus

The AstraPlus model is a depth camera. This type of camera has the unique feature of not only producing a 2D RGB image but also providing depth information and three variables (X, Y, Z) for each pixel. [10] The Astra series by Orbbec is a line of depth cameras that provide 3D sensing capabilities. These cameras are compatible with ROS 2, allowing for seamless integration into robotic systems. The Astra cameras utilize structured light technology to capture depth information, offering high-resolution depth data suitable for various applications, including object detection and environment mapping.

Depth cameras use IR (Infra-Red) technology to generate depth maps of the environment. Unlike standard RGB (Red Green Blue) cameras, depth sensors require two essential components: an IR projector and an IR camera. Although some depth sensors may include an RGB camera, this is not a fundamental requirement for depth detection. The IR projector emits a structured pattern of infrared light that illuminates objects in the scene.

Although invisible to the human eye, this pattern appears as a dense grid of dots when observed by an IR camera. The IR camera, which functions like a standard camera but in the infrared spectrum, captures this pattern as it is distorted by objects struck at different distances. The depth sensor processor then analyses these distortions to calculate depth information. Objects closer to the sensor show a more distorted pattern of points, while those further away appear with a denser distribution. By evaluating the displacement of these points, the system constructs a depth map representing the spatial geometry of the scene. This depth map can be viewed directly from the sensor and processed by a computer for applications such as 3D reconstruction, object recognition and augmented reality.



Figure 3.3: AstraPlus camera

3.1.5 Rosmaster Library

The Rosmaster libraries allow communication with the robotic platform; they contain the serial communication with the microcontroller and translate it into a set of high-level functions for motion control, configuration parameters, and the

motor system. In this section, the main libraries used and the most important functions are analyzed:

set_car_motion(self, v_x, v_y, v_z) this function is responsible for controlling the movement of the robot. In fact, it is sufficient to set the desired velocity along the three axes and the robot will execute the motion. As shown in the ROSMASTER X3 documentation, the velocity v_x in m/s represents the linear velocity of the robot, therefore forward or backward motion; the velocity along y , v_y , represents the lateral velocity in m/s that the robot can achieve thanks to the omnidirectional wheels; and the velocity v_z represents the angular velocity in rad/s. Thanks to the wheels, the robot does not require a steering mechanism, but can turn by simply varying the speed of each wheel.

set_pid_param(self, kp, ki, kd, forever=False) inside the robot, a PID controller is already implemented. Through this library, it is possible to adjust the three parameters of the controller. It consists of three parts:

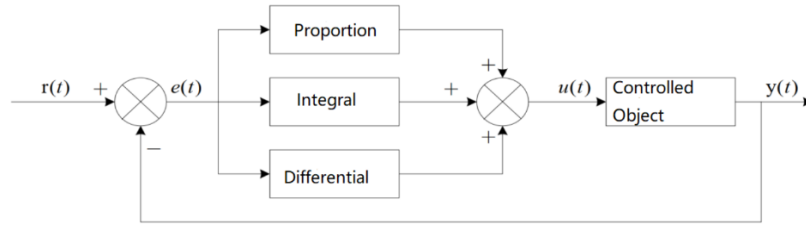


Figure 3.4: PID controller

The **Proportional** part K_p , which is responsible for reacting instantaneously to sudden changes. As the proportional coefficient increases, the strength of the controller effect increases, but oscillations also increase and the stability of the system is degraded.

$$k_p \cdot e(t)$$

Advantages: adding the open-loop proportional coefficient of the system improves steady-state precision and increases the response speed of the system.

Disadvantages: using only a P controller may make the system unstable and increases the system overshoot.

The **Integral** part K_i is such that the higher the integral constant, the weaker its effect. Increasing the integrator constraint makes the system slower in eliminating the deviation but can increase the stability of the system.

$$\frac{k_p}{T_i} \int_0^t e(t) dt$$

The **Differential** K_d part has the role of reducing overshoot, oscillations, and stabilizing the system, especially for higher-order systems. However, the differential action is very sensitive to noisy input signals. Thanks to its contribution, the system response speed is higher and a “prediction” effect is introduced in the dynamic process.

$$K_p \cdot T_d \frac{de(t)}{dt}$$

It is therefore very clear that selecting the correct controller parameters is fundamental in order to obtain robot motion that is as smooth as possible and with minimal oscillations. In our case, the controller values were chosen empirically.

self, led id, red, green, blue this is a purely aesthetic function that allows changing the color of the robot LEDs according to the state in which the Rosmaster is operating.

get battery voltage(self) which simply allows reading the voltage level of the robot batteries.

Chapter 4

Computer Vision

Computer vision plays a key role in the development of self-driving vehicles, enabling the design of advanced driver assistance systems aimed at reducing road accidents and improving overall driving safety. By allowing vehicles to perceive and interpret their surrounding environment, computer vision contributes to the creation of safer and more reliable autonomous driving systems.

In robotics, computer vision can be defined as the process of extracting, characterizing, and interpreting information from images of a three-dimensional world. A vision system essentially takes an image or a video sequence as input, processes it to extract relevant information, and uses this information to generate actions for the robot or, in the automotive context, for the vehicle.

The hierarchical organization of a computer vision system is typically structured into several stages.

1. The process begins with **Perception**, which consists of acquiring the image.
2. This is followed by **Pre-Processing**, which aims at noise reduction and image enhancement.
3. **Segmentation** then divides the image into regions or objects of interest.
4. In the **description** stage, characteristic features such as shape, size, or geometric properties are extracted to distinguish one object from another.
5. **Recognition** is the process that identifies such objects
6. **Interpretation** assigns meaning to the recognized elements based on the application context.

This thesis primarily focuses on the image processing stage of a vision-based lane-following system for autonomous ground vehicles. This stage involves image

transformations, enhancement techniques, filtering, and segmentation, as well as the extraction of low-level features. In particular, emphasis is placed on lane detection through line recognition, where each acquired frame is processed to obtain a binary representation of the lane markings and to estimate a steering angle suitable for vehicle control.

The proposed approach relies exclusively on classical computer vision and image processing techniques, ensuring low computational complexity and real-time performance.

4.1 Bird's Eye View Transformation

One of the key pre-processing steps in lane detection systems is the reduction of perspective distortion introduced by the onboard camera. This is achieved through a Bird's Eye View (BEV) transformation, also known as Inverse Perspective Mapping (IPM).

The method consists in selecting a trapezoidal region of interest in the original image, corresponding to the portion of the road surface that is ahead of the vehicle. This trapezoid is defined so that its upper edge lies near the estimated horizon, while its lower edge coincides with the bottom of the image. A planar homography is then computed and applied to map the trapezoidal region into a rectangular top-down view.

This transformation provides several advantages:

- lane markings become approximately vertical and parallel;
- the lane width is represented as constant in the image;
- the apparent curvature of the lane becomes independent of distance;
- subsequent edge detection and line extraction are significantly simplified.

Moreover, the BEV transformation implicitly performs a region-of-interest selection, excluding irrelevant elements of the environment and reducing visual noise.

4.2 Mask Generation

After the BEV transformation, the image is converted from the RGB color space to the HSV (Hue, Saturation, Value) representation. The HSV color space separates chromatic information from illumination, making it more robust to lighting variations and shadows, which are common in outdoor environments.

The RGB to HSV conversion follows the standard formulation originally proposed by Smith, where pixel intensities are first normalized and then mapped into hue, saturation, and value components. In practical implementations such as OpenCV, the hue channel is normalized to the range $[0,179]$ to fit 8-bit representations.

Once converted, a multivariate thresholding operation is applied. Each pixel is classified by comparing its HSV values with predefined lower and upper bounds. Pixels belonging to lane markings are assigned a value of 255 (white), while background pixels are set to 0 (black), resulting in a binary mask highlighting the lane regions.

4.3 Canny Method

The binary mask is further processed using the Canny edge detection algorithm to extract the contours of the lane markings. The Canny operator is a multi-stage algorithm designed to provide optimal edge detection while minimizing noise sensitivity.

The algorithm consists of the following steps:

1. Apply the Gaussian filter in order to reduce noise Gaussian filters are commonly employed as a pre-processing step in image processing to reduce noise while maintaining edge information. Compared to mean filters, Gaussian filters apply a weighted average based on a Gaussian distribution, where pixels closer to the center of the kernel have a greater influence. This results in a smoother and more natural blurring effect, which helps mitigate the impact of noise on subsequent edge detection algorithms. [11]
2. Computation of the gradient using Sobel operator

The Sobel operator is commonly used to estimate the spatial derivatives of an image, allowing the detection of edges and intensity variations. This operator applies two 3×3 kernels, i.e., two convolution matrices, to the original image in order to compute approximate values of the derivatives in the horizontal and vertical directions.

Let G_x and G_y be the two resulting images, whose pixels represent, respectively, the approximated values of the horizontal and vertical derivatives. The operation is formally described as:

$$G_x = [f(x+1, y-1) + 2f(x+1, y) + f(x+1, y+1)] - [f(x-1, y-1) + 2f(x-1, y) + f(x-1, y+1)]$$

$$G_y = [f(x-1, y+1) + 2f(x, y+1) + f(x+1, y+1)] - [f(x-1, y-1) + 2f(x, y-1) + f(x+1, y-1)]$$

-1	-2	-1
0	0	0
1	2	1

Figure 4.1: Horizontal Sobel stencil

-1	0	1
-2	0	2
-1	0	1

Figure 4.2: Vertical Sobel stencil

3. Suppression of spurious edge responses in order to identify the real edges and discard the false ones.
4. Hysteresis thresholding using two thresholds to preserve strong edges connect the weak one only if they are directly connected to a strong one.

This formulation highlights the local intensity variations, which are essential for detecting the edges of the lines.

4.4 Line Detection and Hough Transform

Linear Hough Transform

The linear Hough transform is used to estimate the two parameters that define a straight line in an image. The transform space is two-dimensional and each point in this space acts as an accumulator for a potential line described by:

$$r = x \cos \theta + y \sin \theta$$

where r is the perpendicular distance from the origin to the line and θ is the angle between the x-axis and the line normal. Each edge point in the image contributes to the accumulator by voting for all possible (r, θ) pairs that satisfy the equation.

The accumulator has two dimensions, corresponding to the unknown parameters r and θ , which are discretized into quantized values. For each pixel (x, y) belonging to an edge, the algorithm evaluates whether there is sufficient evidence of a straight line passing through that pixel. If the condition is satisfied, the corresponding (r, θ) parameters are computed, the associated accumulator bin is identified, and its value is incremented.

The result of the linear Hough transform is a two-dimensional matrix, where one dimension corresponds to the quantized angle θ and the other to the quantized distance r . Each element of the matrix stores the number of edge points that lie on the line represented by the quantized parameters (r, θ) . Therefore, the bin with the highest value indicates the line that is most strongly represented in the input image.

Probabilistic Hough Transform

Lane boundary extraction is performed using the Probabilistic Hough Transform (PHT). Unlike the classical Hough Transform, PHT directly returns finite line segments instead of infinite lines, significantly reducing computational cost while maintaining detection accuracy.

The algorithm maps edge points from the image space into the parameter space and identifies groups of collinear points corresponding to lane markings. Detected line segments are then classified as left or right lane boundaries based on their slope and spatial position. Additional post-processing steps, such as filtering by length, orientation, and continuity, are often applied to remove spurious detections and to improve robustness in real driving scenarios.

Chapter 5

Software Implementation

This chapter describes the software development process adopted to enable autonomous driving functionalities. The focus is placed on the implementation logic, system organization, and design choices made to comply with the project specifications and real-time constraints.

The overall system is composed of a Python-based vision module and a ROS2-based control node. The development concludes with the integration of MQTT communication to enable real-time data exchange between perception and control layers.

5.1 System Architecture

The system was developed following a modular architectural approach to ensure efficient communication, scalability, and clear separation of responsibilities among the different software components. The architecture is divided into two main elements: a Python-based vision script running directly on the Jetson Nano, and a ROS2 node executed inside a Docker container.

The decision to separate the software into two distinct execution environments was driven by practical and performance-related considerations. Motor control and sensor management are handled exclusively within the Docker container, as it already contains all the pre-existing control libraries and interfaces required to communicate with the ROSMASTER hardware. In this context, it is not necessary to implement low-level motor control algorithms; instead, the system only needs to provide desired velocity commands along the x , y , and z axes, which are passed to dedicated control functions.

At the same time, running computationally intensive image processing algorithms inside the Docker container would introduce additional latency due to container overhead. For this reason, the entire vision pipeline was implemented as a standalone

Python script running directly on the ROSMASTER. This choice allows image processing operations to be executed with minimal delay, which is critical for real-time lane detection and steering control.

Communication between the vision module and the control node is handled through the MQTT protocol, enabling lightweight and asynchronous data exchange. From a functional perspective, the system can be divided into two main stages: perception and decision-making, performed by the Python script, and motion execution, handled by the ROS2 node.

