Design and Implementation of an Autonomous Robot Controlled by an Optical Camera-Based Image Processing System

Elena-Minodora ACATINCĂ and Mihai COCA

***Abstract*—This paper presents the development of an autonomous vehicle capable of following a predefined path using data from a Pixy2 camera and an ultrasonic sensor for line and obstacle detection. The system is managed by the RDDRONE-FMUK66 microcontroller, which directly acquires a set of vectors from the Pixy2 vision sensor. Based on the received vectors, a Proportional-Integral-Derivative (PID) algorithm is applied to correct lateral deviations, thereby enabling the vehicle to maintain its trajectory along the track. The software logic is organized as a Finite State Machine (FSM) that manages initialization, data acquisition, vector processing, and control command generation. To ensure testing under reproducible and controllable conditions, the system is replicated within a virtual environment using Gazebo, integrated with the Robot Operating System (ROS 2), where the camera is modeled through simulation. The data generated within the simulator can be accessed via the microcontroller's serial interface, enabling observation and analysis of the system's response. This approach enables the embedded application to operate seamlessly with both real and simulated data, allowing the same control logic to be applied across both environments without the need for modification. The results obtained in both real and simulated tests confirm the stable operation of the PID algorithm, the system's ability to correctly react to obstacles, and the efficiency of path following. The proposed architecture provides flexibility, reliability, and the potential to extend functionality in future robotics applications.**

***Index Terms*—autonomous vehicle, embedded system, FSM, Gazebo simulation, PID control, ROS 2.**

I. Introduction

Autonomous systems are rapidly evolving in the current context, being used in a wide range of applications, from industrial robots to self-driving vehicles or exploration platforms. Missions such as NASA's Curiosity rover have demonstrated the effectiveness of technologies that combine environmental perception and real-time control in the absence of a human operator [1]. These advances have also highlighted the need for safe testing environments prior to deployment on real hardware [2].

This paper focuses on the development of an autonomous robot capable of following a path on a white surface delimited by two black lines and stopping when it senses an obstacle or derails from the track. The system combines real hardware implementation with a virtual simulation environment to enable rapid testing and adjustment of the robot's behavior. The platform is built around the RDDRONE-FMUK66 board and includes a Pixy2 smart camera, which detects the track lines, along with a proximity sensor used for obstacle detection. All components are mounted on a chassis compatible with the NXP kit from the NXP Cup competition.

To create a safe testing environment, this system integrates a simulator built in Gazebo, connected through ROS 2. The simulator enables modeling of the track, sensor behavior, and robot movements within a controlled environment that replicates real operating conditions. In this way, the control logic can be tested and calibrated without the risk of damaging physical components, and the transition from simulation to real-world execution can be achieved without major changes in structure or functionality.

II. System Architecture

The developed system integrates multiple hardware and software components to enable the autonomous operation of a mobile platform capable of following a track and reacting to obstacles.

The overall architecture, illustrated in Fig. 1, is centered around the RDDRONE-FMUK66 microcontroller (MCU) [3], which controls the entire system. It receives a set of vectors from the Pixy2 camera [4] via the Inter-Integrated Circuit (I2C) interface, and these data are used to determine the position relative to the two boundary lines. At the same time, the HC-SR04 proximity sensor [5] sends information about obstacles in front of the robot using digital signals transmitted through General-Purpose Input/Output (GPIO) pins. Based on this information, the application developed on the MCU continuously analyzes the state of the environment and decides how the robot should react.

Steering and speed commands are generated using a PID algorithm [6-7], which calculates the necessary corrections to keep the robot on the track. These commands are transmitted as Pulse Width Modulation (PWM) signals to two components: the servomotor, which controls the

E. M. ACATINCĂ is with the Military Technical Academy "Ferdinand I", Bucharest, Romania (e-mail: elena.acatinca@mta.ro).
M. COCA is with the Military Technical Academy "Ferdinand I", Bucharest, Romania (e-mail: mihai.coca@mta.ro).

Digital Object Identifier 10.32754/JMT.2026.1.08

steering, and the drive motor, connected to an ESC controller, which determines the speed. The system's structure enables real-time autonomous operation and provides the flexibility to integrate additional components or sensors if needed.

