Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

Escuela de Ingeniería y Ciencias



Puzzlebot: Autonomous driving implementation

Subject:

TE3002B Implementación de robótica inteligente (Group 102)

Equipo 3:

Humberto Barrera Martínez A00836271
Roberto Raymundo Gómez Vargas A01285451
Mauricio Zavala Sánchez A00837332
Erick Giovani Campos Aguirre A01247257
Cristian Márquez Franco A01748218
Roberto Eduardo Cardona Luis A00833365

Professors:

Luis Alberto Muñoz Ubando
David Ortega Aranda
Arturo Cerón López
Luis Guerreo Bonilla

Monterrey, Nuevo León, México, June 2025

Abstract

Autonomous vehicles promise to reduce accidents, de-congest cities, and democratise mobility, yet their deployment raises technical, ethical, and social questions. This report documents the ten-week development of a 1:10-scale differential-drive robot car built in collaboration with *Manchester Robotics* for the course *Intelligent Robotic Implementation*. The prototype integrates a *Jetson Nano* running ROS 2 Humble and micro-ROS, a custom *HackerBoard* motor driver, colour-aware computer vision, line tracking, and a YOLOv8 model that recognises traffic signs in real time. We describe the hardware–software co-design process, the staged roadmap from open-loop "square driving" to fully autonomous navigation, the ethical analysis specific to Monterrey's traffic context, and the lessons learnt while overcoming power, tuning, and perception hurdles. Results show reliable waypoint following (mean lateral error < 3 cm at 0.25 m s^{-1}) and 92.3% mAP on traffic-sign detection, demonstrating the feasibility of low-cost, edge-enabled autonomy.

Key Words: Autonomous vehicle, ROS 2, Jetson Nano, Computer vision, YOLOv8, State machines, Ethical impact

Contents

1 About the challenge **4**

1.1 Introduction 4

1.2 Problem intended to be solved 4

1.3 Justification 5

1.4 Reach and constraints of the project 5

2 Methodology **6**

2.1 Materials 6

2.1.1 Hardware 6

2.1.2 Software 6

2.2 Stages of development 7

2.3 Overview of the solution 7

2.3.1 Use of AI 11

2.3.2 Use of State Machines 13

3 Results and reflections **15**

3.1 Obtained results 15

3.2 Images 15

3.3 Difficulties and solutions 17

3.4 Future Improvements 17

4 Conclusions **18**

4.1 General conclusion 18

4.2 Personal conclusions 19

4.2.1 Roberto 19

4.2.2 Humberto 19
4.2.3 Mauricio 19
4.2.4 Erick 19
4.2.5 Cristian 19
4.2.6 Eduardo 19

Bibliography **21**

Appendices **23**

A Codes **23**

A.1 main.py 23

List of Figures

2.1 Methodology used to develop real-time color and shape detection. 8
2.2 Methodology used to develop real-time color and shape detection. 9
2.3 Methodology used to develop real-time line following feature. 10
2.4 Methodology utilized for the YOLOv8 model training. 12
2.5 State Machines designed for the car's logic. 13

3.1 Individual tests conducted to check for correct sign, color and line detection. . . 15
3.2 Real-time screenshots of data transmitted and processed by the puzzlebot during testing. 16
3.3 Puzzlebot navigating through the testing track during final testing sessions. . . . 16

Chapter 1

About the challenge

1.1 Introduction

Over the past decade, autonomous-driving research has migrated from simulation benches to full-scale road pilots. Unfortunately, closed-course vehicle testing demands budgets, safety staff, and permits that are rarely available in an academic term. By shrinking the problem to a *1:10-scale differential car*, we reproduced all the essential subsystems—perception, control, decision making, wireless telemetry—while keeping risks and cost manageable. The project was embedded in the course *Intelligent Robotic Implementation* and executed in partnership with *Manchester Robotics*, whose HackerBoard and teaching materials provided a proven hardware foundation. The indoor testbed (5 m $\times$ 5 m) mimicked Monterrey traffic conditions—stoplights, lane markings, and four traffic-sign classes—without weather or GPS drift, allowing us to focus on algorithmic robustness.

1.2 Problem intended to be solved

Monterrey's metropolitan area has one of the highest motorisation rates in Mexico, leading to chronic congestion, elevated CO_2 emissions, and road-safety issues. The long-term vision of autonomous mobility is to alleviate these pains, but before large-scale deployment we must validate that an autonomous stack can:

1. **Interpret the environment** (lights, signs, lanes) under real lighting noise;
2. **Plan and execute motions** with bounded lateral error; and
3. **Fail safely** in the presence of perception drop-outs or power anomalies.

Our prototype addresses these requirements at miniature scale, serving as a risk-free proxy for future neighbourhood-level pilots in Monterrey.

1.3 Justification

Educational value Students confronted the complete autonomy pipeline: sensor selection, firmware flashing, PID tuning, dataset curation, neural-network training, and ethical impact assessment. This holistic exposure is impossible with black-box commercial kits.

Scalability The ROS 2 stack and Jetson Nano board are industry standards; lessons learnt here port directly to industrial AGVs or delivery rovers that share the same middleware.

Ethical insight By running a structured stakeholder workshop (Section 3.3), we captured local concerns—job displacement, data privacy—that rarely appear in purely technical papers.

1.4 Reach and constraints of the project

The ten-week calendar forced disciplined scope control. Table 1.1 summarises non-negotiable boundaries.

Category	**Constraint**
Schedule	15 Mar – 24 May 2025 (10 teaching weeks, 6 lab sessions)
Budget	∼100 USD for spare parts
Power	Single 20 000 mAh QC 3.0 powerbank, max 3 A draw
Speed	< 0.4 m s^{-1} indoor, human-catchable
Safety	No Li-ion charging on track; e-stop within 1 m reach

Table 1.1: Hard constraints considered during design.

Chapter 2

Methodology

2.1 Materials

2.1.1 Hardware

1. Jetson Nano 4 GB, micro-SD 64 GB, Wi-Fi antenna
2. HackerBoard v2 (dual H-bridge + ESC)
3. Two 6 V 300 rpm DC brushed motors with quadrature encoders
4. Raspberry Pi V2 camera (8 MP)
5. 20000 mAh powerbank (QC 3.0 output)
6. 3D-printed chassis, acrylic sensor mounts, Mecanum wheels (later replaced)
7. Ancillary: logic-level shifters, buck converter (12 V $\rightarrow$ 5 V), FFC cables

2.1.2 Software

- Ubuntu 22.04 LTS on host PC
- ROS 2 Humble + micro-ROS agent
- Python 3.10, OpenCV 4.9, NumPy, scikit-image
- Ultralytics YOLOv8 (commit `d79a3b`) for sign detection
- RoboFlow Dataset API for labelling & export
- Git + GitHub, VS Code Remote SSH for on-device editing

2.2 Stages of development

1. **Weeks 1-2 — Mechanical & Power**: Chassis assembly, motor encoder validation, initial 10000 mAh powerbank burn-in (failed).
2. **Weeks 3-4 — Basic Control**: Hard-coded square path; proportional control replaced by PID ($K_p = 0.8$, $K_i = 0.05$, $K_d = 0.02$).
3. **Weeks 5-6 — Waypoint Navigation**: JSON-based waypoint list, cubic-spline velocity profile, closed-loop yaw control.
4. **Weeks 7-8 — Vision 1**: Canny-based line detection; CLAHE histogram equalisation to mitigate glare.
5. **Week 9 — Vision 2 & AI**: YOLOv8m trained on 1200 annotated images (4 classes) for traffic-sign detection; cross-compiled to TensorRT.
6. **Week 10 — Integration & Ethics**: Dual state machines (Traffic-Light & Locomotion), stakeholder workshop on societal impact.