5.2 Lane Keeping

The Python script, named *lane_keeping.py*, is responsible for managing the graphical user interface, processing camera images, extracting lane information, and computing the steering angle. The resulting control commands are transmitted in real time to the ROS2 node via MQTT communication.

The script integrates multiple processing steps into a single pipeline, including image acquisition, geometric pre-processing, color-based segmentation, edge detection, and line extraction. In addition, an interactive interface is provided to allow real-time tuning of key parameters, such as the region of interest and the HSV color thresholds, facilitating rapid adaptation to different lighting and environmental conditions.

5.2.1 Image Processing

The lane detection process begins with the acquisition of frames from the onboard camera. Each frame undergoes a sequence of pre-processing and feature extraction operations designed to enhance lane visibility and improve detection robustness under real-world conditions.

Bird's Eye View Transformation To reduce perspective distortion and simplify lane geometry, the input image is transformed into a bird's eye view using an Inverse Perspective Mapping (IPM). This transformation assumes that lane markings lie on a planar road surface and is implemented through a homography matrix.

```
1 trap = build_trap_from_rows(cam_w, cam_h, horizon_y, top_off,
    bot_mrg, l_mrg, r_mrg)
2 ipm_M = compute_ipm_transform(trap, CONFIG["BIRDEYE_SIZE"])
3 bev = cv2.warpPerspective(frame, ipm_M, CONFIG["BIRDEYE_SIZE"])
```

Listing 5.1: Inverse Perspective Mapping computation

The trapezoidal region defining the road surface is constructed using four parameters decided by the user.

1. The value `top_off` defines the vertical offset, in pixels, between the estimated horizon line and the upper edge of the trapezoid, allowing exclusion of sky and distant background.
2. The parameter `bot_mrg` defines the margin from the bottom of the image.
3. The parameters `l_mrg` and `r_mrg` define the left and right horizontal margins, respectively, and are used to isolate the lane region while discarding irrelevant lateral areas.

The destination size `BIRDEYE_SIZE` determines the resolution of the warped image. A resolution of 640×480 pixels is selected as a trade-off between spatial accuracy and computational efficiency.

In the figures 5.3 below, the same image is shown before and after the application of the Bird's Eye View transformation, highlighting the effect of the perspective mapping on the scene geometry.



Figure 5.1: Before BEV

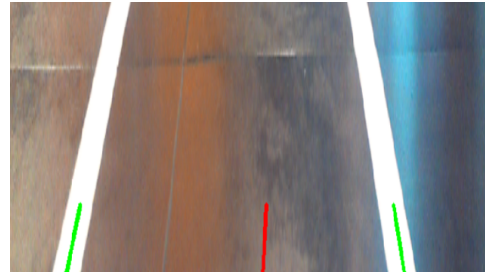


Figure 5.2: After BEV

Figure 5.3: Bird's Eye View transformation effect

Color Space Conversion After the geometric transformation, the new image is converted from the BGR color space to the HSV color space to improve robustness against illumination variations.

```
1 hsv = cv2.cvtColor(bev, cv2.COLOR_BGR2HSV)
```

Listing 5.2: BGR to HSV conversion

In the HSV representation, the Hue channel encodes color information, the Saturation channel represents color intensity, and the Value channel corresponds to brightness. This separation allows color-based segmentation to be less sensitive to shadows and non-uniform lighting compared to RGB-based approaches.

HSV Thresholding Lane segmentation is performed by thresholding the HSV image using lower and upper bounds for each channel.

```
1 mask = cv2.inRange(  
2     hsv,  
3     np.array(hsv_lower, np.uint8),  
4     np.array(hsv_upper, np.uint8)  
5 )
```

Listing 5.3: HSV thresholding for lane segmentation

The tuple `hsv_lower = (H_L, S_L, V_L)` defines the minimum acceptable values for hue, saturation, and brightness, while `hsv_upper = (H_U, S_U, V_U)` defines their maximum values.

Hue thresholds isolate the lane color, saturation thresholds suppress gray regions such as asphalt, and value thresholds prevent detection of excessively dark or overexposed pixels. These parameters are adjustable in real time through graphical trackbars, enabling rapid calibration under different environmental conditions.

Edge Detection Edges are extracted from the binary mask using the Canny edge detection algorithm.

```
1 edges = cv2.Canny(  
2     mask,  
3     CONFIG["EDGE_LOW"],  
4     CONFIG["EDGE_HIGH"]  
5 )
```

Listing 5.4: Canny edge detection

The parameter `EDGE_LOW` represents the lower hysteresis threshold, while `EDGE_HIGH` represents the upper threshold. Gradients below the lower threshold are discarded, whereas gradients above the upper threshold are immediately accepted as edges. Values between the two thresholds are accepted only if connected to strong edges.

This dual-threshold mechanism improves robustness against noise while preserving continuous lane contours.

Region of Interest Selection To reduce computational load and suppress irrelevant features, edge detection results are restricted to a region of interest corresponding to the lower half of the image.

```
1 edges_roi = roi_lower_half(edges)
```

Listing 5.5: Lower-half ROI selection

The selected region contains the most relevant information for short-term steering decisions, while distant features in the upper part of the image typically introduce perspective distortion and unreliable detections.

Line Detection Using the Hough Transform Lane boundaries are extracted from the edge image using the probabilistic Hough transform.

```

1 def detect_line_segments(cropped_edges):
2     return cv2.HoughLinesP(
3         cropped_edges,
4         CONFIG["HOUGH_RHO"],
5         CONFIG["HOUGH_THETA"],
6         CONFIG["HOUGH_THRESH"],
7         minLineLength=CONFIG["HOUGH_MIN_LEN"],
8         maxLineGap=CONFIG["HOUGH_MAX_GAP"],
9     )

```

Listing 5.6: Probabilistic Hough transform for lane detection

The parameter `HOUGH_RHO` represents the resolution of the distance parameter ρ in the Hough space, expressed in pixels. A value of one pixel is used, providing a fine spatial resolution while maintaining acceptable computational cost.

The parameter `HOUGH_THETA` defines the angular resolution of the transform and corresponds to the discretization step of the angle θ . In this work, it is set to $\pi/180$, resulting in a one-degree angular resolution that allows accurate estimation of lane orientation without excessive sensitivity to noise. The threshold parameter `HOUGH_THRESH` specifies the minimum number of votes required in the Hough accumulator for a line to be considered valid. Lower values increase sensitivity but may introduce false detections, while higher values improve robustness at the cost of potentially missing weak lane markings. The chosen value represents a compromise between these two effects. The parameter `minLineLength` defines the minimum length, in pixels, of a detected line segment. Short segments are discarded to avoid spurious detections caused by noise or small artifacts in the image. Finally, the parameter `maxLineGap` specifies the maximum allowed distance between two collinear segments to be merged into a single line. This allows the algorithm to correctly reconstruct lane boundaries even when markings are partially occluded or discontinuous.

Overall, the proposed pipeline combines geometric transformations, color-based segmentation, and classical computer vision techniques to achieve reliable lane detection in real time. The modular structure and parameterized implementation allow easy tuning and adaptation to different environments, while maintaining low computational complexity.

In the following figure 5.4, is proposed a schematic representation of the pipeline, illustrating the sequence of transformations applied to the input image and highlighting how these operations belong to the first three computer vision processes previously analyzed.

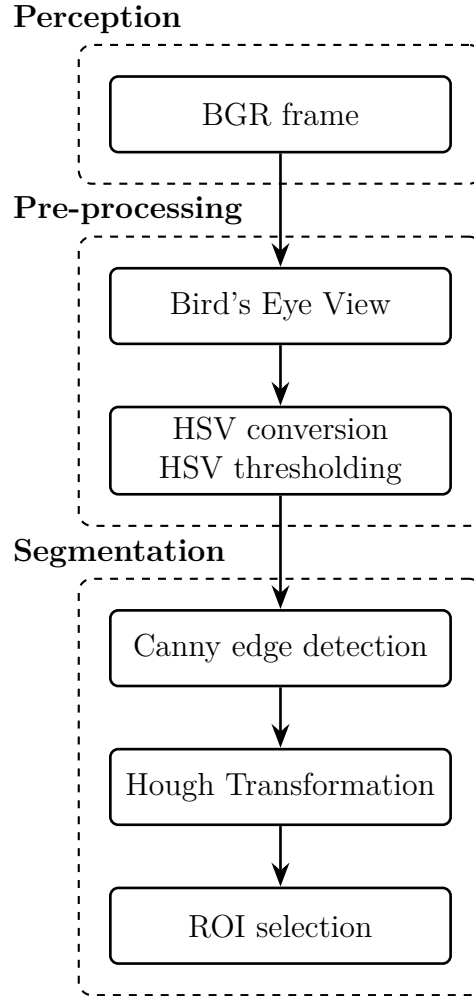


Figure 5.4: Algorithm of lane detection

5.2.2 Steering Angle Computation

At this point the lines are recognized so we have to understand the angle between the position on the ROSMASTER and the trajectory of the lines so that we can find the command to send to the motors

Lane Averaging and Classification The detected line segments returned by the probabilistic Hough transform are processed by a function whose goal is to classify and aggregate lane candidates to obtain stable lane representations.

Each segment is described by its end points (x_1, y_1, x_2, y_2) . From these coordinates, the slope and the intercept are derived as follows:

$$m = \frac{y_2 - y_1}{x_2 - x_1} \quad q = y_1 - mx_1$$

Segments for which $x_1 = x_2$ are discarded, since vertical lines would lead to infinite slopes, resulting in computational problems.

Lane classification is based on the sign of the slope and on spatial constraints. It should be noted that, after the Bird's Eye View (BEV) transformation, the image coordinate system has the origin in the top-left corner, with the horizontal axis increasing to the right and the vertical axis increasing downward.

Segments with negative slope are associated with the left line of the lane, while segments with positive slope are associated with the right one. In order to prevent incorrect associations caused by spurious detections, additional geometric constraints are applied: left-lane segments must lie in the left portion of the image, while right-lane segments must lie in the right portion. These regions are defined as fractions of the image width, to be considered a left-lane segment, a line must lie within the horizontal region

$$x \in \left[0, \frac{2}{3}w\right],$$

while to be considered a right-lane segment, it must be inside

$$x \in \left[\frac{1}{3}w, w\right].$$

For each lane side, the collected slope and intercept pairs are averaged, this operation is like a filter that attenuates the noise due to the Hough transform or a false positive. The averaged parameters, slope and intercept are then converted into pixel coordinates using the `make_points`, which reconstructs a representative lane line by computing two points at fixed vertical positions in the image.

The output of this stage is a list containing one or two lane lines, each expressed as pixel endpoints.

```

1 def average_slope_intercept(frame, segments):
2     lane_lines = []
3     if segments is None:
4         return lane_lines
5
6     h, w = frame.shape[:2]
7     boundary = 1 / 3
8     left_boundary = w * (1 - boundary)
9     right_boundary = w * boundary
10

```

```

11     left_fit = []
12     right_fit = []
13
14     for seg in segments:
15         for x1, y1, x2, y2 in seg:
16             if x1 == x2:
17                 continue
18
19             slope = (y2 - y1) / (x2 - x1)
20             intercept = y1 - slope * x1
21
22             if slope < 0:
23                 if x1 < left_boundary and x2 < left_boundary:
24                     left_fit.append((slope, intercept))
25             else:
26                 if x1 > right_boundary and x2 > right_boundary:
27                     right_fit.append((slope, intercept))
28
29     if len(left_fit) > 0:
30         s, itc = np.average(left_fit, axis=0)
31         lane_lines.append(make_points(frame, s, itc))
32
33     if len(right_fit) > 0:
34         s, itc = np.average(right_fit, axis=0)
35         lane_lines.append(make_points(frame, s, itc))
36
37     return lane_lines

```

Listing 5.7: Lane classification and averaging

Steering Angle Computation The steering angle is computed by the function `get_steering_angle(frame, lane_lines)`. The function is divided into three cases, depending on the set of detected lines:

1. **Case 1:** Two lines are detected. In this case, the lowest point of each line, which corresponds to the point closest to the vehicle, is considered. The lane center is then computed as

$$x_{\text{lane center}} = \frac{x_{\text{left}} + x_{\text{right}}}{2}.$$

Since the lane center does not exactly coincide with the camera center, is applied an offset, computed during the calibration. After calculating these parameters, the lateral displacement is determined:

$$x_{\text{offset}} = x_{\text{lane center}} - x_{\text{mid}}.$$

2. **Case 2:** Only one line is detected. In this case, only the direction of the line is used to infer the steering direction.
3. **Case 3:** No line is detected. In this case, there is no lateral deviation, and the vehicle maintains the previous heading.

After this step, the angle relative to the longitudinal direction is calculated based on the offset.