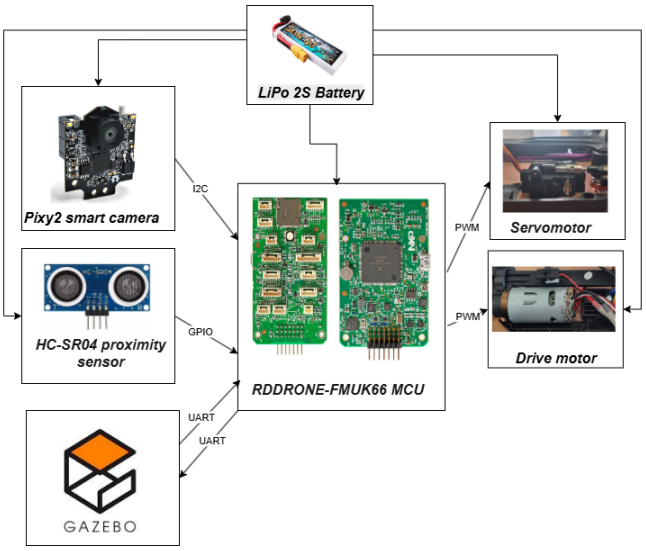

Figure 1. System architecture

The system operates based on an FSM illustrated in Fig. 2, which controls transitions and actions according to the conditions detected in real time. Execution begins in the initialization state, *STATE_INIT*, where all hardware components are configured and the sensors are calibrated. Once this step is completed, the system transitions to the waiting state, *STATE_WAIT*, where it remains idle until it receives a signal from a timing peripheral, i.e., Programmable Interrupt Timer (PIT). If a stop command is activated, the system immediately enters the stop state, *STATE_STOP*, in which all commands to the motors are halted and movement is interrupted. In the absence of such a command, the system proceeds to the obstacle checking state, *STATE_CHECK_OBSTACLE*. In this phase, the signal from the proximity sensor is interpreted to determine whether there is an object in the robot's path. If an obstacle is detected, the system returns to the stop state to prevent a collision. If the path is clear, the system enters the optical data acquisition state, *STATE_READ_CAMERA*. In this state, data comes either from the Pixy2 camera or from the simulated camera. The simulator replaces the real camera, allowing the system to be tested under ideal conditions.

The information received consists of vectors that describe the contrasting segments detected on the track. The Pixy2 camera runs a dedicated application called line tracking mode, which specializes in detecting dark-colored lines on light surfaces. The transmitted vectors correspond to these lines [8]. These data are then processed in the next state, STATE_PROCESS_VECTOR. In this state, the vectors are validated, filtered, and analyzed. Those closest to the ideal center of the track and best describing its edges are selected. Isolated, short, or unstable vectors are discarded.

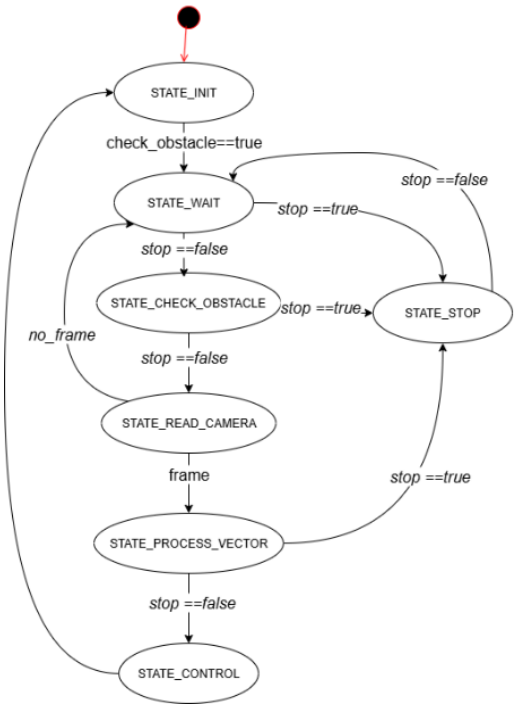

Figure 2. Finite State Machine (FSM) diagram

The result of this processing is used in the control state, *STATE_CONTROL*, where the robot's lateral error relative to the center of the track is calculated, and a PID algorithm is applied to generate steering and speed commands. If a stop condition occurs during any of these stages, the system immediately returns to the safety state, stopping the vehicle in a controlled manner.

The set of vectors can be retrieved from two different sources, through separate communication channels: from the Pixy2 camera via I2C, and from the simulator via Universal Asynchronous Receiver/Transmitter (UART) interface. This is made possible by the modularity of the software architecture and the separation between data acquisition and data processing. Fig. 3 shows the fully assembled robot, with all components mounted on the chassis, ready for testing in either the real or simulated environment.

Figure 3. Fully assembled autonomous robot

 Digital Object Identifier 10.32754/JMT.2026.1.08

III. System Implementation

The software implementation of the autonomous platform is structured in a modular way. The hardware components are managed by a microcontroller, which runs a logic organized as a finite state machine. The Pixy2 smart camera transmits track data in the form of vectors that define the segments of the detected lines, using a coordinate system with the origin in the top-left corner and fixed value ranges for the X-axis (0–78) and Y-axis (0–51).