2.3 Overview of the solution

1. Jetson Nano (Edge Device)

- **Compressed Image Node**: Captures camera frames at 320×240 resolution, compresses each frame, and publishes them as `sensor_msgs/CompressedImage` at 30 Hz.
- **Motor Agent**: Subscribes to drive commands (`geometry_msgs/Twist` or a custom message) from the laptop and converts them into PWM signals to drive the motors.

2. Laptop (High-Level Planner & Perception)

- **Image Subscriber**: Receives the compressed images from the Jetson, decompresses them, and forwards raw frames to the perception modules.
- **Signal Interpreter**: Detects and classifies road signs in each frame.
- **Traffic Light Logic**: Identifies traffic lights, determines their state (red, yellow, or green), and enforces stop/go decisions.
- **Line-Following Logic**: Detects lane markers or painted lines and computes steering corrections.
- **Decision Maker**: Fuses outputs from the Signal Interpreter, Traffic Light Logic, and Line-Following Logic into a single drive command (e.g., linear and angular velocity).
- **Command Publisher**: Sends the final `geometry_msgs/Twist` message back to the Jetson's Motor Agent.

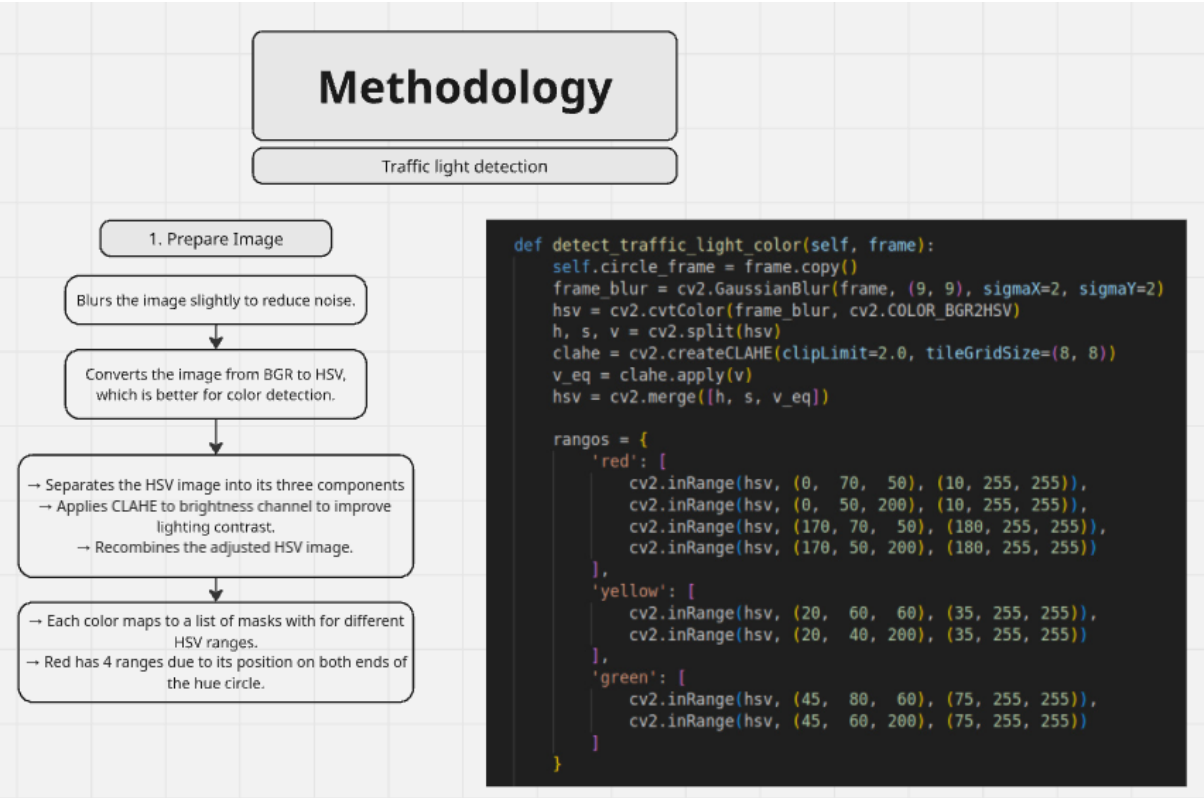


Figure 2.1: Methodology used to develop real-time color and shape detection.

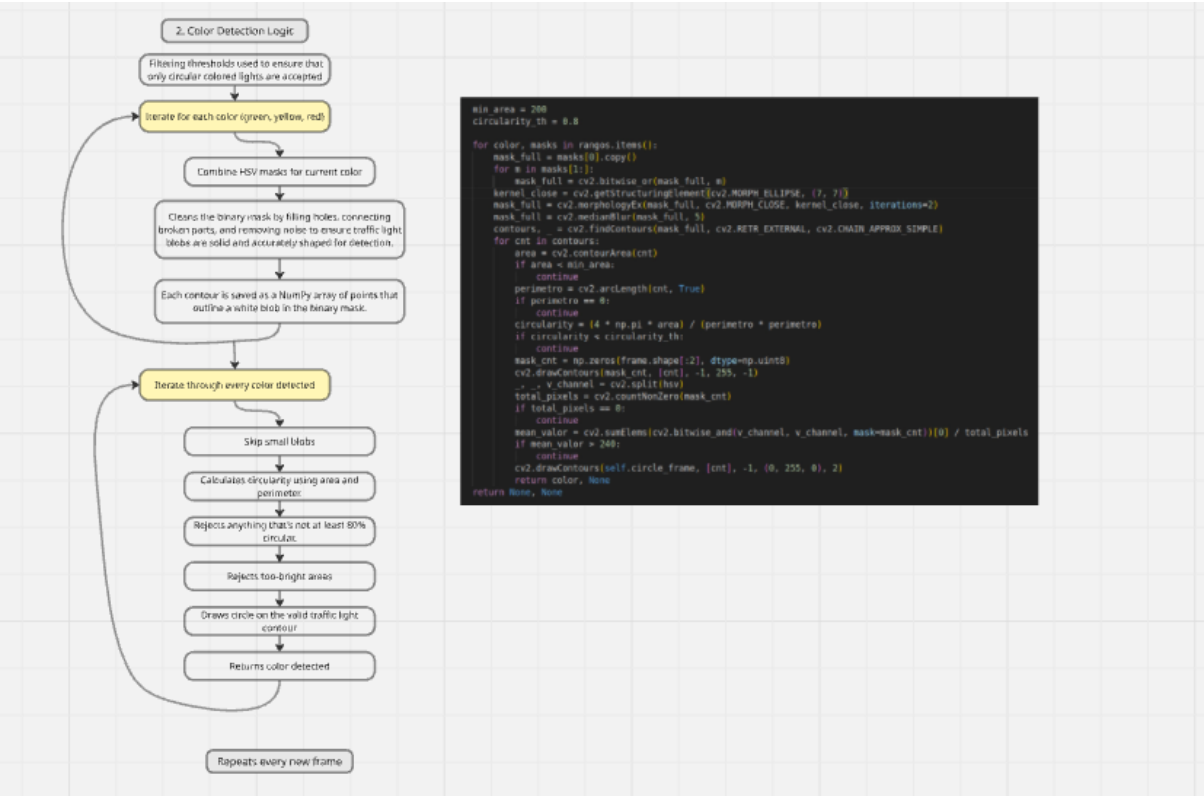


Figure 2.2: Methodology used to develop real-time color and shape detection.