The output of this process is thus the steering angle, converted to degrees, representing how much the ROSMASTER is rotated relative to the lane center.

```

1 def get_steering_angle(frame, lane_lines):
2     h, w = frame.shape[:2]
3
4     if len(lane_lines) == 2:
5         _, _, lx2, _ = lane_lines[0]
6         _, _, rx2, _ = lane_lines[1]
7         mid = w / 2 + CONFIG["PIXEL_TRIM"]
8         x_offset = (lx2 + rx2) / 2.0 - mid
9         y_offset = h / 2.0
10
11    elif len(lane_lines) == 1:
12        x1, _, x2, _ = lane_lines[0]
13        x_offset = x2 - x1
14        y_offset = h / 2.0
15
16    else:
17        x_offset = 0.0
18        y_offset = h / 2.0
19
20    angle_to_mid_rad = math.atan2(x_offset, y_offset)
21    deviation_deg = angle_to_mid_rad * 180.0 / math.pi
22    return float(deviation_deg)

```

Listing 5.8: Function to compute the steering angle based on detected lane lines

Angle Smoothing When the ROSMASTER drives based on the detected lane lines, it cannot use the instantaneous steering angle value, as this value can vary rapidly from one frame to another due to noise, image interference, or segmentation imperfections. If this value were used directly, the ROSMASTER could steer abruptly and unstably, causing oscillations. Therefore, it was necessary to find a proper compromise between stability and responsiveness to sudden changes in the road, such as sharp curves. For this reason, a smoothing filter is applied, using an exponential moving average, where the angle is smoothed with a damping coefficient set during calibration and the value from the previous frame before being sent to the controller.

```
1 angle_s =(1 - ANGLE_SMOOTH) * angle_dev+ANGLE_SMOOTH * angle_s
```

Listing 5.9: Steering angle smoothing with exponential moving average

This implies that the final angle used by the ROSMASTER does not follow each variation instantaneously, but is a weighted combination of the new and previous values. As the smoothing coefficient α increases, the angle changes more slowly, resulting in greater stability and fewer oscillations.

Velocity Control After the smoothed steering angle is computed, the ROSMASTER is given an adaptive behavior: as the steering angle increases, corresponding to a sharper curve, the ROSMASTER reduces its forward velocity, while on straight segments, when the angle approaches zero, the forward velocity is increased.

A nominal base velocity, VX_BASE , is chosen. When the steering angle is small, a speed boost is applied to improve efficiency on straight segments, while as the angle increases, i.e., when the ROSMASTER is turning more sharply, the velocity is progressively reduced to improve stability. This behavior can be mathematically expressed as:

$$v_x = VX_BASE \left(1 + STRAIGHT_GAIN \cdot \frac{STRAIGHT_ANGLE_THRESH - |\theta_s|}{STRAIGHT_ANGLE_THRESH} \right),$$

where $|\theta_s|$ is the absolute value of the smoothed steering angle and `straight_angle_thresh` is the threshold below which the segment is considered straight.

To prevent the robot from moving too fast or too slow, the final velocity is clamped between a minimum and maximum velocity, VX_MIN and VX_MAX .

In this way, the ROSMASTER maintains an appropriate velocity: higher on straight segments and lower in the curves, improving both driving efficiency and stability.

Correction A small correction mechanism, called a *nudge*, is applied to the steering when only a single lane line is detected. This is necessary because the ROSMASTER could otherwise get too close to the line or even drive partially over it.

The worst-case scenario occurs when the robot is perfectly aligned on top of the line. In this situation, the robot, as currently programmed, would be unable to relocate the lane it has lost and could remain straddling the line. To prevent this, a corrective nudge is applied whenever the robot approaches too close to the detected line.

First, the position of the detected line relative to the center of the camera is computed:

```

1 mid = CONFIG["BIRDEYE_SIZE"][0] / 2.0 + CONFIG["PIXEL_TRIM"]
2 if x_at > (mid + CONFIG["NUDGE_RIGHT_THRESHOLD_PX"]):
3     nudge = +CONFIG["NUDGE_SINGLE_RIGHT_DEG"]
4 elif x_at < (mid - CONFIG["NUDGE_LEFT_THRESHOLD_PX"]):
5     nudge = -CONFIG["NUDGE_SINGLE_LEFT_DEG"]

```

Listing 5.10: Compute line position relative to center

This determines whether the line is too far to the right or to the left of the center. Depending on this, a small additional steering angle (*nudge*) is applied in the opposite direction to push the robot back toward the lane center:

```

1 if nudge != 0.0:
2     angle_s = float(clamp(angle_s + nudge, -CONFIG["ANGLE_CLIP"],
3                           CONFIG["ANGLE_CLIP"]))

```

Listing 5.11: Apply nudge to steering angle

This correction ensures that the robot does not drift too close to the line and maintains a safer trajectory when only one lane marking is visible.

5.2.3 MQTT Communication

The final computation steering angle and velocity commands are transmitted to the ROS2 control node using the MQTT protocol. Only if the system is armed the code start to send the information about the angle

```

1 send_command({
2     "command": "trajectory_angle",
3     "angle": angle_s,
4     "vx_mps": vx
5 })

```

Listing 5.12: MQTT command publishing

MQTT provides lightweight, asynchronous communication with low overhead, making it suitable for real-time control applications on embedded platforms.

5.2.4 Interface

User Interface Overview The user interface has been designed to be as clear and clean as possible. The user can perform two main tasks:

The first task is to calibrate the H, S, and V values in order to easily adapt the color recognition according to the environment. This action is performed using trackbars, which allow the user to either move the sliders or directly click on the values and enter the desired number, enabling adjustments to identify one color rather than another.

The second task is to select the trapezoid for the Bird's Eye View, based on the environment, how far the user wants to "see ahead", the amount of lateral interference, or to adapt to small or large displacements of the camera.

When the program is launched, a main window and the two trackbar panels are displayed. The main window shows, on the left, the camera view with the detected lane lines and the steering angle indication, and a column on the right.

The right column, instead, displays the region of interest with the trapezoid drawn, allowing the user to move the trackbars and immediately see the correspondence between the visualized region and the selected trapezoid. Additionally, there is a display showing the HSV values, the steering angle, and system status information.

At the bottom, a legend for the keyboard commands is provided:

1. Pressing the **s** key start/stop begins or stops the transmission of commands to the code running in the Follow_road docker.
2. The **r** key loads the Bird's Eye View trapezoid information into the code.
3. The **w** key saves any modified information so that when the code is closed and relaunched, no further adjustments are needed.
4. The **q** key closes the application.

In the figure 5.5 below, the user interface of the follow-line system is shown.

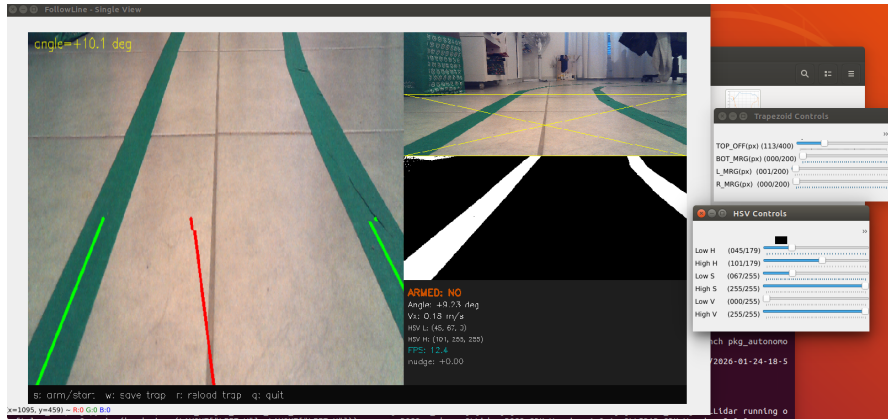


Figure 5.5: Interface

On the left side, the main visualization shows the detected lane lines in green, representing the trajectory that the robot is required to follow, while the red line indicates the heading direction used to compute the steering angle.

On the top-right panel, the limits of the trapezoidal region used for the bird's-eye view transformation are highlighted in yellow. This region defines the area of

interest for the perspective transformation and can be adjusted in real time. Immediately below, the binary mask obtained from the HSV-based color segmentation is displayed.

The simultaneous visualization of multiple processing stages allows for rapid debugging of motion-related issues. In particular, it enables the identification of problems caused either by incorrect line detection or by external visual interferences.

Below the trapezoid control panel, that allows to fix the trapezoid in order to have the desired shape, the HSV trackbars are provided, allowing real-time tuning of the color segmentation thresholds.

Finally, the bottom-right panel displays key system information, including the robot armed state, the commanded linear velocity, the steering angle, the HSV threshold values, the nudge signal, and the processing frame rate (FPS). The FPS value represents the number of frames processed per second and was particularly useful during the development phase to identify algorithmic configurations that introduced significant delays in the control commands sent to the robot.

5.3 Follow Road

The code executed inside the Docker container is responsible for reading the steering angle (`angle_deg`) received from the previously described module. This angle is then processed by the control logic and transformed into a commanded angular velocity around the vertical axis (wz), expressed in radians per second. The resulting wz value is then sent as a motion command to the four drive motors of the robot, ensuring that the vehicle follows the desired trajectory.

In addition to the motion control, the code also manages two auxiliary tasks. First, it controls the status LEDs of the robot, providing a visual indication of the current operating state, such as *idle*, *active*, or *emergency stop*. Second, it monitors the environment using the LIDAR sensor, detecting potential obstacles and triggering a safety response if any object is detected within a predefined safety distance.

The function `compute_angular_from_angle` implements a non-linear mapping between the steering angle (expressed in degrees) and the commanded angular velocity around the vertical axis, wz , expressed in radians per second. This mapping is designed to ensure smooth behavior for small deviations while providing a stronger response for larger curves.

In the following figure 5.6, the variation of the control gain as a function of the steering angle is shown.

Initially, the input angle is saturated to a maximum absolute value (`max_angle_deg`), chosen as 45° . This prevents the generation of excessively large angular commands,

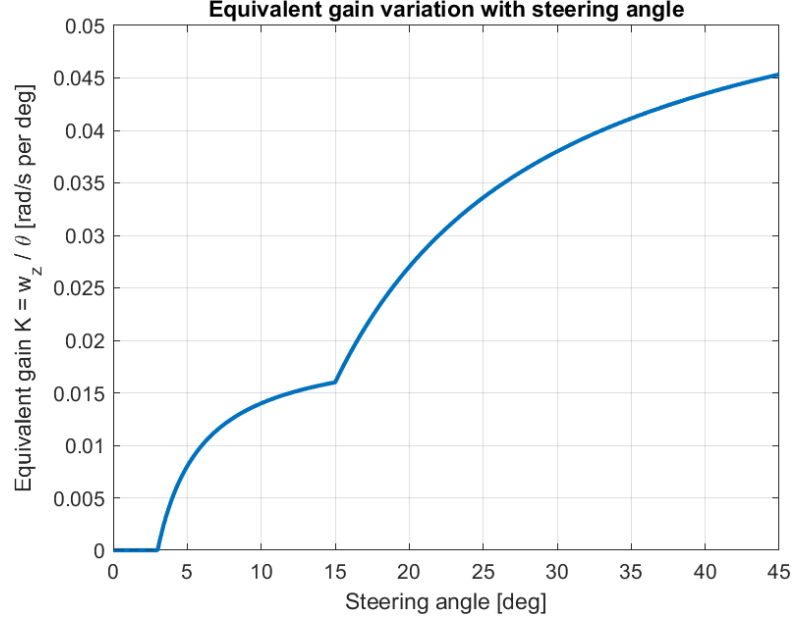


Figure 5.6: Gain variation

which could compromise the robot’s stability or result from erroneous angle computations. The sign of the input angle is stored, while the subsequent control computations consider only the absolute value.

A deadband region is applied to avoid unnecessary oscillations. Specifically, if the absolute angle satisfies:

$$A \leq \text{angle_deadband_deg},$$

the commanded angular velocity is set to zero $w_z = 0$.

This prevents the robot from reacting to small fluctuations caused by sensor noise, segmentation errors, or minor perception inaccuracies.

When the steering angle exceeds the deadband, an effective angle is computed:

$$A_{\text{eff}} = A - \text{angle_deadband_deg},$$

and a piecewise proportional control law is applied. A knee threshold, `angle_knee`, defines the transition between two gain regions. For small to moderate steering angles ($A_{\text{eff}} \leq \text{angle_knee_deg} - \text{angle_deadband_deg}$), the commanded angular velocity is computed using a low proportional gain:

$$w_z = \text{angular_gain_lo} \cdot A_{\text{eff}}.$$

For larger steering angles ($A_{\text{eff}} > \text{angle_knee_deg} - \text{angle_deadband_deg}$), the

angular velocity increases more rapidly using a higher proportional gain:

$$wz = \text{angular_gain_lo} \cdot \text{angle_knee_rel} + \text{angular_gain_hi} \cdot (A_{\text{eff}} - \text{angle_knee_rel})$$

The original sign of the input angle (s) is then reapplied to wz :

$$WZ \leftarrow WZ \cdot s.$$

The output is the final angular velocity command, wz , as a floating-point number. In summary, this function implements a **non-linear proportional controller with deadband and gain scheduling**, designed to improve both smoothness and responsiveness of the robot in a lane-following task.

```

1 def compute_angular_from_angle(self, angle_deg: float) -> float:
2
3     # Clip input angle
4     max_angle = 45.0
5     a = float(max(min(angle_deg, max_angle), -max_angle))
6     s = 1.0 if a >= 0 else -1.0
7     A = abs(a)
8
9     # Deadband
10    db = self.angle_deadband_deg
11    if A <= db:
12        wz = 0.0
13    else:
14        # Effective angle
15        Aeff = A - db
16        knee = max(1e-3, self.angle_knee_deg - db)
17
18        # Non-linear proportional control
19        if Aeff <= knee:
20            # Low-gain
21            wz = self.angular_gain_lo * Aeff
22        else:
23            # High-gain
24            wz = self.angular_gain_lo * knee + self.
angular_gain_hi * (Aeff - knee)
25
26
27        wz *= s
28
29
30    if self.INVERT_SIGN:
31        wz = -wz
32
33    return float(wz)

```

Listing 5.13: Non-linear mapping from steering angle to angular velocity (wz) with deadband and piecewise proportional gains.