The Pixy2 camera returns a variable number of vectors per frame, depending on the artifacts of the dark lines present on a light background. However, practical tests have shown that a maximum of eight vectors is sufficient for efficient and stable road direction detection. If fewer vectors are detected, the remaining ones are automatically filled with invalid values, specifically value 128, which lies outside the coordinate system, to preserve the constant structure of the data.

A real capture of the received vectors is presented in Fig. 4. A right-hand curve is clearly visible. However, the detected vectors are fragmented, disordered, and in some cases useless for identifying the track. This capture was obtained with the Pixy2 camera hand-held over a white surface with several black lines. If the first two received vectors were used directly without additional verification, the vehicle might misinterpret the track and generate an unjustified correction, which would compromise the stability of the movement.

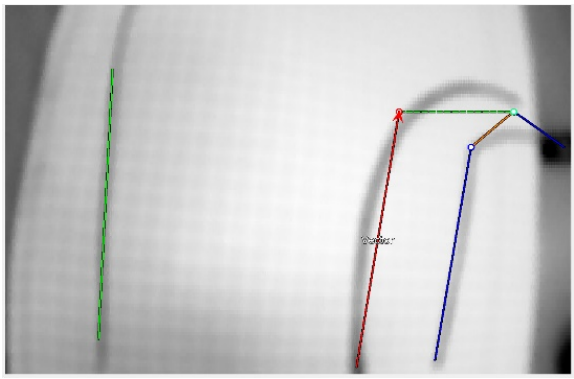

Figure 4. Capture of vector perception

Fig. 5 graphically illustrates the set of vectors obtained based on the next frame relative to the frame captured in Fig. 4. False vectors caused by shadows or objects near the track are present. Fragmented vectors are also included. The goal is to keep only the vectors that accurately describe the real track boundaries. Invalid vectors are automatically removed, such as those with zero length, incorrect coordinates, or predefined values associated with detection errors. Vectors that are too short, which do not provide a clear direction and may introduce unwanted fluctuations in trajectory calculation, are also excluded. Through this filtering process, only the vectors that consistently describe the track boundaries are retained. This filtering and validation process is illustrated in Listing 1, which outlines the main steps used to eliminate irrelevant or noisy vectors and retain only those that reliably represent the track boundaries. The actual validation logic, where each vector is checked and added to the list of valid ones, is implemented in lines 6-8.

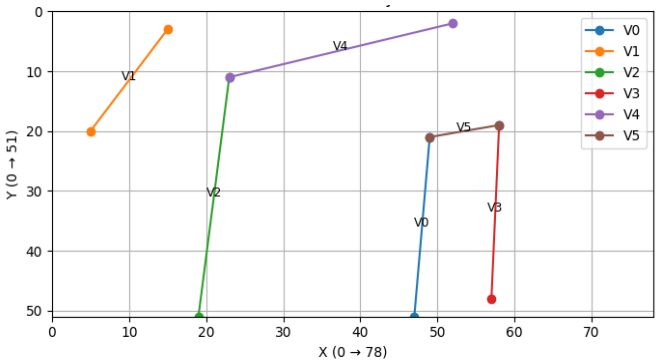

Figure 5. Initial detection of line vectors from the camera

In the next stage, all remaining vectors are reoriented so that their direction is from bottom to top, regardless of how they were initially detected by the camera (lines 4-5 of Listing 1). This uniformization of direction is essential for vectors to be correctly compared and connected. Without this correction, the process of combining segments could yield incorrect results, as the geometric orientation of each vector would vary.

Once aligned, the vectors are analyzed to identify segments that belong to the same boundary. If two or more endpoints are sufficiently close in space, the corresponding vectors are considered continuous and are merged into a longer segment (line 14 of Listing 1). This results in a clearer and more stable representation of the track. This process is illustrated in Fig. 6, where the fragmented vectors V2 and V4 are joined to form a new vector, V3, which accurately represents the direction of the outer line.