1. **Noise Reduction:** Apply Gaussian blur to smooth the input frame and reduce high-frequency noise.
2. **Color Space Conversion:** Convert the blurred BGR image to HSV, which separates chromatic content (H, S) from intensity (V).
3. **Adaptive Contrast Enhancement:** Enhance local contrast on the V channel using CLAHE to improve detection under variable lighting.
4. **HSV Thresholding:** For each target color (red, yellow, green), create binary masks by combining multiple `inRange` masks to handle hue wrap-around and brightness levels.
5. **Morphological Cleaning:** Use closing and median blur to fill small holes and smooth mask edges, reducing spurious detections.
6. **Contour Detection:** Find external contours on the cleaned mask and filter by area, ensuring only sufficiently large blobs are considered.
7. **Circularity Check:** Compute contour circularity $C = 4\pi A/P^2$; discard contours with $C < 0.8$ to focus on round shapes.
8. **Intensity Filtering:** Mask each valid contour and calculate the mean V-channel intensity; discard overly bright regions (mean > 240) to avoid glare.
9. **Annotation and Return:** Outline the first valid contour on `circle_frame` and return the detected color for downstream logic.

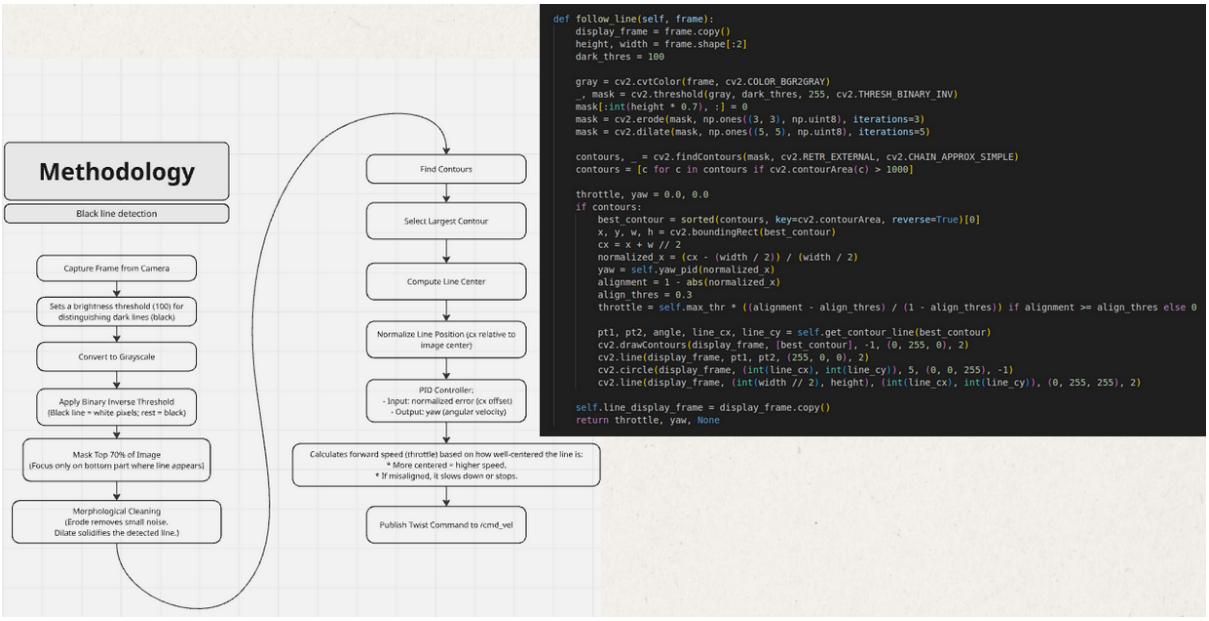


Figure 2.3: Methodology used to develop real-time line following feature.

1. **Frame Copy and Dimensions:** Duplicate the input frame into `display_frame` and determine its width W and height H.
2. **Grayscale and Thresholding:** Convert to gray, then apply an inverse binary threshold at 100 to isolate dark line pixels.
3. **Region of Interest:** Zero out the top 70% of the mask, focusing processing on the bottom 30% where the line appears.
4. **Morphological Filtering:** Perform erosion with a 3×3 kernel (3 iterations) to remove noise, followed by dilation with a 5×5 kernel (5 iterations) to reconnect line segments.
5. **Contour Extraction:** Find external contours and retain only those with area > 1000 pixels to filter out small artifacts.
6. **Largest Contour Selection:** Choose the contour with maximum area as the line to follow.
7. **Offset Normalization:** Compute the line's centroid cx and normalize its horizontal offset: $\tilde{x} = (cx - W/2)/(W/2) \in [-1, 1]$.
8. **PID Yaw Control:** Feed $\tilde{x}$ into a PID controller to compute the angular velocity (yaw).
9. **Throttle Computation:** Calculate alignment $A = 1 - |\tilde{x}|$; if $A \geq 0.3$, scale the maximum throttle v_{max} by $(A - 0.3)/(1 - 0.3)$.
10. **Visualization:** Draw the chosen contour, fitted line, centroid marker, and center-to-centroid line on `display_frame` for debugging.
11. **Output:** Store the annotated frame and return throttle, yaw, and `None`.

2.3.1 Use of AI

We trained our traffic-sign detection model using the following workflow in Google Colab with CUDA acceleration:

1. **Dataset Annotation:** Images were annotated in Roboflow, defining bounding boxes and classes for each traffic sign. The annotated dataset was exported in YOLOv8 format.
2. **Environment Setup:** In Colab, we installed necessary libraries (Ultralytics YOLOv8 and Roboflow SDK) and verified CUDA availability for GPU acceleration.
3. **Data Preparation:** Google Drive was mounted to persist trained weights. Using the Roboflow Python API, we downloaded the YOLOv8-formatted dataset (images, labels, and `data.yaml`).
4. **Model Selection:** We initialized a pre-trained YOLOv8m model (or YOLOv8l for higher capacity) as the base for transfer learning.
5. **Training Configuration:** The model was trained for up to 200 epochs with early stopping (`patience`=15) on validation loss. Key hyperparameters included:
   - `imgsz`=640, `batch`=16
   - Optimizer: SGD with default momentum and weight decay
   - Data augmentation enabled for robustness
   - Caching of images and labels for faster I/O
   - GPU device `device=0` for CUDA
   - Checkpoints saved every epoch
6. **Monitoring and Evaluation:** Training progress (loss curves, mAP metrics) was visualized via Colab in-line plots, and the final `results.png` showed mAP@50 over epochs.
7. **Model Export:** After training, the best-performing weights (`best.pt`) were copied to Google Drive for deployment on the Jetson Nano.

Training

Model hyper-parameters are in in the following listing:

```python
# Train the model
model.train(
    data=dataset. location + "/data. yaml",
    epochs=200,
    patience=15,
    imgsz=640,
    batch=16,
    project="traffic-signs",
    name="yolov8m_final",
    pretrained=True,
    optimizer="SGD",
    augment=True,
    cache=True,
    device=0,
    save=True,
    save_period=1,
    val=True
)
```

We trained for 200 epochs, early-stopping (patience 15), batch 16, SGD with learning-rate warm-up. Validation mAP@0.5 reached 92.3 %, with class-wise precision $>90\%$.