LiDAR The theLidar function processes the data recorded by the sensor after each scan received from the scan subscriber. For every scan, the measured distances are analyzed and either considered or discarded based on their relevance. In this work, an obstacle is of interest only if it is located in front of the robot. Therefore, a region of interest of $\pm 60^\circ$ has been defined, so that only obstacles within this frontal area, which may obstruct the robot's path, are taken into account.

The figure 5.7 below illustrates the region of interest considered for obstacle detection. If an obstacle is located within this area, the robot will recognize it as an obstacle and stop instead obstacles outside this region are ignored.

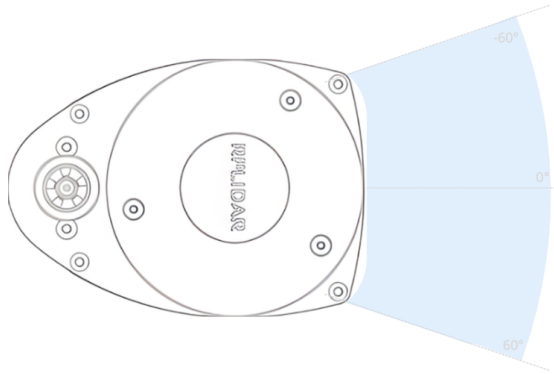


Figure 5.7: LiDAR Range

LED It was decided to activate the robot LEDs with different colors according to the system state, for two main reasons:

- **Visual feedback:** similarly to the graphical interface, the goal was to make the robot as intuitive as possible to use. The LED activation is used as the main indicator of the robot operational state, allowing real-time monitoring of the system behavior without the need to observe a screen.
- **Support during experimental validation:** during software development, the LEDs proved to be very useful for interpreting the robot behavior and facilitated the identification of errors or malfunctions during testing.

The adopted color code is as follows: **green** indicates the armed state, when the algorithm correctly detects the lane and receives the `steering_angle` command, **blue** when the robot detects the lane markings but is not armed and therefore does not initiate motion, **red** when the robot is armed and detects an obstacle through the LiDAR sensor, and finally, **purple** when the robot is not armed but detects an obstacle.

In the figure 5.8 below, LEDs of different colors are shown according to the robot's state.

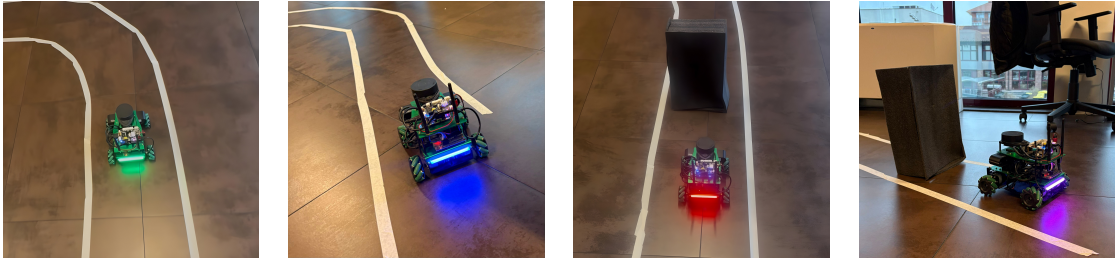


Figure 5.8: LEDs colors

Chapter 6

Tests and results

In this chapter, the final performance results of the robot are presented and discussed. The objective is to evaluate the overall behavior of the autonomous system during the experimental tests, focusing on robustness, adaptability to environmental changes, and path-following capabilities.

As previously described, the control architecture is based on a ROS 2 framework running inside a Docker container. For this reason, the first required steps consist of starting the container and accessing the robot workspace, which contains all the ROSEMASTER X3 packages, custom-developed nodes, and required libraries. These operations are performed directly from the terminal using the following commands:

```
./run_docker.sh  
cd ~/yahboomcar_ros_ws/yahboomcar_ws
```

Once inside the workspace, the autonomous control node can be launched using the following command:

```
ros2 launch pkg_autonomous autonomous_follow_launch.py
```

Simultaneously, a second script is executed outside the Docker container to handle the lane-keeping algorithm:

```
python3 lane_keeping.py
```

After these steps, the robot is fully initialized and ready to start the autonomous navigation task.

6.1 Adaptability of the Robot

One of the main requirements of this work, especially in comparison with previous thesis implementations, was to ensure robustness against variations in lighting

conditions and environmental configurations. This requirement has been fully satisfied.

As shown in the images below, the robot is capable of generating accurate binary masks by adjusting only the HSV threshold values, even under significantly different operating conditions. Specifically, two representative scenarios are considered: a green line on a white floor and a white line on a brown floor. In both cases, the segmentation process successfully isolates the lane markings, producing clean and reliable masks.

It is important to note that in the scenario with the brown floor, large windows are present on the sides of the environment. These windows allow natural light to enter and reflect on the floor surface, introducing non-uniform illumination and reflections. Despite these challenging conditions, the vision-based algorithm remains stable and does not suffer from false detections.

Furthermore, the robot was tested in an environment containing several white objects close to the path. By properly adjusting the region of interest through the trapezoidal mask, it is possible to exclude these disturbing elements from the image processing pipeline. As a result, such objects do not interfere with line detection and, consequently, with the computation of the steering angle.

In the image on the right, an object with a color similar to the robot (a green plastic bag) is visible within the camera field of view. Even in this case, the object can be effectively ignored by modifying the trapezoidal region of interest, confirming the flexibility and robustness of the proposed approach.

These results clearly demonstrate that the adaptability requirement of the robot has been successfully fulfilled.

6.2 Path Following Performance

The path-following performance of the robot is evaluated by analyzing the steering angle behavior in different driving scenarios. The following plots show the steering angle as a function of time for three representative cases: straight motion, a U-shaped right turn, and a slalom section.

The Steering Angle with respect to the Time graph, shown in figure 6.1, provides a clear representation of the robot behavior along the entire trajectory, which can be divided into three main phases.

Path (0-15s): During the initial straight segment, the steering angle remains close to zero, with small oscillations limited to approximately $\pm 5^\circ$. This behavior indicates stable motion along the lane, with minor corrective actions required to compensate for noise and small disturbances.

Right U-Turn (15-30s): In the U-shaped curve, a rapid increase in the steering angle is observed, reaching the maximum values allowed by the clamping mechanism ($\pm 25^\circ$). Frequent oscillations between positive and negative values appear in this phase, highlighting the effort of the controller to follow a tight curvature under actuator saturation constraints. The presence of sharp peaks is a direct consequence of steering angle saturation.

Slalom Section (30-45): In the final slalom segment, the steering angle alternates rapidly between positive and negative values, exhibiting a characteristic oscillatory pattern. This behavior is expected when navigating through closely spaced obstacles that require continuous and fast steering adjustments. Once again, peaks reaching the clamping limits confirm saturation of the steering command.

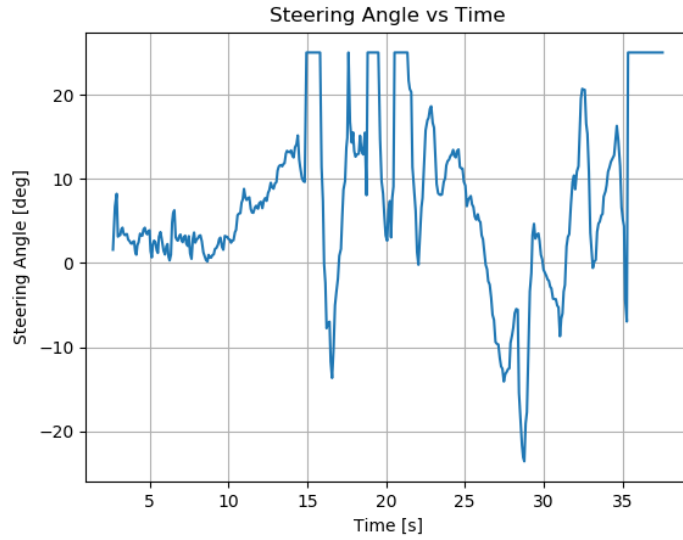


Figure 6.1: Full path behaviour

Overall, the steering angle profile highlights three distinct behaviors: stability during straight motion, saturation and oscillations during the U-turn, and reactivity during the slalom section, demonstrating the effectiveness of the control strategy in different driving conditions.

6.3 Robot limitation

It can be observed that the robot does not exhibit a perfectly linear behavior. This is mainly due to the fact that the motor command is not generated from a continuous and smooth reference trajectory. Instead, the steering input depends directly on the

visual information acquired by the onboard camera. This implementation choice is driven by two primary factors:

Blind Area: The camera configuration leaves approximately 20 cm of unobserved space in front of the robot. This limited field of view significantly affects trajectory stability, especially during curved segments. The absence of a visible reference line in this region makes it difficult to maintain a stable and smooth path, particularly in tight curves. If the curvature is too sharp, the robot would not be able to negotiate it without a control strategy of this type. In figure 6.2, we can see a representation of the zone where the robot cannot see anything.

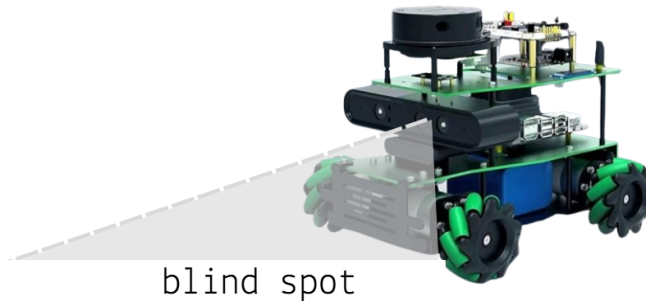


Figure 6.2: Blind Spots of the robot

During the design phase, a trade-off had to be considered. The system could have been optimized for smoother motion, but only for predefined paths characterized by wide curves and gradual direction changes. Alternatively, a slightly less smooth behavior was accepted in order to guaranty adaptability and robustness across a wider range of trajectories, including sharp curves and sudden direction variations.

Navigation Condition: Several lane detection algorithms are based on the column histogram accumulator approach. A cumulative histogram (or cumulative column accumulation) is a data analysis technique that represents the total count (or percentage) of data points that fall within a specific bin together with all lower valued bins. In the context of lane detection, this method is typically used to identify regions with a high concentration of lane-related features along the image columns.

However, in this specific case, such techniques could not be adopted. The robot is a prototype specially designed for indoor operation. When applying histogram-based lane detection methods in indoor environments, the probability of misinterpreting floor tile edges or perspective lines as lane markings becomes excessively high. An operation that could have been achieved in only two processing

stages through a histogram accumulator approach had instead to be implemented using the complete image processing pipeline described above. Consequently, the algorithm becomes computationally more demanding and introduces additional latency into the system.

Chapter 7

Conclusion and future works

The rapid evolution of Advanced Driver Assistance Systems (ADAS) represents one of the most significant technological transformations in the automotive sector. Throughout this work, existing ADAS technologies have been analyzed, highlighting both their strengths and their current limitations.

Regarding the practical part of the project, the objective was to investigate in detail how a Lane Keeping system should operate in order to contribute to autonomous driving. The development was carried out on a prototype platform, the ROSmaster X3, and therefore the results are not directly applicable to a real vehicle. Nevertheless, the performance achieved can be considered a very positive outcome given the hardware constraints of the platform.

Although the system does not yet provide perfect trajectory tracking, the results demonstrate that the design and implementation of an autonomous lane keeping controller is feasible. The project allowed the identification of the main requirements and specifications that such a system must satisfy, including perception reliability, robustness to environmental variations, and stable control performance. Through the integration of data from a camera and LiDAR sensor, a stable and robust behavior was achieved. The system proved to be not excessively sensitive to lighting variations and maintained a relatively low tracking error. However, the generated trajectory is not yet fully homogeneous and smooth, as it is still partially based on instantaneous error correction rather than predictive path estimation.