Finally, the algorithm selects two vectors considered the most representative for describing the left and right edges of the track, based on two essential criteria: the slope and the distance from the center of the image. To determine which edge each vector belongs to, the slope of the segment is analyzed. Vectors with a negative slope are considered to belong to the left edge, while those with a positive slope are associated with the right edge. In this way, the vectors are divided into two groups. This filtering and selection process is implemented in lines 19-26 of Listing 1, where each vector is evaluated based on slope and distance to the ideal center. However, if multiple vectors share the same slope, the algorithm retains only the one closest to the ideal center of the image. This choice is necessary because it is not possible to determine with certainty, based solely on slope and position, whether a vector belongs to the left or right edge. In some cases, a false vector (for example, caused by a shadow) may be closer to the center than the real ones and will be retained instead. Therefore, keeping only one vector for each direction is a compromise solution that reduces classification errors, even though it may sometimes exclude

a correct vector. After the vectors have been grouped, the distance between their central point and the center of the camera image is calculated. Among all the vectors in a group, the one closest to the center is chosen as the most suitable to describe that edge. In this way, a single vector is selected for each side of the track, forming a pair that is later used in the direction calculation; thus, *V0* in Fig. 7 is a filtered and processed vector.

```
Input: list of vectors inputVectors[1..n]
Output: the left and right track edges leftLine,
rightLine

 1. Initialize two empty lists: validVectors and
mergedVectors
 2. Set validCount to 0
 3.
 4. For each vector v in inputVectors
 5.  Normalize vector v so that (x0, y0) is the top
point
 6.  If v is valid (correct length and within image
bounds)
 7.   Add v to validVectors
 8.   Increment validCount
 9.
10. If validCount is zero
11.  Mark both leftLine and rightLine as invalid
12.  Set stop to 1 and exit the function
13.
14. Merge connected vectors from validVectors into
mergedVectors to create longer and more stable segments
15.
16. Initialize leftIdx and rightIdx to -1
17. Initialize minLeftDist and minRightDist to infinity
18.
19. For each vector v in mergedVectors
20.  Compute the center X coordinate: centerX = (x0 +
x1) / 2
21.  Compute the slope: slope = (x1 - x0) / (y1 - y0 +
0.01)
22.  Compute the distance to the ideal center: dist =
|centerX - CENTER_X|
23.  If slope < 0 and dist < minLeftDist
24.   Update leftIdx and minLeftDist
25.  If slope > 0 and dist < minRightDist
26.   Update rightIdx and minRightDist
27.
28. If leftIdx is valid
29.  Set leftLine = mergedVectors[leftIdx]
30. Else
31.  Set leftLine to a default vertical line on the
left
32.
33. If rightIdx is valid
34.  Set rightLine = mergedVectors[rightIdx]
35. Else
36.  Set rightLine to a default vertical line on the
right
37.
38. If both lines are invalid
39.  Set stop = 1
40. Else
41.  Set stop = 0
```

Listing 1. Pseudocode for filtering and processing the list of vectors

In situations where the camera fails to detect both edges, as occurs when the vehicle is too close to one side or when detection errors appear, the algorithm compensates for the missing edge by generating a virtual line. This fallback mechanism is handled in lines 33-36 of Listing 1, where a default line is generated when no real vector is available on one side. If only one real vector is available, the system automatically creates a second vector on the opposite side, in a standard position, to maintain balance in the analysis process and avoid errors in steering control. This approach is illustrated in Fig. 7, where the missing vector is replaced by a generated one, *V1*, allowing calculations to continue in a stable manner.

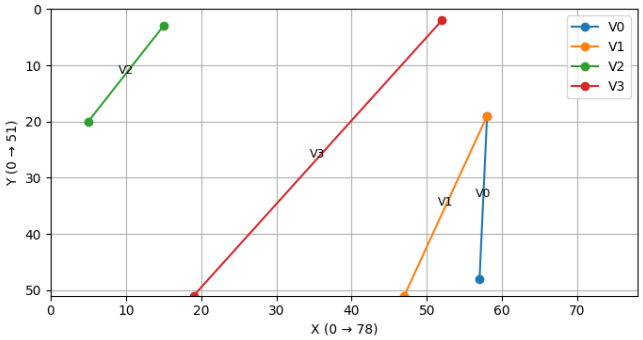

Figure 6. Filtered and merged vectors visualization

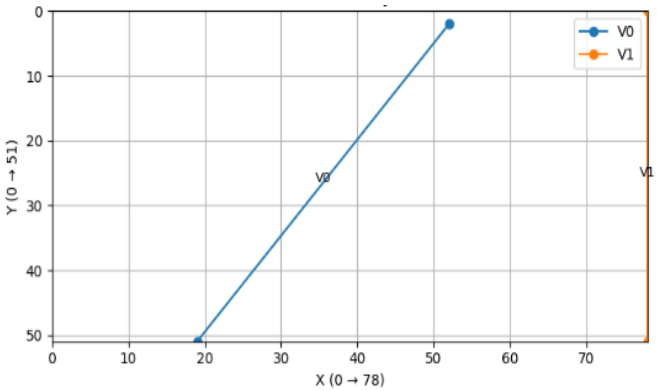

Figure 7. Filtered vectors used in the control algorithm

The final vectors obtained in this way are used to determine the vehicle's lateral position by calculating the lateral error through the comparison between their average position and the center of the camera image. This computation is shown in lines 1-2 of Listing 2. The resulting lateral error serves as the input for the PID algorithm, which generates the control command to keep the robot on track.