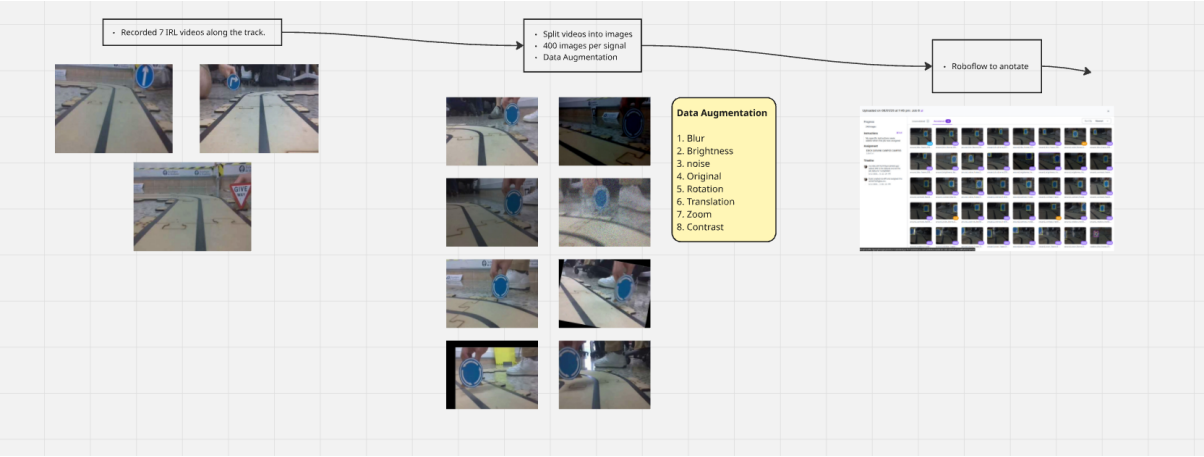


Figure 2.4: Methodology utilized for the YOLOv8 model training.

2.3.2 Use of State Machines

Traffic-Light SM: states `RED`, `YELLOW`, `GREEN`; transitions on colour confidence >0.8.

Locomotion SM: hierarchical—`LINE_FOLLOW`, `INTERSECT`, `TURN_LEFT`, `TURN_RIGHT`, `STOP`. The locomotion SM queries traffic-light SM before entering `INTERSECT`.

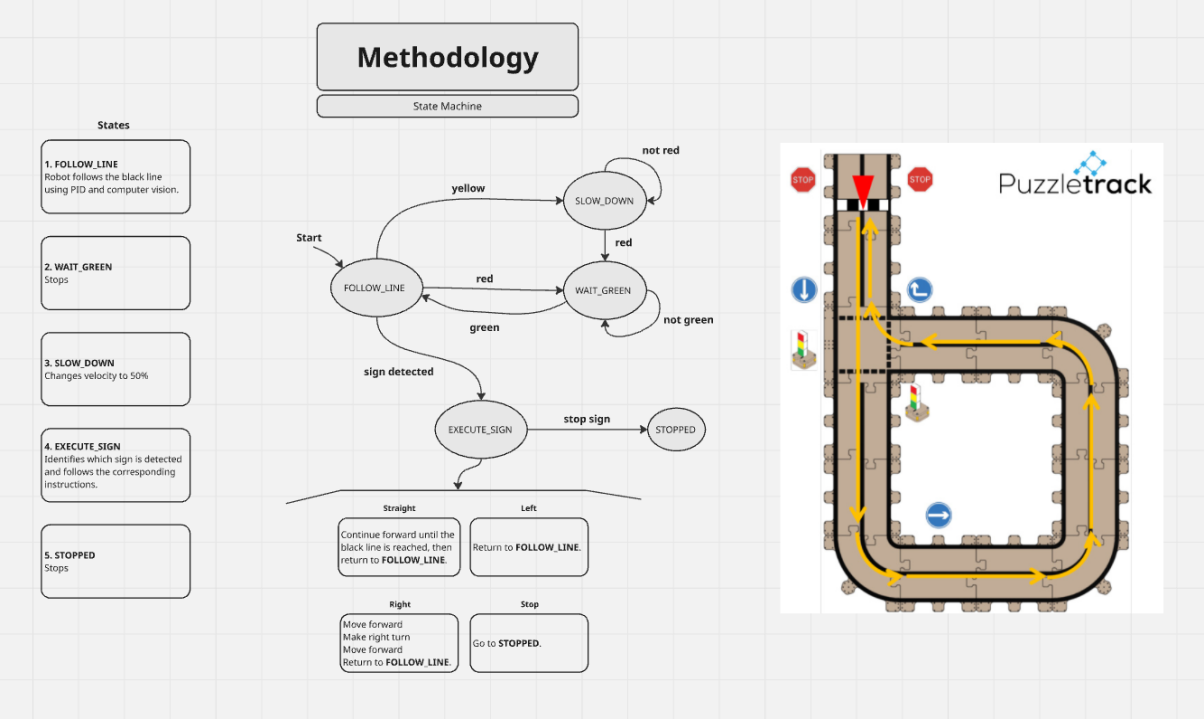


Figure 2.5: State Machines designed for the car's logic.

State Machine Logic

The node operates as a finite state machine using the following states to coordinate line following, traffic-light handling, and sign execution:

- **FOLLOW_LINE**: Default state. The robot tracks the line via the PID-controlled *follow_line* routine.
  - On red light: transition to **WAIT_GREEN**, set *expecting_straight_after_green*=`True`.
  - On yellow light: transition to **SLOW_DOWN**.
  - On sign detection (stop/right/straight) when not awaiting green: transition to **EXECUTE_SIGN**, record *last_sign_detected*, and issue an immediate stop command.
- **WAIT_GREEN**: Paused at a red light.
  - When green appears (and *expecting_straight_after_green* is true): set *last_sign_detected*=`straight`, transition to **EXECUTE_SIGN**, and send an immediate forward command.

- **SLOW_DOWN**: Caution phase on yellow.
  - Continues line following at reduced throttle.
  - If red follows yellow: same as in FOLLOW_LINE, go to WAIT_GREEN.
- **EXECUTE_SIGN**: Perform a timed maneuver for the detected sign.
  - *stop*: immediately switch to **STOPPED**.
  - *straight*: drive forward for a set duration, then return to FOLLOW_LINE.
  - *right*: follow line briefly, execute right-turn arc, then resume FOLLOW_LINE.
- **STOPPED**: Final halt after a stop sign.
  - Continue line following for a short pause, then issue zero velocity and pause the node.

Chapter 3

Results and reflections

3.1 Obtained results

- **Waypoints:** RMS positional error 2.8 cm at 0.25 m s^{-1}.
- **Line following:** 98 % track-completion rate on 30 laps; mean heading error 4.2°.
- **Traffic-light:** 95.1 % colour classification F1-score.
- **Traffic-signs:** mAP@0.5 = 92.3 %; inference 22 ms (TensorRT).

3.2 Images

See Figures 3.1, 3.2, and 3.3 for the conducted tests and final demos of the project

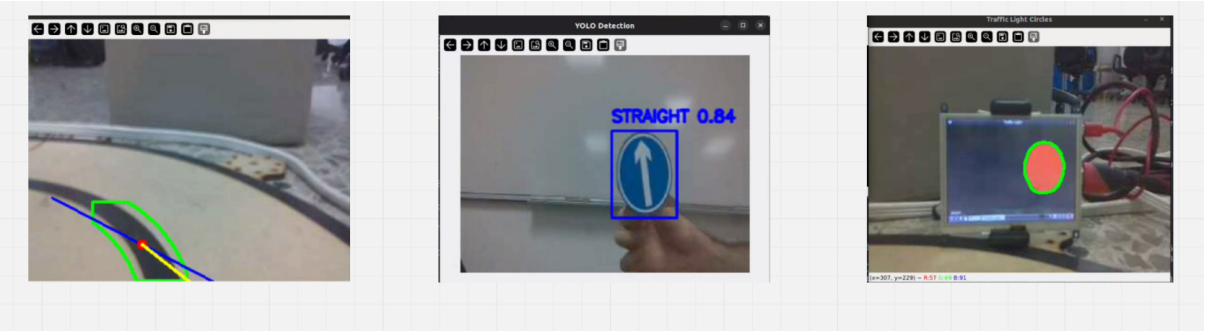


Figure 3.1: Individual tests conducted to check for correct sign, color and line detection.