Several future improvements can be considered. First, repositioning the camera to obtain a wider and longer field of view without blind zones would allow the robot to anticipate curves more effectively. This would enable trajectory estimation based on path prediction rather than reactive error compensation, resulting in smoother and more uniform motion. Although the system is sufficiently stable under lighting changes, it is not completely insensitive to them. This limitation could be mitigated by integrating an infrared-based sensor, reducing dependency on visible light conditions.

Beyond lane keeping, multiple additional ADAS functionalities could be implemented on the platform. Future developments may include the integration of artificial intelligence algorithms for traffic sign recognition and the addition of a radar sensor to enable more advanced maneuvering capabilities. These extensions would further transform the prototype into a more comprehensive autonomous driving research platform.

In conclusion, this work demonstrates that even within the constraints of a prototype system, it is possible to design and implement a functional and robust Lane Keeping system. The results confirm both the complexity and the potential of ADAS technologies, emphasizing how perception, sensor fusion, and control strategies must be carefully integrated to achieve reliable autonomous behavior.

Bibliography

- [1] Jorgen Frohm. *Levels of Automation in production systems*. Chalmers University of Technology Goteborg, 2008 (cit. on p. 6).
- [2] sheridan. «Rumination on Automation, 1998». In: *IFAC Proceedings Volumes* 31.26 (1998), pp. 19–27 (cit. on p. 6).
- [3] Bassim Abdulbaqi Jumaa, Anwaar Mousa Abdulhassan, and Ammar Mousa Abdulhassan. «Advanced driver assistance system: A review of systems and technologies». In: *Int. J. Adv. Res. Comput. Eng. Technol.(IJARCET)* 8.6 (2019), pp. 231–234 (cit. on p. 8).
- [4] Joshue Perez, David Gonzalez, and Vicente Milanes. «Vehicle control in ADAS applications: State of the art». In: *Intelligent transport systems: technologies and applications* (2015), p. 206 (cit. on pp. 9, 10).
- [5] Angelos Amditis et al. «A situation-adaptive lane-keeping support system: Overview of the safelane approach». In: *IEEE Transactions on Intelligent Transportation Systems* 11.3 (2010), pp. 617–629 (cit. on p. 11).
- [6] Ricardo Roriz, Jorge Cabral, and Tiago Gomes. «Automotive LiDAR technology: A survey». In: *IEEE Transactions on Intelligent Transportation Systems* 23.7 (2021), pp. 6282–6297 (cit. on p. 13).
- [7] NVIDIA Corporation. *NVIDIA Jetson Nano System-on-Module Datasheet*. v1.1. DA-09366-001_v1.1. NVIDIA Corporation. May 2022. URL: <https://developer.nvidia.com/embedded/downloads> (cit. on p. 19).
- [8] Yahboom Technology. *ROS Robot Expansion Board: STM32F103RCT6 Driver and Control Board*. <https://github.com/YahboomTechnology/ROS-robot-expansion-board>. Version 3.0, Support for Raspberry Pi 5 and Jetson Nano. 2026 (cit. on p. 19).
- [9] Shanghai Slamtec Co., Ltd. *RPLIDAR A1 Low Cost 360 Degree Laser Range Scanner Introduction and Datasheet*. rev. 2.2. Model: A1M8. Shanghai Slamtec Co., Ltd. Feb. 2019. URL: <https://www.slamtec.com> (cit. on p. 20).

- [10] Orbbec Inc. *Orbbec Astra+ Depth Camera Datasheet*. V3.0. Document No.: D 20211129. Orbbec Inc. Shenzhen, China, Mar. 2022. URL: <https://www.orbbec.com> (cit. on p. 21).
- [11] Mehdi Mafi, Harold Martin, Mercedes Cabrerizo, Jean Andrian, Armando Barreto, and Malek Adjouadi. «A comprehensive survey on impulse and Gaussian denoising filters for digital images». In: *Signal Processing* 157 (2019), pp. 236–260 (cit. on p. 26).

Appendix A

Lane_Keeping

```
1
2 import cv2
3 import numpy as np
4 import math
5 import time
6 import json
7 import os
8 import sys
9 import threading
10 import paho.mqtt.client as mqtt
11
12 # ----- MQTT setup (same topic as your app) -----
13 broker_ip = "localhost"
14 broker_port = 1883
15 topic = "robot/control"
16
17 client = mqtt.Client()
18
19
20 def on_connect(client, userdata, flags, rc, properties=None)
21     :
22     if rc == 0:
23         print("Connected to MQTT Broker!")
24     else:
25         print("MQTT connect failed rc=", rc)
26
27 client.on_connect = on_connect
28 try:
29     client.connect(broker_ip, broker_port, 60)
```

```

30     client.loop_start()
31 except Exception as e:
32     print("Warning: cannot connect to MQTT broker:", e)
33
34
35 def send_command(message, angle=None):
36     try:
37         if isinstance(message, str) and message.startswith('{'):
38             client.publish(topic, message)
39         else:
40             command = {"command": message}
41             if angle is not None:
42                 command["angle"] = angle
43             client.publish(topic, json.dumps(command))
44     except Exception as e:
45         print("send_command error:", e)
46
47
48 def send_activate():
49     try:
50         client.publish(topic, json.dumps({"command": "
activate"}))
51     except Exception as e:
52         print("send_activate error:", e)
53
54
55 def send_stop():
56     try:
57         client.publish(topic, json.dumps({"command": "stop"
}))
58     except Exception as e:
59         print("send_stop error:", e)
60
61
62 def truncate(f, n):
63     s = '{}'.format(f)
64     if 'e' in s or 'E' in s:
65         return '{0:.{1}f}'.format(f, n)
66     i, p, d = s.partition('.')
67     return '.'.join([i, (d + '0' * n)[:n]])
68
69
70 # ----- Config (adapted from your Follow_Line_Core)
    -----

```

```

71 CONFIG = {
72     "CAM_INDEX": 0,
73     "CAM_W": 1280,
74     "CAM_H": 720,
75     "BIRDEYE_SIZE": (640, 480),
76     "EDGE_LOW": 50,
77     "EDGE_HIGH": 100,
78     "HOUGH_RHO": 1,
79     "HOUGH_THETA": np.pi / 180.0,
80     "HOUGH_THRESH": 10,
81     "HOUGH_MIN_LEN": 5,
82     "HOUGH_MAX_GAP": 150,
83     "ANGLE_SMOOTH": 0.2,
84     "ANGLE_CLIP": 30.0,
85     "PIXEL_TRIM": -5,
86     "VX_BASE": 0.25,
87     "VX_MIN": 0.08,
88     "VX_MAX": 0.30,
89     "STRAIGHT_GAIN": 0.40,
90     "STRAIGHT_ANGLE_THRESH": 5.0,
91     "NUDGE_SINGLE_RIGHT_DEG": 1.8,
92     "NUDGE_RIGHT_THRESHOLD_PX": 10,
93     "NUDGE_SINGLE_LEFT_DEG": 1.8,
94     "NUDGE_LEFT_THRESHOLD_PX": 10,
95     "NUDGE_Y_REF": 0.78,
96     "NUDGE_ENABLE_MAX_ABS": 7.0,
97     "PUB_HZ": 30.0,
98     "SHOW_WINDOWS": True,
99     "DRAW_TRAP": True,
100     # default HSV (can be passed or adjusted)
101     "HSV_LOWER": (45, 67, 0),
102     "HSV_UPPER": (101, 255, 255),
103 }
104
105 # --- layout (personalizza qui) ---
106 LAYOUT = {
107     # Left big panel (heading / BEV)
108     "LEFT_W": 960,
109     "LEFT_H": 720,
110     # Right column width and individual panel heights
111     "RIGHT_W": 320,
112     "RIGHT_TOP_H": 240,      # trapezoid view
113     "RIGHT_MID_H": 240,      # mask/edges view (increase this
114     # to enlarge mask)
115     "RIGHT_BOTTOM_H": 240,   # stats panel

```

```

115 }
116 # --- end layout ---
117
118
119 # ----- helper utilities (copied / adapted) -----
120 def clamp(x, lo, hi):
121     return max(lo, min(hi, x))
122
123
124 def estimate_horizon_y(img_w, img_h, fov_h_deg, pitch_deg):
125     fov_h = np.deg2rad(fov_h_deg)
126     fov_v = 2.0 * np.arctan(np.tan(fov_h / 2.0) * (img_h /
127         img_w))
128     fy = (img_h / 2.0) / np.tan(fov_v / 2.0)
129     cy = img_h / 2.0
130     return int(np.clip(int(round(cy - fy * math.tan(math.
131         radians(pitch_deg)))), 0, img_h - 1))
132
133
134 def compute_ipm_transform(src_pts, dst_size):
135     w_dst, h_dst = dst_size
136     dst_pts = np.array([[0, 0], [w_dst - 1, 0], [0, h_dst -
137         1], [w_dst - 1, h_dst - 1]], np.float32)
138     src_pts = np.array(src_pts, np.float32)
139     M = cv2.getPerspectiveTransform(src_pts, dst_pts)
140     return M
141
142
143 def homography_rotate_about_center(w, h, deg):
144     th = np.deg2rad(deg)
145     c, s = np.cos(th), np.sin(th)
146     cx, cy = w / 2.0, h / 2.0
147     T1 = np.array([[1, 0, -cx], [0, 1, -cy], [0, 0, 1]], np.
148         float32)
149     R = np.array([[c, -s, 0], [s, c, 0], [0, 0, 1]], np.
150         float32)
151     T2 = np.array([[1, 0, cx], [0, 1, cy], [0, 0, 1]], np.
152         float32)
153     return T2 @ R @ T1
154
155
156 def build_trap_from_rows(img_w, img_h, horizon_y,
157     top_offset_px, bottom_margin_px, left_margin_px,
158     right_margin_px):

```

```

151     y_top = int(np.clip(horizon_y + top_offset_px, 0, img_h
152                        - 2))
153     y_bot = int(np.clip(img_h - 1 - bottom_margin_px, y_top
154                        + 1, img_h - 1))
155     xL = int(np.clip(left_margin_px, 0, img_w - 2))
156     xR = int(np.clip(img_w - 1 - right_margin_px, xL + 1,
157                        img_w - 1))
158     return np.array([[xL, y_top], [xR, y_top], [xL, y_bot],
159                    [xR, y_bot]], np.float32)
160
161 def detect_edges_bgr(frame_bgr, lower, upper):
162     hsv = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2HSV)
163     mask = cv2.inRange(hsv, np.array(lower, np.uint8), np.
164                        array(upper, np.uint8))
165     edges = cv2.Canny(mask, CONFIG["EDGE_LOW"], CONFIG["
166                        EDGE_HIGH"])
167     return edges, mask
168
169 def roi_lower_half(edges):
170     h, w = edges.shape
171     mask = np.zeros_like(edges)
172     polygon = np.array([[0, h), (0, h // 2), (w, h // 2), (
173                        w, h)], np.int32)
174     cv2.fillPoly(mask, polygon, 255)
175     return cv2.bitwise_and(edges, mask)
176
177 def detect_line_segments(cropped_edges):
178     return cv2.HoughLinesP(
179         cropped_edges,
180         CONFIG["HOUGH_RHO"],
181         CONFIG["HOUGH_THETA"],
182         CONFIG["HOUGH_THRESH"],
183         np.array([]),
184         minLineLength=CONFIG["HOUGH_MIN_LEN"],
185         maxLineGap=CONFIG["HOUGH_MAX_GAP"],
186     )
187
188 def make_points(frame, slope, intercept):
189     h, w = frame.shape[:2]
190     y1 = h
191     y2 = h // 2

```

```

189     if abs(slope) < 1e-6:
190         slope = 1e-3
191     x1 = int((y1 - intercept) / slope)
192     x2 = int((y2 - intercept) / slope)
193     return [x1, y1, x2, y2]
194
195
196 def average_slope_intercept(frame, segments):
197     lane_lines = []
198     if segments is None:
199         return lane_lines
200     h, w = frame.shape[:2]
201     boundary = 1 / 3
202     left_boundary = w * (1 - boundary)
203     right_boundary = w * boundary
204     left_fit = []
205     right_fit = []
206     for seg in segments:
207         for x1, y1, x2, y2 in seg:
208             if x1 == x2:
209                 continue
210             slope = (y2 - y1) / (x2 - x1)
211             intercept = y1 - slope * x1
212             if slope < 0:
213                 if x1 < left_boundary and x2 < left_boundary
:
214                 left_fit.append((slope, intercept))
215             else:
216                 if x1 > right_boundary and x2 >
right_boundary:
217                 right_fit.append((slope, intercept))
218     if len(left_fit) > 0:
219         s, itc = np.average(left_fit, axis=0)
220         lane_lines.append(make_points(frame, s, itc))
221     if len(right_fit) > 0:
222         s, itc = np.average(right_fit, axis=0)
223         lane_lines.append(make_points(frame, s, itc))
224     return lane_lines
225
226
227 def get_steering_angle(frame, lane_lines):
228     h, w = frame.shape[:2]
229     if len(lane_lines) == 2:
230         _, _, lx2, _ = lane_lines[0]
231         _, _, rx2, _ = lane_lines[1]