The PID algorithm uses three main components, each with a well-defined role in the system's response: the proportional component provides an immediate correction directly proportional to the current error, the integral component accumulates long-term deviations, and the derivative component dampens anticipated oscillations caused by sudden changes in position. These components are computed in lines 4-9 of Listing 2, which together define the control signal.

 Digital Object Identifier 10.32754/JMT.2026.1.08

```
Input: two lane vectors v1 and v2
Output: PWM signals for steering and speed

 1. Compute x_center from v1 and v2
 2. error ← CENTER_X - x_center
 3.
 4. integral ← integral + error
 5. Clamp integral to [-1.0, 1.0]
 6. derivative ← error - previous_error
 7. previous_error ← error
 8. control ← Kp * error + Ki * integral + Kd *
derivative
 9. Clamp control to [-1.0, 1.0]
10.
11. If control ≈ 0.0 → steering_pwm ← 700
12. If control > 0.0 → choose value >700
13. If control < 0.0 → choose value <700
14.
15. speed_pwm ← 770
16.
17. set_PWM(steering_motor, steering_pwm)
18. set_PWM(driving_motor, speed_pwm)
```

Listing 2. Pseudocode for the control algorithm using PID

The final control value is given by the mathematical formula of the PID control (1), which uses the coefficients K_p, K_i, and K_d. The implementation of this control logic is detailed in Listing 2, where the PID terms are computed based on the current error and applied to generate the steering command.

$$control = K_p \times error + K_i \times integral + K_d \times derivative \quad (1)$$

Based on empirical testing, the final chosen coefficients were $K_p = 0.045$, $K_i = 0$, and $K_d = 0.06$, resulting in a stable and accurate correction of the vehicle's direction. The control signal generated by the PID algorithm is then converted into a PWM pulse sent to the servomotor, directly determining the wheel position and, implicitly, the direction of movement. This mapping from control value to discrete PWM levels is performed in lines 11-13 of Listing 2. In parallel, the vehicle's speed can be adjusted according to the error and the complexity of the track, but in the current implementation stage, it is kept at a constant value (line 15 of Listing 2) to facilitate the evaluation and optimization of the PID algorithm's performance. All these processes are integrated into a coherent architecture based on successive states, which includes periodic track checking and controlled stopping in case of an obstacle or leaving the track. The final commands are sent to the motors in lines 17-18 of Listing 2.

To test and validate the behavior of the autonomous vehicle in a controlled environment, a simulator was developed based on the Gazebo and ROS 2 platforms. Gazebo is a 3D simulation software that enables realistic modeling of physical environments, including vehicle dynamics and the behavior of virtual sensors [9]. ROS 2 is a software framework designed for robotic applications, offering modular communication through nodes that exchange information via standardized messages [10]. The simulator created in Gazebo includes a simplified track built in Blender and a model of an autonomous vehicle, which receives control commands sent from the physical microcontroller via the UART interface.

The software architecture is composed of several specialized ROS 2 nodes, illustrated in Fig. 8. The nodes exchange data with each other using dedicated topics, with each node having a clearly defined role in the execution flow. In the diagram, the nodes are represented as rectangles, while the topics are shown as oval elements, in order to clearly highlight the separation between processing units and the communication channels between them.

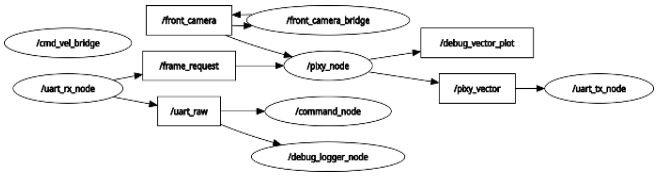

Figure 8. Structure of nodes and topics in the ROS 2 architecture

The main node for image processing is *pixy_node*, which mimics the behavior of the Pixy2 camera. It receives frames from a virtual front-mounted camera on the vehicle, shown in Fig. 9 (the left side), converts them to grayscale, and resizes them to the resolution specific to the Pixy2. The image is then binarized, resulting in a simplified and clear representation of the track lines.