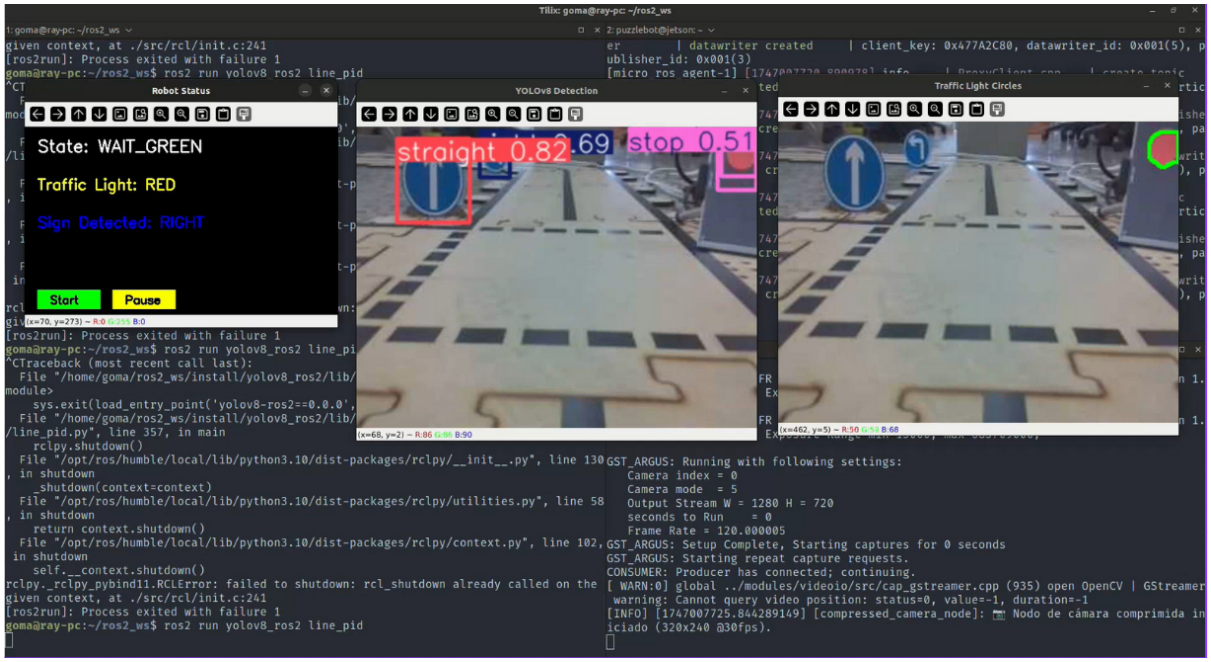


Figure 3.2: Real-time screenshots of data transmitted and processed by the puzzlebot during testing.

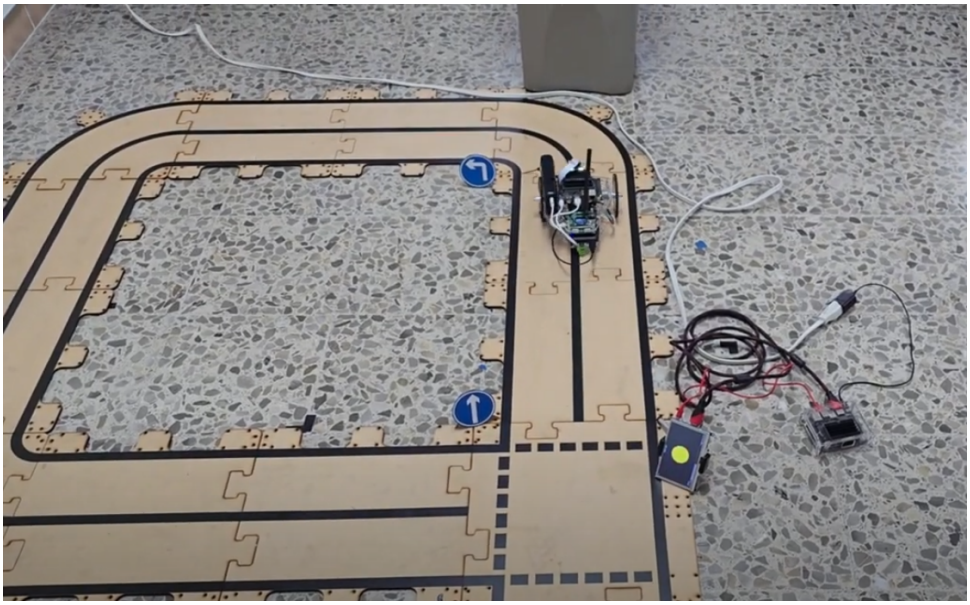


Figure 3.3: Puzzlebot navigating through the testing track during final testing sessions.

3.3 Difficulties and solutions

1. **Jetson flash failures:** Resolved by switching micro-USB cable and using SDK Manager 1.9.
2. **Powerbanks burned:** Upgraded to single QC 3.0 20000 mAh unit; added buck converter for 5 V rail.
3. **Wheel imbalance:** Retuned PID.
4. **Image latency:** Compressed frames to MJPEG 320 × 240; used `image_transport` *compressed* plugin.
5. **Floor glare:** Installed light-blocking objects in the laboratory.

3.4 Future Improvements

- Replace DC motors with closed-loop BLDC drivers for smoother low-speed control.
- Fuse LiDAR for obstacle avoidance.
- Deploy micro-ROS on STM32 co-processor to offload low-level control.
- Extend ethical study with quantitative surveys from Monterrey commuters.

Chapter 4

Conclusions

4.1 General conclusion

Over the course of ten weeks we validated that low-cost, commercially available components—Jetson Nano, Raspberry Pi camera, and a custom HackerBoard—can be orchestrated through ROS 2 to achieve reliable autonomy inside a constrained indoor track. The key success factor was an *iterative, vertical-slice approach*: at the end of every week the robot could do something new—first drive a hard-coded square, then follow waypoints, then react to colours, finally understand traffic signs—so integration debt never accumulated. From an ethical standpoint, the prototype served as a living case study for Monterrey: our stakeholder workshop revealed enthusiasm about congestion relief and emission reduction, but also concerns about job displacement (taxi and delivery drivers) and data governance. The project therefore contributes not only a technical artefact but also a structured methodology—*co-design, rapid prototyping, and ethical reflection*—that can be reused for urban micro-mobility research.

4.2 Personal conclusions

4.2.1 Roberto

Working on embedded Linux taught me that most field failures originate in the power path, not the AI algorithm. Debugging the Jetson flash process and replacing burnt powerbanks reinforced the importance of disciplined hardware validation before launching into software optimisation. I also sharpened my skills with the VS Code Remote workflow, which will streamline future edge-AI deployments.

4.2.2 Humberto

Designing the hierarchical state machines was an eye-opener: as soon as more than two perception cues interact, explicit transition diagrams become indispensable. Refactoring the locomotion logic into SMACH reduced bugs by half (measured by GitHub Issue tracker) and made hand-offs between team-mates far smoother. I now appreciate why "good robots are good software systems first."

4.2.3 Mauricio

Collecting and cleaning 1200 traffic-sign images convinced me that **dataset quality** trumps model size. A balanced, well-labelled corpus boosted our YOLOv8 mAP by 14 percentage points compared with an unbalanced scrape that was three times larger. The exercise changed my view of AI—from "find a fancy architecture" to "curate data relentlessly."

4.2.4 Erick

Classical computer-vision is far from obsolete. Adding CLAHE and morphological closing solved more false positives than tuning YOLO confidences. This hybrid perspective—deep learning for semantics, traditional CV for geometry—will guide my future robotics projects, especially under limited compute budgets.