```

```

232         mid = w / 2 + CONFIG["PIXEL_TRIM"]
233         x_offset = (lx2 + rx2) / 2.0 - mid
234         y_offset = h / 2.0
235     elif len(lane_lines) == 1:
236         x1, _, x2, _ = lane_lines[0]
237         x_offset = x2 - x1
238         y_offset = h / 2.0
239     else:
240         x_offset = 0.0
241         y_offset = h / 2.0
242     angle_to_mid_rad = math.atan2(x_offset, y_offset)
243     deviation_deg = angle_to_mid_rad * 180.0 / math.pi
244     return float(deviation_deg)
245
246
247 def overlay_heading(frame, lane_lines, angle_dev,
248                    nudge_applied=0.0):
249     img = frame.copy()
250     # draw lines
251     if lane_lines:
252         for x1, y1, x2, y2 in lane_lines:
253             cv2.line(img, (x1, y1), (x2, y2), (0, 255, 0),
254                     3)
255     # heading
256     h, w = frame.shape[:2]
257     x1, y1 = int(w / 2), h
258     steer_rad = math.radians(angle_dev)
259     x2 = int(x1 - (h / 2) * math.tan(steer_rad))
260     y2 = int(h / 2)
261     cv2.line(img, (x1, y1), (x2, y2), (0, 0, 255), 3)
262     cv2.putText(img, f"angle={angle_dev:.1f} deg", (10, 20),
263                cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 255), 1)
264     if abs(nudge_applied) > 1e-6:
265         cv2.putText(img, "NUDGE L", (10, 42), cv2.
266            FONT_HERSHEY_SIMPLEX, 0.6, (0, 165, 255), 2)
267     return img
268
269
270 def single_line_x_at_y(line_pts, y_ref):
271     x1, y1, x2, y2 = line_pts
272     if x1 == x2:
273         return float(x1)
274     m = (y2 - y1) / (x2 - x1 + 1e-9)
275     q = y1 - m * x1
276     return float((y_ref - q) / (m + 1e-9))

```

```

273
274
275 # ----- Frame grabber thread -----
276 class FrameGrabber:
277     def __init__(self, device=0, width=1280, height=720, fps
    =None):
278         self.cap = cv2.VideoCapture(device)
279         if not self.cap.isOpened():
280             raise RuntimeError("Camera not available")
281         try:
282             self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
283             self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)
284             self.cap.set(cv2.CAP_PROP_BUFFERSIZE, 1)
285             if fps:
286                 self.cap.set(cv2.CAP_PROP_FPS, fps)
287         except Exception:
288             pass
289         self.lock = threading.Lock()
290         self.frame = None
291         self.stopped = False
292         self.thread = threading.Thread(target=self._loop,
    daemon=True)
293         self.thread.start()
294
295     def _loop(self):
296         while not self.stopped:
297             ok, f = self.cap.read()
298             if not ok:
299                 time.sleep(0.003)
300                 continue
301             with self.lock:
302                 self.frame = f
303
304     def read(self):
305         with self.lock:
306             return None if self.frame is None else self.
    frame.copy()
307
308     def release(self):
309         self.stopped = True
310         try:
311             self.thread.join(timeout=0.5)
312         except Exception:
313             pass
314         try:

```

```

315         self.cap.release()
316     except Exception:
317         pass
318
319
320 # ----- load src_trap if present -----
321 def load_src_trap(path="src_trap.json"):
322     if os.path.exists(path):
323         try:
324             with open(path, "r") as f:
325                 pts = json.load(f)
326                 if isinstance(pts, list) and len(pts) == 4:
327                     return np.array(pts, dtype=np.float32)
328         except Exception as e:
329             print("Failed to load src_trap.json:", e)
330     return None
331
332
333 # ----- Trackbar windows creation -----
334 def create_trap_trackbars(win_name, top_off, bot_mrg, l_mrg,
335                             r_mrg):
336     cv2.namedWindow(win_name, cv2.WINDOW_NORMAL)
337     cv2.resizeWindow(win_name, 320, 120)
338     cv2.createTrackbar("TOP_OFF(px)", win_name, int(top_off),
339                        400, lambda v: None)
340     cv2.createTrackbar("BOT_MRG(px)", win_name, int(bot_mrg),
341                        200, lambda v: None)
342     cv2.createTrackbar("L_MRG(px)", win_name, int(l_mrg),
343                        200, lambda v: None)
344     cv2.createTrackbar("R_MRG(px)", win_name, int(r_mrg),
345                        200, lambda v: None)
346     img = np.zeros((120, 320, 3), dtype=np.uint8)
347     cv2.imshow(win_name, img)
348
349
350 def create_hsv_trackbars(win_name, hsv_lower, hsv_upper):
351     cv2.namedWindow(win_name, cv2.WINDOW_NORMAL)
352     cv2.resizeWindow(win_name, 320, 200)
353     low_h, low_s, low_v = hsv_lower
354     high_h, high_s, high_v = hsv_upper
355     cv2.createTrackbar("Low H", win_name, int(low_h), 179,
356                        lambda v: None)
357     cv2.createTrackbar("High H", win_name, int(high_h), 179,
358                        lambda v: None)

```

```

352     cv2.createTrackbar("Low S", win_name, int(low_s), 255,
lambda v: None)
353     cv2.createTrackbar("High S", win_name, int(high_s), 255,
lambda v: None)
354     cv2.createTrackbar("Low V", win_name, int(low_v), 255,
lambda v: None)
355     cv2.createTrackbar("High V", win_name, int(high_v), 255,
lambda v: None)
356     img = np.zeros((200, 320, 3), dtype=np.uint8)
357     cv2.imshow(win_name, img)
358
359
360 def read_hsv_trackbars(win_name):
361     low_h = cv2.getTrackbarPos("Low H", win_name)
362     high_h = cv2.getTrackbarPos("High H", win_name)
363     low_s = cv2.getTrackbarPos("Low S", win_name)
364     high_s = cv2.getTrackbarPos("High S", win_name)
365     low_v = cv2.getTrackbarPos("Low V", win_name)
366     high_v = cv2.getTrackbarPos("High V", win_name)
367     # enforce sensible ranges
368     if high_h < low_h:
369         high_h = low_h
370     if high_s < low_s:
371         high_s = low_s
372     if high_v < low_v:
373         high_v = low_v
374     return (int(low_h), int(low_s), int(low_v)), (int(high_h)
), int(high_s), int(high_v))
375
376
377 # ----- Main loop: single-window UI with trackbars
-----
378 def run_single_window_with_trackbars(hsv_lower=None,
hsv_upper=None, src_trap=None):
379     hsv_lower = hsv_lower or CONFIG["HSV_LOWER"]
380     hsv_upper = hsv_upper or CONFIG["HSV_UPPER"]
381
382     # init camera
383     cam = FrameGrabber(CONFIG["CAM_INDEX"], CONFIG["CAM_W"],
CONFIG["CAM_H"])
384     first = None
385     while first is None:
386         first = cam.read()
387         time.sleep(0.01)
388     cam_h, cam_w = first.shape[:2]

```

```

389
390     # compute horizon + default trap if needed
391     horizon_y = int(cam_h / 2) # fallback
392     try:
393         horizon_y = estimate_horizon_y(cam_w, cam_h,
fov_h_deg=70.0, pitch_deg=7.0)
394     except Exception:
395         pass
396
397     # default trapezoid params
398     top_off, bot_mrg, l_mrg, r_mrg = 30, 30, 6, 6
399
400     # if src_trap provided, derive margins
401     if isinstance(src_trap, np.ndarray) and src_trap.shape
== (4, 2):
402         xL, y_top = src_trap[0]; xR, _ = src_trap[1]; _,
y_bot = src_trap[2]
403         top_off = int(round(y_top - horizon_y))
404         bot_mrg = int(round((cam_h - 1) - y_bot))
405         l_mrg = int(round(xL))
406         r_mrg = int(round((cam_w - 1) - xR))
407
408     # state
409     angle_s = 0.0
410     armed = False
411     last_pub = 0.0
412     last_t = time.time()
413     fps = 0.0
414     fcounter = 0
415     _current_v = 0.0
416
417     window_name = "FollowLine - Single View"
418     cv2.namedWindow(window_name, cv2.WINDOW_NORMAL)
419
420     mask_window = "HSV Mask"
421     cv2.namedWindow(mask_window, cv2.WINDOW_NORMAL)
422     cv2.resizeWindow(mask_window, 640, 480)
423     # place main window at a known screen location to try to
position controls under panels
424     main_win_x, main_win_y = 100, 50
425     try:
426         cv2.resizeWindow(window_name, LAYOUT["LEFT_W"] +
LAYOUT["RIGHT_W"], max(LAYOUT["LEFT_H"], LAYOUT["
RIGHT_TOP_H"] + LAYOUT["RIGHT_MID_H"] + LAYOUT["
RIGHT_BOTTOM_H"]))

```

```

427         cv2.moveWindow(window_name, main_win_x, main_win_y)
428     except Exception:
429         pass
430
431     # create trackbar windows and try to position them under
432     # the respective small panels
433     trap_win = "Trapezoid Controls"
434     hsv_win = "HSV Controls"
435     create_trap_trackbars(trap_win, top_off, bot_mrg, l_mrg,
436                           r_mrg)
437     create_hsv_trackbars(hsv_win, hsv_lower, hsv_upper)
438
439     # try to position these windows so they appear under the
440     # right column panels:
441     right_col_x = main_win_x + LAYOUT["LEFT_W"]
442     right_col_y = main_win_y
443     try:
444         cv2.moveWindow(trap_win, right_col_x, right_col_y +
445                       LAYOUT["RIGHT_TOP_H"] + 10)
446     except Exception:
447         pass
448     try:
449         cv2.moveWindow(hsv_win, right_col_x, right_col_y +
450                       LAYOUT["RIGHT_TOP_H"] + LAYOUT["RIGHT_MID_H"] + 20)
451     except Exception:
452         pass
453
454     try:
455         while True:
456             f = cam.read()
457             if f is None:
458                 time.sleep(0.005)
459                 continue
460
461             # update timing
462             now = time.time()
463             dt = max(1e-6, now - last_t)
464             last_t = now
465             fcounter += 1
466             if fcounter % 10 == 0:
467                 fps = 1.0 / dt
468
469             # read trackbars (apply to trapezoid + hsv)
470             try:

```

```

466         top_off = cv2.getTrackbarPos("TOP_OFF(px)",
trap_win)
467         bot_mrg = cv2.getTrackbarPos("BOT_MRG(px)",
trap_win)
468         l_mrg = cv2.getTrackbarPos("L_MRG(px)",
trap_win)
469         r_mrg = cv2.getTrackbarPos("R_MRG(px)",
trap_win)
470     except Exception:
471         pass
472
473     try:
474         hsv_lower, hsv_upper = read_hsv_trackbars(
hsv_win)
475     except Exception:
476         hsv_lower = CONFIG["HSV_LOWER"]
477         hsv_upper = CONFIG["HSV_UPPER"]
478
479         # compute trap / ipm
480         trap = build_trap_from_rows(cam_w, cam_h,
horizon_y, top_off, bot_mrg, l_mrg, r_mrg)
481         ipm_M = compute_ipm_transform(trap, CONFIG["
BIRDEYE_SIZE"])
482
483         # BEV warp
484         bev = cv2.warpPerspective(f, ipm_M, CONFIG["
BIRDEYE_SIZE"], flags=cv2.INTER_LINEAR)
485
486         # mask + edges (use hsv_lower/hsv_upper from
trackbars)
487         edges, mask = detect_edges_bgr(bev, hsv_lower,
hsv_upper)
488         edges_roi = roi_lower_half(edges)
489         cv2.imshow(mask_window, mask)
490
491         # hough & lines
492         segs = detect_line_segments(edges_roi)
493         lane_lines = average_slope_intercept(bev, segs)
494
495         # angle
496         angle_dev = get_steering_angle(bev, lane_lines)
497         angle_s = (1.0 - CONFIG["ANGLE_SMOOTH"]) *
angle_dev + CONFIG["ANGLE_SMOOTH"] * angle_s
498         angle_s = float(clamp(angle_s, -CONFIG["
ANGLE_CLIP"], CONFIG["ANGLE_CLIP"]))