From this image, the node extracts the main contours, generating vectors equivalent to those detected by the real camera. The result of the image processing and the representation of the extracted vectors are shown in Fig. 9 (the right side). The detected vectors are then transmitted through the *uart_tx_node* to the MCU serial interface for input to the PID algorithm. Communication between the simulator and the board is managed by the *uart_rx_node* and *uart_tx_node*, responsible for transmitting the vector set to the MCU and receiving the motor control commands via the UART interface, respectively. In the *command_node*, the commands generated by the MCU are converted into messages compatible with Gazebo, thereby controlling the speed and direction of the virtual vehicle.

The *debug_logger_node* ensures the logging of received data for later analysis of system performance. These data include the calculated lateral error, the control signal value generated by the PID algorithm, and the PWM values applied for steering and motor speed.

The entire ROS 2 architecture and the interconnections between nodes are shown in Fig. 8. This modular system allows testing and validation of the control algorithm in a virtual environment before physical implementation, providing a safe, repeatable, and efficient framework for development. The use of the simulator facilitates rapid parameter adjustment and the detection of potential issues before they occur in the real environment. In addition to the

previously mentioned nodes, the architecture also includes *front_camera_bridge*, which mediates the video data stream from the simulator to the visual processing system, and *cmd_vel_bridge*, which acts as a bridge node for the speed and steering commands generated by the controller, forwarding them to the communication interface with the physical MCU. These components ensure full integration between the simulator and the control logic, keeping the system structure clear and modular.

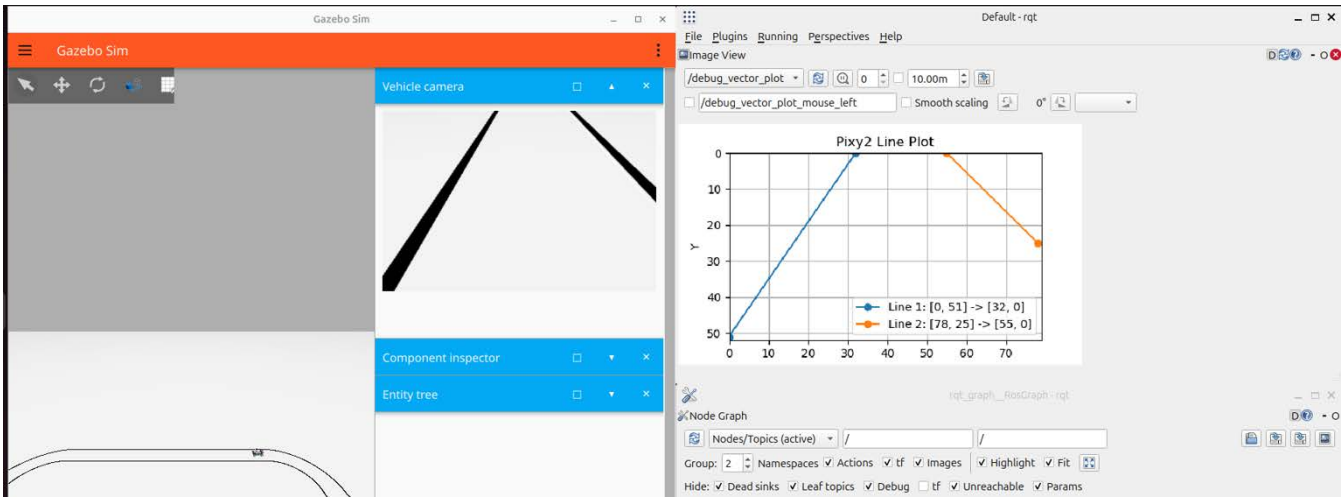

Figure 9. Virtual camera output and detected vector plot

IV. Testing and Validation

To validate the developed system, successive tests were conducted on both hardware and software components, aiming to confirm not only their individual functionality but also their integration into a robust platform. The testing initially focused on verifying individual components such as the Pixy2 camera, the proximity sensor, the servomotor, and the drive motor, to ensure their correct configuration and initialization immediately after system startup.

Subsequently, the evaluation moved to the vector processing function, where various scenarios were analyzed, such as the detection of a single vector, the appearance of multiple fragmented vectors, and the filtering of irrelevant or redundant ones, ensuring that only the relevant vectors are used in the control algorithm.

The PID algorithm implemented for steering control was subjected to extensive testing to validate its behavior both under static conditions and during the vehicle's motion. Scenarios with manually fixed vectors were analyzed to confirm the system's predictable response, followed by dynamic tests that observed its reaction to changes in the relative position of the detected vectors. The PID algorithm demonstrated stable and accurate performance, being configured with optimal parameters to keep the robot on track.

In real tests, the vehicle ran on a specially prepared physical track. The autonomous robot followed the defined path and responded appropriately to obstacles or sudden changes in the road direction (Fig. 10). The observed behavior was consistent with the simulation results and demonstrated the proper functioning of all components. The tests conducted confirmed the stability and performance of the implemented system, validating both the individual hardware and software modules, as well as their full integration into a cohesive and functional platform.