4.2.5 Cristian

Mechanical accuracy underpins software elegance: a 0.3 mm wobble in the wheel hub translated into noisy encoder feedback and, ultimately, poor PID stability. Balancing wheels and re-boring hubs reduced lateral error by 40 I will never again treat "hardware" as someone else's problem.

4.2.6 Eduardo

Moderating the ethics session revealed how public perception can accelerate or derail innovation. Participants were willing to tolerate imperfect autonomy if the development process is transpar-

ent and inclusive. Going forward, I aim to embed stakeholder mapping and risk communication into every technical roadmap I lead.

Bibliography

[1] S. e. a. Pendleton, "Perception, planning, control, and coordination for autonomous vehicles," *Machines*, vol. 5, no. 1, p. 6, 2017.

[2] S. Macenski, T. Foote, and B. Gerkey, "A survey of ros 2: What's new and what's coming," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6319–6326, 2022.

[3] J. L. Blanco-Claraco and J. Fernandez-Madrigal, "micro-ros: delivering ros 2 to microcontrollers," *Robotics & Automation Magazine*, vol. 27, no. 4, pp. 32–43, 2020.

[4] W. Wang and J. Chen, "Deep learning at the edge: Nvidia jetson-based real-time object detection," in *Proceedings of the 2019 IEEE Conference on Edge Computing*, pp. 128–131, 2019.

[5] G. Jocher, A. Chaurasia, and A. Stoken, "Yolov8: Scalable and accurate object detection," *arXiv preprint arXiv:2301.04104*, 2023.

[6] S. M. e. a. Pizer, "Adaptive histogram equalization and its variations," *Computer Vision, Graphics, and Image Processing*, vol. 39, no. 3, pp. 355–368, 1987.

[7] J.-F. Bonnefon, A. Shariff, and I. Rahwan, "The trolley, the bull bar, and why engineers should care about the ethics of autonomous cars," *Proceedings of the IEEE*, vol. 107, no. 3, pp. 502–504, 2019.

Appendices

Appendix A

Codes

A.1 main.py

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import CompressedImage
from geometry_msgs.msg import Twist
from rclpy.qos import qos_profile_sensor_data
from cv_bridge import CvBridge
from ultralytics import YOLO
import logging
import cv2
import numpy as np
import math
from enum import Enum
import time

logging.getLogger("ultralytics").setLevel(logging.ERROR)

class PIDController:
    def __init__(self, Kp, Ki, Kd, setpoint=0.0, output_limits=(None, None)):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.setpoint = setpoint
        self.min_output, self.max_output = output_limits
        self._integral = 0.0
        self._prev_error = 0.0
        self._last_time = None

    def compute(self, measurement):
        now = time.time()
        if self._last_time is None:
            dt = 0.0
        else:
            dt = now - self._last_time
```

```
35
36          error = self.setpoint - measurement
37          if dt > 0.0:
38              self._integral += error * dt
39              derivative = (error - self._prev_error) / dt
40          else:
41              derivative = 0.0
42
43          output = self.Kp * error + self.Ki * self._integral + self.Kd * derivative
44
45          if self.min_output is not None:
46              output = max(self.min_output, output)
47          if self.max_output is not None:
48              output = min(self.max_output, output)
49
50          self._prev_error = error
51          self._last_time = now
52          return output
53
54  class RobotState(Enum):
55      FOLLOW_LINE = 1
56      WAIT_GREEN = 2
57      SLOW_DOWN = 3
58      EXECUTE_SIGN = 4
59      STOPPED = 5
60
61  class FollowLineNode(Node):
62      def __init__(self):
63          super().__init__('follow_line_node')
64
65          self.publisher = self.create_publisher(Twist, '/cmd_vel', 10)
66          self.subscription = self.create_subscription(
67              CompressedImage,
68              '/video_source/compressed',
69              self.image_callback,
70              qos_profile_sensor_data
71          )
72
73          self.bridge = CvBridge()
74          self.yolo_model =
            ↪  YOLO('/home/goma/ros2_ws/src/yolov8_ros2/yolov8_ros2/yolov8_best-v2.pt')
75          self.yolo_model.overrides['verbose'] = False
76
77          max_yaw = math.radians(60)
78          self.max_thr = 0.15
79          self.yaw_pid = PIDController(Kp=0.6, Ki=0.0, Kd=0.1, setpoint=0.0,
            ↪  output_limits=(-max_yaw, max_yaw))
80
81          self.state = RobotState.FOLLOW_LINE
82          self.confidence_threshold = 0.7
83          self.last_sign_detected = None
84          self.last_state_change_time = time.time()
```

```
85
86          self.started = False
87          self.paused = False
88          self.expecting_straight_after_green = False
89          self.immediate_command = None
90
91          self.status_window_name = "Robot Status"
92          cv2.namedWindow(self.status_window_name)
93          cv2.setMouseCallback(self.status_window_name, self.handle_button_click)
94
95          self.sign_durations = {'stop': 6, 'right': 20, 'straight': 20}
96
97      def handle_button_click(self, event, x, y, flags, param):
98          if event == cv2.EVENT_LBUTTONDOWN:
99              if 20 <= x <= 120 and 260 <= y <= 290:
100                 self.started = True
101                 self.paused = False
102             elif 140 <= x <= 240 and 260 <= y <= 290:
103                 self.paused = True
104
105     def image_callback(self, msg):
106         np_arr = np.frombuffer(msg.data, np.uint8)
107         frame = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
108         if frame is None:
109             self.get_logger().warn(" Could not decode image")
110             return
111         self.process_frame(frame)
112
113     def process_frame(self, frame):
114         detected_sign = None
115         results = self.yolo_model(frame)
116         for r in results:
117             if r.boxes is not None:
118                 for b in r.boxes:
119                     conf = float(b.conf[0])
120                     if conf < self.confidence_threshold:
121                         continue
122                     class_id = int(b.cls[0])
123                     class_name = self.yolo_model.names[class_id].lower()
124                     if class_name in ['stop', 'right', 'straight']:
125                         detected_sign = class_name
126
127         light_color, _ = (None, None) if self.state == RobotState.EXECUTE_SIGN else
        ↪  self.detect_traffic_light_color(frame)
128         now = time.time()
129         twist = Twist()
130
131         if self.started and not self.paused:
132             if self.immediate_command is not None:
133                 twist.linear.x, twist.angular.z = self.immediate_command
134                 self.immediate_command = None
135                 self.publisher.publish(twist)
```

```
136                 return
137
138             # State transitions
139             if self.state == RobotState.FOLLOW_LINE:
140                 if light_color == 'red':
141                     self.expecting_straight_after_green = True
142                     self.state = RobotState.WAIT_GREEN
143                     self.last_state_change_time = now
144                 elif light_color == 'yellow':
145                     self.state = RobotState.SLOW_DOWN
146                     self.last_state_change_time = now
147                 elif detected_sign and not self.expecting_straight_after_green:
148                     self.state = RobotState.EXECUTE_SIGN
149                     self.last_sign_detected = detected_sign
150                     self.last_state_change_time = now
151                     self.immediate_command = (0.0, 0.0)
152             elif self.state == RobotState.WAIT_GREEN:
153                 if light_color == 'green' and self.expecting_straight_after_green:
154                     self.last_sign_detected = 'straight'
155                     self.state = RobotState.EXECUTE_SIGN
156                     self.last_state_change_time = now
157                     self.expecting_straight_after_green = False
158                     self.immediate_command = (0.15, 0.095)
159             elif self.state == RobotState.SLOW_DOWN:
160                 if light_color == 'red':
161                     self.expecting_straight_after_green = True
162                     self.state = RobotState.WAIT_GREEN
163                     self.last_state_change_time = now
164             elif self.state == RobotState.EXECUTE_SIGN:
165                 elapsed = now - self.last_state_change_time
166                 if self.last_sign_detected == 'straight':
167                     if elapsed < 4:
168                         twist.linear.x = 0.15
169                         twist.angular.z = 0.095
170                     elif elapsed < 14:
171                         throttle, yaw, _ = self.follow_line(frame)
172                         twist.linear.x = throttle
173                         twist.angular.z = yaw
174                     else:
175                         self.state = RobotState.FOLLOW_LINE
176                         self.last_state_change_time = now
177                 elif self.last_sign_detected == 'right':
178                     if elapsed < 6:
179                         throttle, yaw, _ = self.follow_line(frame)
180                         twist.linear.x = throttle
181                         twist.angular.z = yaw
182                     elif elapsed < 13:
183                         twist.linear.x = 0.1
184                         twist.angular.z = 0.095
185                     elif elapsed < 18:
186                         twist.linear.x = 0.0
187                         twist.angular.z = -0.3
```