```

```

499         # nudge logic (single-line nudges)
500         nudge = 0.0
501         if len(lane_lines) == 1 and abs(angle_s) <=
CONFIG["NUDGE_ENABLE_MAX_ABS"]:
502             y_ref = CONFIG["BIRDEYE_SIZE"][1] * CONFIG["
NUDGE_Y_REF"]
503             x_at = single_line_x_at_y(lane_lines[0],
y_ref)
504             mid = CONFIG["BIRDEYE_SIZE"][0] / 2.0 +
CONFIG["PIXEL_TRIM"]
505             if x_at > (mid + CONFIG["
NUDGE_RIGHT_THRESHOLD_PX"]):
506                 nudge = +CONFIG["NUDGE_SINGLE_RIGHT_DEG"
]
507             elif x_at < (mid - CONFIG["
NUDGE_LEFT_THRESHOLD_PX"]):
508                 nudge = -CONFIG["NUDGE_SINGLE_LEFT_DEG"]
509                 if nudge != 0.0:
510                     angle_s = float(clamp(angle_s + nudge, -
CONFIG["ANGLE_CLIP"], CONFIG["ANGLE_CLIP"]))
511
512         # vx computation with straight boost
513         dev_abs = abs(angle_s)
514         thr = float(CONFIG.get("STRAIGHT_ANGLE_THRESH",
5.0))
515         gain = float(CONFIG.get("STRAIGHT_GAIN", 0.40))
516         if dev_abs <= thr and thr > 1e-6:
517             frac = (thr - dev_abs) / thr
518             vx = CONFIG["VX_BASE"] * (1.0 + gain * frac)
519         else:
520             vx = CONFIG["VX_BASE"] * (1.0 - min(0.8,
dev_abs / 35.0))
521         vx = float(clamp(vx, CONFIG["VX_MIN"], CONFIG["
VX_MAX"]))
522
523         # publish when armed
524         if armed and (now - last_pub) >= (1.0 / CONFIG["
PUB_HZ"]):
525             try:
526                 send_command(json.dumps({"command": "
trajectory_angle", "angle": float(angle_s), "vx_mps":
float(vx)}))
527             except Exception:
528                 pass
529

```

```

530         last_pub = now
531
532         # Build displays:
533         head_vis = overlay_heading(bev, lane_lines,
angle_s, nudge_applied=nudge)
534         left_large = cv2.resize(head_vis, (LAYOUT["
LEFT_W"], LAYOUT["LEFT_H"]))
535
536         vis_trap = f.copy()
537         poly = trap.astype(np.int32).reshape((-1, 1, 2))
538         cv2.polylines(vis_trap, [poly], True, (0, 255,
255), 2)
539         right_top = cv2.resize(vis_trap, (LAYOUT["
RIGHT_W"], LAYOUT["RIGHT_TOP_H"]))
540
541         mask_color = cv2.cvtColor(mask, cv2.
COLOR_GRAY2BGR)
542         edges_color = cv2.cvtColor(edges, cv2.
COLOR_GRAY2BGR)
543         combined_mask = np.hstack([mask_color,
edges_color])
544         # take left portion and resize to the right
panel size
545         right_mid = cv2.resize(combined_mask[:, :LAYOUT[
"RIGHT_W"]], (LAYOUT["RIGHT_W"], LAYOUT["RIGHT_MID_H"]))
546
547         # stats panel sized dynamically
548         stat_panel = np.zeros((LAYOUT["RIGHT_BOTTOM_H"],
LAYOUT["RIGHT_W"], 3), dtype=np.uint8)
549         stat_panel[:] = (30, 30, 30)
550         txt_y = 30
551         line_h = 22
552         cv2.putText(stat_panel, f"ARMED: {'YES' if armed
else 'NO'}", (8, txt_y), cv2.FONT_HERSHEY_SIMPLEX, 0.6,
553                     (0, 200, 0) if armed else (0, 80,
200), 2)
554         txt_y += line_h
555         cv2.putText(stat_panel, f"Angle: {angle_s:.2f}
deg", (8, txt_y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,
255, 255), 1)
556         txt_y += line_h
557         cv2.putText(stat_panel, f"Vx: {vx:.2f} m/s", (8,
txt_y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255),
1)
558         txt_y += line_h

```

```

559         cv2.putText(stat_panel, f"HSV L: {hsv_lower}",
(8, txt_y), cv2.FONT_HERSHEY_SIMPLEX, 0.38, (200, 200,
200), 1)
560         txt_y += line_h
561         cv2.putText(stat_panel, f"HSV H: {hsv_upper}",
(8, txt_y), cv2.FONT_HERSHEY_SIMPLEX, 0.38, (200, 200,
200), 1)
562         txt_y += line_h
563         cv2.putText(stat_panel, f"FPS: {fps:.1f}", (8,
txt_y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (200, 200, 0), 1)
564         txt_y += line_h
565         cv2.putText(stat_panel, f"nudge: {nudge:+.2f}",
(8, txt_y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (180, 180,
180), 1)
566
567         # Compose right column (dynamic heights)
568         right_col_h = LAYOUT["RIGHT_TOP_H"] + LAYOUT["
RIGHT_MID_H"] + LAYOUT["RIGHT_BOTTOM_H"]
569         right_col = np.zeros((right_col_h, LAYOUT["
RIGHT_W"], 3), dtype=np.uint8)
570         y0 = 0
571         right_col[y0:y0 + LAYOUT["RIGHT_TOP_H"], :, :] =
right_top; y0 += LAYOUT["RIGHT_TOP_H"]
572         right_col[y0:y0 + LAYOUT["RIGHT_MID_H"], :, :] =
right_mid; y0 += LAYOUT["RIGHT_MID_H"]
573         right_col[y0:y0 + LAYOUT["RIGHT_BOTTOM_H"], :,
:] = stat_panel
574
575         # Compose full canvas
576         canvas_h = max(LAYOUT["LEFT_H"], right_col_h)
577         canvas_w = LAYOUT["LEFT_W"] + LAYOUT["RIGHT_W"]
578         canvas = np.zeros((canvas_h, canvas_w, 3), dtype
=np.uint8)
579         # center left vertically if canvas_h > LEFT_H
580         y_off_left = (canvas_h - LAYOUT["LEFT_H"]) // 2
581         canvas[y_off_left:y_off_left + LAYOUT["LEFT_H"],
0:LAYOUT["LEFT_W"], :] = left_large
582         # place right column at top-right
583         canvas[0:right_col_h, LAYOUT["LEFT_W"]:LAYOUT["
LEFT_W"] + LAYOUT["RIGHT_W"], :] = right_col
584
585         # UI hint bar bottom
586         hint_h = 36
587         cv2.rectangle(canvas, (0, canvas_h - hint_h), (
canvas_w, canvas_h - 1), (10, 10, 10), -1)

```

```

588         cv2.putText(canvas, "s: arm/start  w: save trap
r: reload trap  q: quit", (10, canvas_h - 12),
589                     cv2.FONT_HERSHEY_SIMPLEX, 0.6, (200,
200, 200), 1)

590
591         cv2.imshow(window_name, canvas)
592         key = cv2.waitKey(1) & 0xFF
593
594         if key == ord('q'):
595             if armed:
596                 try:
597                     send_stop()
598                 except Exception:
599                     pass
600             break
601         elif key == ord('s'):
602             armed = not armed
603             if armed:
604                 try:
605                     send_activate()
606                 except Exception:
607                     pass
608             else:
609                 try:
610                     send_stop()
611                 except Exception:
612                     pass
613         elif key == ord('w'):
614             try:
615                 trap_list = trap.astype(int).tolist()
616                 with open("src_trap.json", "w") as fjs:
617                     json.dump(trap_list, fjs)
618                 print("Saved src_trap.json:", trap_list)
619             except Exception as e:
620                 print("Error saving trap:", e)
621         elif key == ord('r'):
622             loaded = load_src_trap("src_trap.json")
623             if isinstance(loaded, np.ndarray) and loaded
        .shape == (4, 2):
624                 xL, y_top = loaded[0]; xR, _ = loaded
        [1]; _, y_bot = loaded[2]
625                 top_off = int(round(y_top - horizon_y))
626                 bot_mrg = int(round((cam_h - 1) - y_bot)
        )
627                 l_mrg = int(round(xL))

```

```

628         r_mrg = int(round((cam_w - 1) - xR))
629         # update trackbars to reflect reloaded
        trap
630         try:
631             cv2.setTrackbarPos("TOP_OFF(px)",
        trap_win, int(top_off))
632             cv2.setTrackbarPos("BOT_MRG(px)",
        trap_win, int(bot_mrg))
633             cv2.setTrackbarPos("L_MRG(px)",
        trap_win, int(l_mrg))
634             cv2.setTrackbarPos("R_MRG(px)",
        trap_win, int(r_mrg))
635             print("Reloaded src_trap.json")
636             except Exception:
637                 pass
638         else:
639             print("No src_trap.json to reload or
        invalid format")
640
641         finally:
642             cam.release()
643             cv2.destroyAllWindows()
644
645
646 if __name__ == "__main__":
647     src = None
648     try:
649         src = load_src_trap("src_trap.json")
650     except Exception:
651         src = None
652     run_single_window_with_trackbars(hsv_lower=CONFIG["
        HSV_LOWER"], hsv_upper=CONFIG["HSV_UPPER"], src_trap=src)
653     try:
654         client.loop_stop()
655         client.disconnect()
656     except Exception:
657         pass
658     print("Exit.")

```

Appendix B

Follow_Road

```
1
2 import math
3 import json
4 from time import time
5 from typing import Tuple, Optional
6
7 import numpy as np
8 import paho.mqtt.client as mqtt
9
10 # ROS2
11 import rclpy
12 from rclpy.node import Node
13 from sensor_msgs.msg import LaserScan
14
15 # Libreria hardware Yahboom / Rosmaster
16 from Rosmaster_Lib import Rosmaster
17
18 RAD2DEG = 180.0 / math.pi
19 DEG2RAD = math.pi / 180.0
20
21
22 class FollowRoad(Node):
23     def __init__(self, name: str = "follower_road_fast"):
24         super().__init__(name)
25
26         # ---- Car (STM32 / Rosmaster) setup ----
27         self.car = Rosmaster()
28         self.car.set_car_type(1)
29         self.car.create_receive_threading()
30
```

```

31         # ---- MQTT ----
32         # Compat Paho 2.x: prova API v1, fallback a default
33         try:
34             self.mqtt_client = mqtt.Client(mqtt.
CallbackAPIVersion.VERSION1)
35         except Exception:
36             self.mqtt_client = mqtt.Client()
37             self.mqtt_client.on_connect = self.on_connect
38             self.mqtt_client.on_message = self.on_message
39         try:
40             self.mqtt_client.connect("localhost", 1883, 60)
41             self.mqtt_client.loop_start()
42         except Exception as e:
43             self.get_logger().error(f"MQTT connect error: {e
}")
44
45         # ---- Parametri ----
46         # Velocità nominale (m/s)
47         self.declare_parameter("linear_speed", 0.20)
48         self.linear_speed = float(self.get_parameter("
linear_speed").value)
49
50         # Guadagno angolare:  $w_z = \text{angular\_gain} * \text{angle\_deg}$ 
51         self.declare_parameter("angular_gain", 0.02)
52         self.angular_gain = float(self.get_parameter("
angular_gain").value)
53
54         # Inversione segno sterzo (se meccanica invertita)
55         self.declare_parameter("invert_sign", True)
56         self.INVERT_SIGN = bool(self.get_parameter("
invert_sign").value)
57
58         # Anteprima in BEV per calcolo angolo da coeffs (
pixel)
59         self.declare_parameter("preview_y_px", 120.0)
60         self.preview_y_px = float(self.get_parameter("
preview_y_px").value)
61
62         # Feed-forward di curvatura ( $w_z_{ff} = k_{ff} * v_x *
kappa$ )
63         self.declare_parameter("k_ff", 1.0)
64         self.k_ff = float(self.get_parameter("k_ff").value)
65
66         # Timeout dati (s)
67         self.declare_parameter("coeffs_timeout_s", 0.5)

```

```

68         self.coeffs_timeout_s = float(self.get_parameter("
coeffs_timeout_s").value)
69         self.declare_parameter("angle_timeout_s", 0.3)
70         self.angle_timeout_s = float(self.get_parameter("
angle_timeout_s").value)
71
72         # Rate-limit (deg/s) e ramp vx (m/s^2)
73         self.declare_parameter("wz_rate_dps_base", 120.0)
74         self.wz_rate_dps_base = float(self.get_parameter("
wz_rate_dps_base").value)
75         self.declare_parameter("wz_rate_dps_boost", 6.0) #
+boost per grado di delta
76         self.wz_rate_dps_boost = float(self.get_parameter("
wz_rate_dps_boost").value)
77         self.declare_parameter("vx_accel", 1.0) # m/s^2
78         self.vx_accel = float(self.get_parameter("vx_accel")
.value)
79         # 1) Dentro __init__ (dopo gli altri
declare_parameter) aggiungi:
80         self.declare_parameter("angle_deadband_deg", 3.5)
# nessuna sterzata sotto questa soglia
81         self.declare_parameter("angle_knee_deg", 12.0)
# ginocchio della curva
82         self.declare_parameter("angular_gain_lo", 0.01) #
guadagno per |ang| <= knee
83         self.declare_parameter("angular_gain_hi", 0.05) #
guadagno per |ang| > knee
84
85         self.angle_deadband_deg = float(self.get_parameter("
angle_deadband_deg").value)
86         self.angle_knee_deg = float(self.get_parameter("
angle_knee_deg").value)
87         self.angular_gain_lo = float(self.get_parameter("
angular_gain_lo").value)
88         self.angular_gain_hi = float(self.get_parameter("
angular_gain_hi").value)
89
90         # Safety / LIDAR
91         self.declare_parameter("distance_front", 2.0)
92         self.distance_front = float(self.get_parameter("
distance_front").value)
93         self.declare_parameter("response_dist_front", 0.5)
94         self.response_dist_front = float(self.get_parameter(
"response_dist_front").value)
95         self.laser_angle = 130.0 # non usato: compat