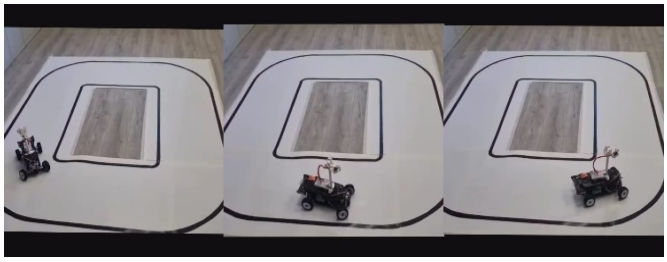

Figure 10. Real-world track navigation

To evaluate the stop-and-recovery mechanism, specific tests were conducted to analyze the system's response to the appearance of an obstacle or the temporary loss of data from the optical sensor. The results showed that the system detects obstacles and triggers the stop signal, and after the obstacle is removed, the vehicle automatically resumes autonomous movement, confirming the robustness of the implemented logic and the system's ability to regain control without external intervention.

Fig. 11 illustrates a moment from the testing of the stop mechanism, in which the autonomous vehicle detects an obstacle placed on the track and responds accordingly by coming to a complete stop before a collision occurs. The image is a frame captured from a video recording made during the experiment and highlights the system's behavior under real conditions, confirming the proper functioning of the proximity sensor and the implementation logic of the *STATE_STOP* state.

Validation in the virtual environment was carried out using the Gazebo simulator integrated with ROS 2. In this phase, the accuracy of the virtual sensor replicating the Pixy2 camera was tested, along with the synchronization of the data flow between the ROS nodes and the embedded

 Digital Object Identifier 10.32754/JMT.2026.1.08

application. The data generated in the simulator were sent to the PID controller on the MCU, and the resulting commands were returned to the virtual vehicle, allowing a complete analysis of the system's behavior in a controlled and safe environment.

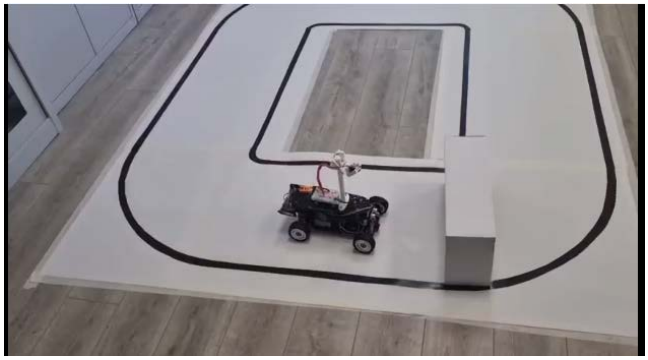

Figure 11. Image captured during testing of the stopping mechanism

The evolution of key parameters during a test in the simulator is illustrated in Fig. 12. The first graph shows the lateral error of the robot's position relative to the center of the track, reflecting how the system detects and corrects deviations, especially on curved sections. It can be observed that the error increases in the 180-degree turns, as seen in the track. In those sections, the control signal also varies more significantly, and the command sent to the servomotor (steer PWM) follows a repetitive pattern corresponding to left turns. The second graph shows the value of the command generated by the PID algorithm, which closely follows the evolution of the error and demonstrates the stability of the control response. The third graph displays the PWM signal applied to the servomotor for steering, confirming continuous and balanced adjustment of the direction according to the track being followed. As the PID control focuses exclusively on steering behavior, the final graph shows the constant speed of the vehicle.

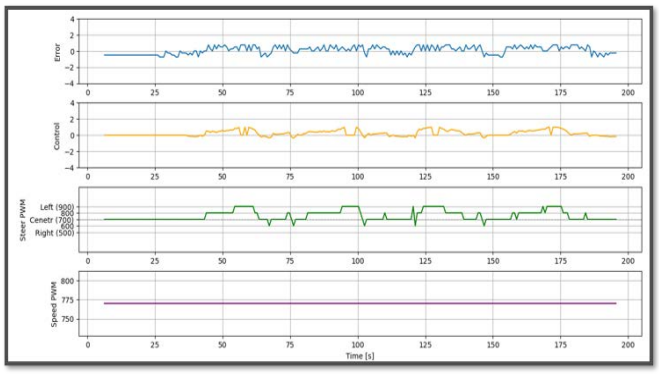

Figure 12. Evolution of the error, control output, and PWM signals

V. Conclusion

This work demonstrated the viability of an autonomous system capable of following a predefined path in both physical and virtual environments, using an embedded board integrated with a ROS 2 and Gazebo-based simulator. The development of this dual system involved both the configuration and validation of hardware components, as well as the implementation of modularized software architecture designed to ensure efficient communication between subsystems. The result is an autonomous vehicle that behaves in a similar and predictable manner in both environments.