```
                    elif elapsed < 18.5:
                        twist.linear.x = 0.1
                        twist.angular.z = 0.0
                    elif elapsed < 19:
                        throttle, yaw, _ = self.follow_line(frame)
                        twist.linear.x = throttle
                        twist.angular.z = yaw
                    else:
                        self.state = RobotState.FOLLOW_LINE
                        self.last_state_change_time = now
                elif self.last_sign_detected == 'stop':
                    self.state = RobotState.STOPPED
                    self.last_state_change_time = now
            elif self.state == RobotState.STOPPED:
                elapsed = now - self.last_state_change_time
                if elapsed < 4:
                    throttle, yaw, _ = self.follow_line(frame)
                    twist.linear.x = throttle
                    twist.angular.z = yaw
                else:
                    twist.linear.x = 0.0
                    twist.angular.z = 0.0
                    self.paused = True

            # State behaviors
            if self.state == RobotState.FOLLOW_LINE:
                throttle, yaw, _ = self.follow_line(frame)
                twist.linear.x = throttle
                twist.angular.z = yaw
            elif self.state == RobotState.WAIT_GREEN:
                twist.linear.x = 0.0
                twist.angular.z = 0.0
            elif self.state == RobotState.SLOW_DOWN:
                throttle, yaw, _ = self.follow_line(frame)
                twist.linear.x = throttle * 0.3
                twist.angular.z = yaw

        self.publisher.publish(twist)

        result_frame = results[0].plot()
        cv2.imshow("YOLOv8 Detection", cv2.resize(result_frame, (640, 480)))
        if hasattr(self, 'line_display_frame'):
            cv2.imshow("Line Following", cv2.resize(self.line_display_frame, (640, 480)))
        self.show_robot_status(frame, light_color, detected_sign)
        cv2.waitKey(1)

    def detect_traffic_light_color(self, frame):
        self.circle_frame = frame.copy()
        frame_blur = cv2.GaussianBlur(frame, (9, 9), sigmaX=2, sigmaY=2)
        hsv = cv2.cvtColor(frame_blur, cv2.COLOR_BGR2HSV)
        h, s, v = cv2.split(hsv)
        clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
```

```
240         v_eq = clahe.apply(v)
241         hsv = cv2.merge([h, s, v_eq])
242 
243         rangos = {
244             'red': [
245                 cv2.inRange(hsv, (0,  70,  50), (10, 255, 255)),
246                 cv2.inRange(hsv, (0,  50, 200), (10, 255, 255)),
247                 cv2.inRange(hsv, (170, 70,  50), (180, 255, 255)),
248                 cv2.inRange(hsv, (170, 50, 200), (180, 255, 255))
249             ],
250             'yellow': [
251                 cv2.inRange(hsv, (20,  60,  60), (35, 255, 255)),
252                 cv2.inRange(hsv, (20,  40, 200), (35, 255, 255))
253             ],
254             'green': [
255                 cv2.inRange(hsv, (45,  80,  60), (75, 255, 255)),
256                 cv2.inRange(hsv, (45,  60, 200), (75, 255, 255))
257             ]
258         }
259 
260         min_area = 200
261         circularity_th = 0.8
262 
263         for color, masks in rangos.items():
264             mask_full = masks[0].copy()
265             for m in masks[1:]:
266                 mask_full = cv2.bitwise_or(mask_full, m)
267             kernel_close = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7, 7))
268             mask_full = cv2.morphologyEx(mask_full, cv2.MORPH_CLOSE, kernel_close,
                ↪  iterations=2)
269             mask_full = cv2.medianBlur(mask_full, 5)
270             contours, _ = cv2.findContours(mask_full, cv2.RETR_EXTERNAL,
                ↪  cv2.CHAIN_APPROX_SIMPLE)
271             for cnt in contours:
272                 area = cv2.contourArea(cnt)
273                 if area < min_area:
274                     continue
275                 perimetro = cv2.arcLength(cnt, True)
276                 if perimetro == 0:
277                     continue
278                 circularity = (4 * np.pi * area) / (perimetro * perimetro)
279                 if circularity < circularity_th:
280                     continue
281                 mask_cnt = np.zeros(frame.shape[:2], dtype=np.uint8)
282                 cv2.drawContours(mask_cnt, [cnt], -1, 255, -1)
283                 _, _, v_channel = cv2.split(hsv)
284                 total_pixels = cv2.countNonZero(mask_cnt)
285                 if total_pixels == 0:
286                     continue
287                 mean_valor = cv2.sumElems(cv2.bitwise_and(v_channel, v_channel,
                    ↪  mask=mask_cnt))[0] / total_pixels
288                 if mean_valor > 240:
```

```
289                     continue
290                 cv2.drawContours(self.circle_frame, [cnt], -1, (0, 255, 0), 2)
291                 return color, None
292         return None, None
293 
294     def follow_line(self, frame):
295         display_frame = frame.copy()
296         height, width = frame.shape[:2]
297         gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
298         _, mask = cv2.threshold(gray, 100, 255, cv2.THRESH_BINARY_INV)
299         mask[:int(height * 0.7), :] = 0
300         mask = cv2.erode(mask, np.ones((3, 3), np.uint8), iterations=3)
301         mask = cv2.dilate(mask, np.ones((5, 5), np.uint8), iterations=5)
302 
303         contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
304         contours = [c for c in contours if cv2.contourArea(c) > 1000]
305 
306         throttle, yaw = 0.0, 0.0
307         if contours:
308             best_contour = max(contours, key=cv2.contourArea)
309             x, y, w, h = cv2.boundingRect(best_contour)
310             cx = x + w // 2
311             normalized_x = (cx - (width / 2)) / (width / 2)
312             yaw = self.yaw_pid.compute(normalized_x)
313             alignment = 1 - abs(normalized_x)
314             align_thres = 0.3
315             throttle = self.max_thr * ((alignment - align_thres) / (1 - align_thres)) if
            ↪  alignment >= align_thres else 0
316 
317             pt1, pt2, _, line_cx, line_cy = self.get_contour_line(best_contour)
318             cv2.drawContours(display_frame, [best_contour], -1, (0, 255, 0), 2)
319             cv2.line(display_frame, pt1, pt2, (255, 0, 0), 2)
320             cv2.circle(display_frame, (int(line_cx), int(line_cy)), 5, (0, 0, 255), -1)
321             cv2.line(display_frame, (int(width // 2), height), (int(line_cx), int(line_cy)),
            ↪  (0, 255, 255), 2)
322 
323         self.line_display_frame = display_frame.copy()
324         return throttle, yaw, None
325 
326     def get_contour_line(self, c, fix_vert=True):
327         vx, vy, cx, cy = cv2.fitLine(c, cv2.DIST_L2, 0, 0.01, 0.01).flatten()
328         scale = 100
329         pt1 = (int(cx - vx * scale), int(cy - vy * scale))
330         pt2 = (int(cx + vx * scale), int(cy + vy * scale))
331         angle = math.degrees(math.atan2(vy, vx))
332         if fix_vert:
333             angle -= 90 * np.sign(angle)
334         return pt1, pt2, angle, cx, cy
335 
336     def show_robot_status(self, frame, light_color, detected_sign):
337         status_frame = np.zeros((300, 500, 3), dtype=np.uint8)
```