```

```

96
97     # ---- Stati ----
98     self.ready: bool = False
99     self.follow_line_active: bool = False
100    self.emergency_stopped: bool = False
101    self.obstacle_detected: bool = False
102
103    # Dati traiettoria
104    self._coeffs: Optional[Tuple[float, float, float]] =
None
105    self._coeffs_time: float = 0.0
106    self._angle_cmd_deg: float = 0.0
107    self._angle_time: float = 0.0
108
109    self._last_command_time = time()
110    self.watchdog_timeout = 2.0 # sec
111
112    # Stato controllo
113    self._current_v = 0.0
114    self._last_wz = 0.0 # rad/s
115
116    # ---- ROS ----
117    self.set_led_color(0, 0, 255)
118    self.sub_laser = self.create_subscription(LaserScan,
"/scan", self.lidar_callback, 10)
119
120    # Timer rapidi
121    self.control_dt = 0.02 # 50 Hz
122    self.control_timer = self.create_timer(self.
control_dt, self.control_loop)
123    self.watchdog_timer = self.create_timer(0.5, self.
_check_watchdog)
124    # ----- Utils robot -----
125    def set_led_color(self, r: int, g: int, b: int):
126        try:
127            self.car.set_colorful_lamps(0xFF, r, g, b)
128        except Exception as e:
129            self.get_logger().warning(f"set_led_color failed
: {e}")
130
131    def set_motion(self, vx: float, vy: float, wz: float):
132        """Wrapper compatibile con varianti Rosmaster."""
133        try:
134            self.car.set_car_motion(vx, vy, wz)
135        except TypeError:

```

```

136         # alcune versioni hanno set_car_motion(vx, wz)
137         try:
138             self.car.set_car_motion(vx, wz)
139         except Exception as e2:
140             self.get_logger().warning(f"set_car_motion(
vx,wz) failed: {e2}")
141         except Exception as e:
142             self.get_logger().warning(f"set_car_motion
failed: {e}")
143
144     def _check_watchdog(self):
145         if self.follow_line_active and (time() - self.
_last_command_time) > self.watchdog_timeout:
146             self.get_logger().warning("Watchdog timeout:
entering idle")
147             self.follow_line_active = False
148             self.set_motion(0.0, 0.0, 0.0)
149             self.set_led_color(255, 0, 0)
150
151     # ----- MQTT -----
152     def on_connect(self, client, userdata, flags, rc,
properties=None):
153         self.get_logger().info(f"MQTT connected rc={rc}")
154         self.ready = True
155         self.set_led_color(0, 0, 255)
156         client.subscribe("robot/control")
157
158     def _update_last_command(self):
159         self._last_command_time = time()
160
161     def on_message(self, client, userdata, msg):
162         """Riceve i comandi dal follow_line"""
163         try:
164             message = json.loads(msg.payload.decode("utf-8")
)
165         except Exception as e:
166             self.get_logger().warning(f"Invalid MQTT payload
: {e}")
167             return
168
169         self._update_last_command()
170         cmd = message.get("command", "")
171
172         if cmd == "activate":
173             self.follow_line_active = True

```

```

174         self.emergency_stopped = False
175         self.set_led_color(0, 255, 0)
176         self.get_logger().info("Follow-Line activated")
177         return
178
179     if cmd == "deactivate":
180         self.follow_line_active = False
181         self._current_v = 0.0
182         self._last_wz = 0.0
183         self.set_motion(0.0, 0.0, 0.0)
184         self.set_led_color(0, 0, 255)
185         self.get_logger().info("Follow-Line deactivated")
186     )
187     return
188
189     if cmd in ("emergency_stop", "stop"):
190         self.follow_line_active = False
191         self.emergency_stopped = (cmd == "emergency_stop")
192
193         self.set_motion(0.0, 0.0, 0.0)
194         self.set_led_color(255, 0, 0)
195         self.get_logger().warning(cmd.replace("_", " ").title())
196     return
197
198     if cmd == "backward" and self.follow_line_active:
199         self.set_motion(-abs(self.linear_speed) * 0.2,
200 0.0, 0.0)
201         self.set_led_color(255, 0, 0)
202         self.get_logger().info("Robot moving backwards (MQTT command)")
203     return
204
205     if cmd == "trajectory_motion" and self.follow_line_active:
206         try:
207             vx = float(message.get("vx", 0.0))
208             vy = float(message.get("vy", 0.0))
209             wz = float(message.get("wz", 0.0))
210             self.set_motion(vx, vy, wz)
211         except Exception as e:
212             self.get_logger().warning(f"trajectory_motion failed: {e}")
213     return

```

```

212         if cmd == "trajectory_coeffs" and self.
follow_line_active:
213             # Telemetria/diagnostica
214             coeffs = message.get("coeffs", [0, 0, 0])
215             try:
216                 a, b, c = coeffs
217                 self._coeffs = (float(a), float(b), float(c))
218
219                 self._coeffs_time = time()
220                 self.get_logger().info(f"Coeffs received: a
={a:.4f}, b={b:.4f}, c={c:.4f}")
221             except Exception:
222                 self.get_logger().warning("Invalid coeffs
payload")
223             return
224
225         if cmd == "trajectory_angle":
226             if not self.follow_line_active:
227                 self.get_logger().debug("trajectory_angle
while inactive; ignored")
228                 return
229
230             # log RX base
231             self.get_logger().info(f"[rx] trajectory_angle
angle={message.get('angle')} vx_mps={message.get('vx_mps
')}")
232
233             # 1) angolo e vx target
234             try:
235                 angle_deg = float(message.get("angle", 0.0))
236                 # deviazione centrata 0 [deg]
237             except Exception:
238                 angle_deg = 0.0
239             try:
240                 vx_target = float(message.get("vx_mps", self
.linear_speed))
241             except Exception:
242                 vx_target = float(self.linear_speed)
243
244             # 2) angolo -> wz (clip + invert)
245             wz_cmd = self.compute_angular_from_angle(
angle_deg)
246
247             # 3) ramp vx
248             v_step = 0.03

```

```

247         if self._current_v < vx_target:
248             self._current_v = min(self._current_v +
v_step, vx_target)
249         else:
250             self._current_v = max(self._current_v -
v_step, vx_target)
251
252         # 4) invio comandi
253         try:
254             try:
255                 self.car.set_pid_param(0.1, 3, 3, 0)
256             except Exception:
257                 pass
258             self.set_motion(float(self._current_v), 0.0,
float(wz_cmd))
259         except Exception as e:
260             self.get_logger().warning(f"set_car_motion
failed: {e}")
261
262             self.set_led_color(0, 255, 0)
263             self.get_logger().info(f"[angle] deg={angle_deg
:+.2f} wz={wz_cmd:+.3f} vx={self._current_v:+.2f}")
264             # aggiorna timestamp "recente" per l'uso in
control_loop
265             self._angle_cmd_deg = angle_deg
266             self._angle_time = time()
267             return
268
269             self.get_logger().info(f"Unknown or ignored MQTT
command: {cmd}")
270
271         # ----- LIDAR -----
272         def lidar_callback(self, scan_data: LaserScan) -> None:
273             try:
274                 ranges = np.array(scan_data.ranges, dtype=np.
float32)
275             except Exception:
276                 return
277
278             obs = False
279             ang_min = scan_data.angle_min
280             dang = scan_data.angle_increment
281             for i, d in enumerate(ranges):
282                 if not np.isfinite(d):
283                     continue

```

```

284         ang = (ang_min + dang * i) * RAD2DEG
285         if abs(ang) > 160 and d < self.
response_dist_front:
286             obs = True
287             break
288         self.obstacle_detected = obs
289
290     # ----- Helpers controllo -----
291
292     def compute_angular_from_angle(self, angle_deg: float):
293         max_angle = 45.0
294         a = float(max(min(angle_deg, max_angle), -max_angle)
)
295         s = 1.0 if a >= 0 else -1.0
296         A = abs(a)
297
298         db = self.angle_deadband_deg
299         if A <= db:
300             wz = 0.0
301         else:
302             # rimuovi il deadband dalla misura
303             Aeff = A - db
304             knee = max(1e-3, self.angle_knee_deg - db) #
knee relativo dopo deadband
305
306             if Aeff <= knee:
307                 # regione "dolce"
308                 wz = self.angular_gain_lo * Aeff
309             else:
310                 # oltre il ginocchio aumenta il guadagno
311                 wz = self.angular_gain_lo * knee + self.
angular_gain_hi * (Aeff - knee)
312
313             wz *= s
314
315             # inversione opzionale
316             if self.INVERT_SIGN:
317                 wz = -wz
318
319             return float(wz)
320
321     def compute_wz_from_angle(self, angle_deg: float):
322         return self.compute_angular_from_angle(angle_deg)
323

```

```

324     def angle_from_coeffs(self, a: float, b: float,
325     y_preview: float):
326         dy = 2.0 * a * y_preview + b
327         return -math.degrees(math.atan(dy))
328
329     def kappa_from_coeffs(self, a: float, b: float,
330     y_preview: float):
331         dy = 2.0 * a * y_preview + b
332         return (2.0 * a) / ((1.0 + dy * dy) ** 1.5 + 1e-9)
333
334     # ----- Control loop 50 Hz -----
335     def control_loop(self):
336         if not self.ready or self.emergency_stopped:
337             return
338
339         # Safety / idle
340         if self.obstacle_detected or not self.
341         follow_line_active:
342             self.set_led_color(255 if self.obstacle_detected
343             else 0, 0, 255 if not self.follow_line_active else 0)
344             self._current_v = max(0.0, self._current_v -
345             self.vx_accel * self.control_dt)
346             self.set_motion(0.0, 0.0, 0.0)
347             return
348
349             self.set_led_color(0, 255, 0)
350
351             now = time()
352             have_coeffs = (now - self._coeffs_time) < self.
353             coeffs_timeout_s and (self._coeffs is not None)
354             have_angle = (now - self._angle_time) < self.
355             angle_timeout_s
356
357             wz_ff = 0.0
358             angle_ref_deg = 0.0
359
360             if have_coeffs:
361                 a, b, _ = self._coeffs
362                 angle_ref_deg = self.angle_from_coeffs(a, b,
363                 self.preview_y_px)
364                 kappa = self.kappa_from_coeffs(a, b, self.
365                 preview_y_px)
366                 vz = max(0.0, self.linear_speed)
367                 wz_ff = self.k_ff * vz * kappa

```

```

360         elif have_angle:
361             angle_ref_deg = float(self._angle_cmd_deg)
362             wz_ff = 0.0
363         else:
364             self._current_v = max(0.0, self._current_v -
self.vx_accel * self.control_dt)
365             self.set_motion(0.0, 0.0, 0.0)
366             return
367
368
369         wz_fb = self.compute_wz_from_angle(angle_ref_deg)
370
371
372         wz_target = float(wz_fb + wz_ff)
373
374
375         delta = wz_target - self._last_wz
376         delta_deg = abs(angle_ref_deg)
377         rate_dps = self.wz_rate_dps_base + self.
wz_rate_dps_boost * min(10.0, delta_deg)
378         max_step = (rate_dps * DEG2RAD) * self.control_dt
379         wz_cmd = self._last_wz + math.copysign(max_step,
delta) if abs(delta) > max_step else wz_target
380         self._last_wz = float(wz_cmd)
381
382
383         target_v = max(0.0, self.linear_speed)
384         dv = self.vx_accel * self.control_dt
385         if self._current_v < target_v:
386             self._current_v = min(self._current_v + dv,
target_v)
387         else:
388             self._current_v = max(self._current_v - dv,
target_v)
389
390
391         self.set_motion(self._current_v, 0.0, wz_cmd)
392
393
394         if int(time() * 2) % 10 == 0:
395
396
397         f"[{src}] ang_ref={angle_ref_deg:+.2f} wz_fb = {wz_fb:+.3f}
wz_ff = {wz_ff:+.3f} | vx = {self._current_v:.2f}"
398     )

```

```
399
400     # ----- Entrypoint helpers -----
401     def destroy_node(self):
402         try:
403             self.set_motion(0.0, 0.0, 0.0)
404             self.set_led_color(0, 0, 0)
405         except Exception:
406             pass
407         try:
408             self.mqtt_client.loop_stop()
409         except Exception:
410             pass
411         super().destroy_node()
412
413
414     def main(args=None):
415         rclpy.init(args=args)
416         node = FollowRoad()
417         try:
418             rclpy.spin(node)
419         except KeyboardInterrupt:
420             node.get_logger().info("KeyboardInterrupt: stopping
robot")
421         finally:
422             try:
423                 node.destroy_node()
424             except Exception:
425                 pass
426             rclpy.shutdown()
427
428
429     if __name__ == "__main__":
430         main()
```