Following the performed tests, the system proved capable of quickly and accurately processing visual data received from the Pixy2 camera or the virtual sensor in Gazebo, extracting relevant information in the form of vectors, and generating precise commands for steering control. Communication between the MCU and the ROS nodes via the UART interface ensured efficient synchronization of the data flow, which contributed to the overall effectiveness of system testing. Additionally, the implemented mechanisms for obstacle detection and the handling of temporary data loss functioned as expected, providing increased safety and robustness. All initially defined objectives were achieved, and the results obtained confirm that the proposed architecture is solid, well-suited to the requirements, and open to future improvements.

In this context, there are several clear development directions that could significantly enhance the performance of the current system.

One main limitation identified during practical tests is the lack of real feedback for the vehicle's speed. At present, speed control is performed exclusively by manually adjusting the PWM signal, without the possibility of dynamic adaptation based on driving conditions. Integrating a dedicated speed sensor would allow fine-tuning of acceleration, real-time speed adjustment based on the track, and the implementation of controlled and precise stops.

On the software side, improving the way the simulator reproduces the real behavior of the Pixy2 camera would open up important opportunities for testing complex scenarios, such as intersections. In the current implementation, the algorithm filters to find only two vectors, but the real camera can detect multiple vectors simultaneously. Adapting the simulator node to provide this extended type of information would allow the testing and validation of more advanced control algorithms, closer to the behavior of the physical system. Once the simulation is extended, it also becomes possible to test more efficient control methods than the currently used PID algorithm. Conducting comparative studies between multiple control methods would represent a valuable continuation of this project.

The experience gained highlighted the major advantages of the chosen approach. Parallel testing in both the simulated and real environments enabled early identification of issues, rapid and safe development of solutions, and a significant reduction of risks to the hardware components. At the same time, this process emphasized the importance of carefully calibrating the simulator parameters to accurately reproduce the real behavior of the vehicle, thereby ensuring consistency between the results obtained in the virtual environment and those from physical tests.

In conclusion, this paper provides a solid foundation for further development and research in the field of small-scale autonomous vehicles. The proposed system is stable, flexible, and extensible, and through the additional integration of specialized sensors, the diversification of simulated scenarios, and the experimentation with advanced control algorithms, it can become a valuable tool for both research and education in the field of robotics.

REFERENCES

[1] R. Welch, D. Limonadi, and R. Manning, "Systems engineering the Curiosity Rover: A retrospective," in *2013 8th International Conference on System of Systems Engineering*, 2013, pp. 70–75. doi:10.1109/SYSoSE.2013.6575245

[2] A. Faisal, T. Yigitcanlar, M. Kamruzzaman, and G. Currie, "Understanding autonomous vehicles: A systematic literature review on capability, impact, planning and policy," *J. Transp. Land Use*, vol. 12, no. 1, Jan. 2019. doi:10.5198/jtlu.2019.1405

[3] *K66 Sub-Family Reference Manual*, Rev. 2, Freescale Semiconductor Inc., 2015.

[4] "Wiki:V2:Overview [documentation]," *Pixycam.com.* [Online]. Available: https://docs.pixycam.com/wiki/doku.php?id=wiki:v2:overview.

[5] *HC-SR04 Ultrasonic Sensor: User's Manual*, V1.0, Cytron Technologies Sdn. Bhd., 2013.

[6] H. Unbehauen, Ed., *Control Systems, Robotics and Automation – Volume II: System Analysis and Control: Classical Approaches II*. Oxford, UK: EOLSS Publications, 2009.

[7] C. Knospe, "PID control," *IEEE Control Syst.*, vol. 26, no. 1, pp. 30–31, Feb. 2006. doi:10.1109/MCS.2006.1580151

[8] "Products – PixyCam," *Pixycam.com*. [Online]. Available: https://pixycam.com/products/.

[9] "Gazebo Harmonic — Gazebo harmonic documentation," *Gazebosim.org*. [Online]. Available: https://gazebosim.org/docs/harmonic/install/.

[10] "Understanding ROS 2 topics — ROS 2 Documentation: Crystal documentation," *Ros.org*. [Online]. Available: https://docs.ros.org/en/crystal/Tutorials/Topics/Understanding-ROS2-Topics.html.

 Digital Object Identifier 10.32754/JMT.2026.1.08