```
338         state_text = self.state.name if self.state != RobotState.EXECUTE_SIGN else
            ↪ f"EXECUTE_{self.last_sign_detected.upper()}"
339         cv2.putText(status_frame, f"State: {state_text}", (20, 40),
340                     cv2.FONT_HERSHEY_SIMPLEX, 0.9, (255, 255, 255), 2)
341         color_text = light_color.upper() if light_color else "None"
342         cv2.putText(status_frame, f"Traffic Light: {color_text}", (20, 100),
343                     cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 255), 2)
344         sign_text = detected_sign.upper() if detected_sign else "None"
345         cv2.putText(status_frame, f"Sign Detected: {sign_text}", (20, 160),
346                     cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 0, 0), 2)
347
348         cv2.rectangle(status_frame, (20, 260), (120, 290), (0, 255, 0), -1)
349         cv2.putText(status_frame, "Start", (40, 282), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0,
            ↪ 0), 2)
350         cv2.rectangle(status_frame, (140, 260), (240, 290), (0, 255, 255), -1)
351         cv2.putText(status_frame, "Pause", (160, 282), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0,
            ↪ 0), 2)
352
353         cv2.imshow(self.status_window_name, status_frame)
354         if hasattr(self, 'circle_frame'):
355             cv2.imshow("Traffic Light Circles", cv2.resize(self.circle_frame, (640, 480)))
356
357     def destroy_node(self):
358         cv2.destroyAllWindows()
359         super().destroy_node()
360
361 def main(args=None):
362     rclpy.init(args=args)
363     node = FollowLineNode()
364     try:
365         rclpy.spin(node)
366     except KeyboardInterrupt:
367         pass
368     finally:
369         node.destroy_node()
370         rclpy.shutdown()
371
372 if __name__ == '__main__':
373     main()
```

TE3002B

PUZZLEBOT

Autonomous driving challenge

Team #3

Roberto Raymundo Gómez Vargas - A01285451
Humberto Barrera Martínez - A00836271
Mauricio Zavala Sánchez - A00837332
Erick Giovani Campos Aguirre - A01247257
Cristian Márquez Franco - A01748218
Roberto Eduardo Cardona Luis - A00833365

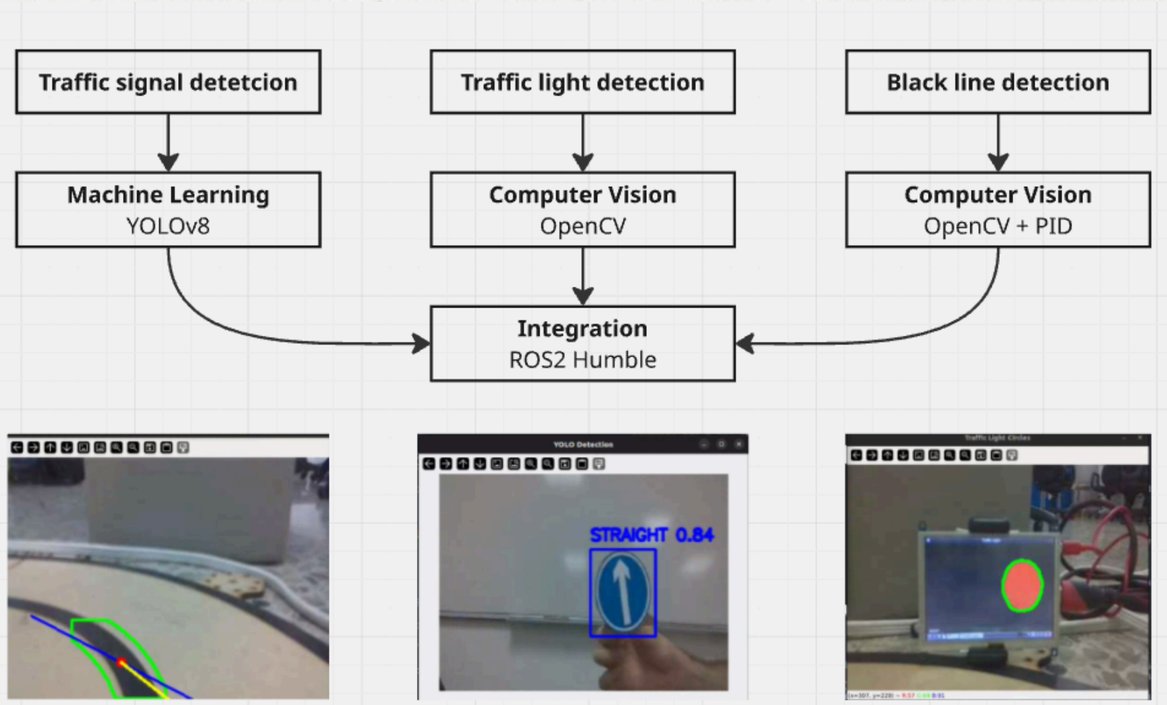
INTRODUCTION

In this challenge, we were tasked with designing and programming a fully autonomous mobile robot using the PuzzleBot platform.

Our main goal was to enable the robot to navigate a predefined track by following a black line, recognizing traffic signs , and reacting to traffic lights—red, yellow, and green—just like a real autonomous vehicle.

To achieve this, we combined several key technologies: computer vision and machine learning using a custom-trained YOLOv8 model, real-time image processing with OpenCV, and robot control using ROS 2 Humble running on Ubuntu 22.04.

The result is a robot that can make decisions on its own—slowing down for yellow lights, stopping at red lights, and executing maneuvers based on the detected signs.



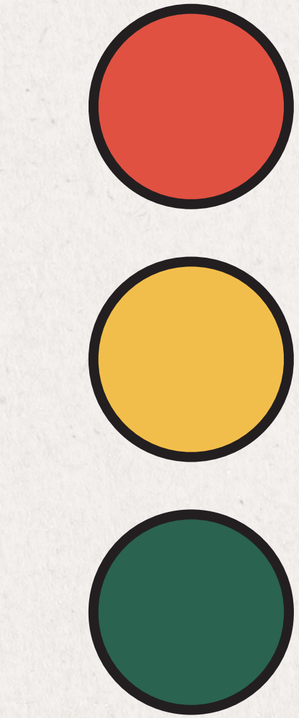
ABOUT THE CHALLENGE

- Develop an autonomous scale vehicle in 10 weeks capable of navigating a track with intersections, guiding line, traffic lights, and road signs, all using low-cost educational hardware.

- Ethical approach: Analyze social impact, road safety, and employment in Monterrey; justify decision-making criteria of the algorithm.

- Success indicators: Complete trajectory without human intervention, correct signal detection (> 95%), lateral deviation < 2 cm, stable speed.

- Constraints: Variable laboratory lighting, limited power (30,000 mAh), indoor Wi-Fi, defective parts.







SOLUTION DEVELOPMENT

Phase 1 – Hard-coded PWM

Draw a square (assembly check).

Phase 2 – Waypoints + PID

Consume list (x, y, θ); controller tuned via Ziegler–Nichols.

Phase 3 – Traffic-light CV

HSV segmentation; modulates speed and full stop.

Phase 4 – Line follower

Canny + morphology → centroid; error → angular PID.

Phase 5 – YOLOv8 signs

FSM ignores line, obeys stop/turn signs at intersection.

HARDWARE

Movile Base

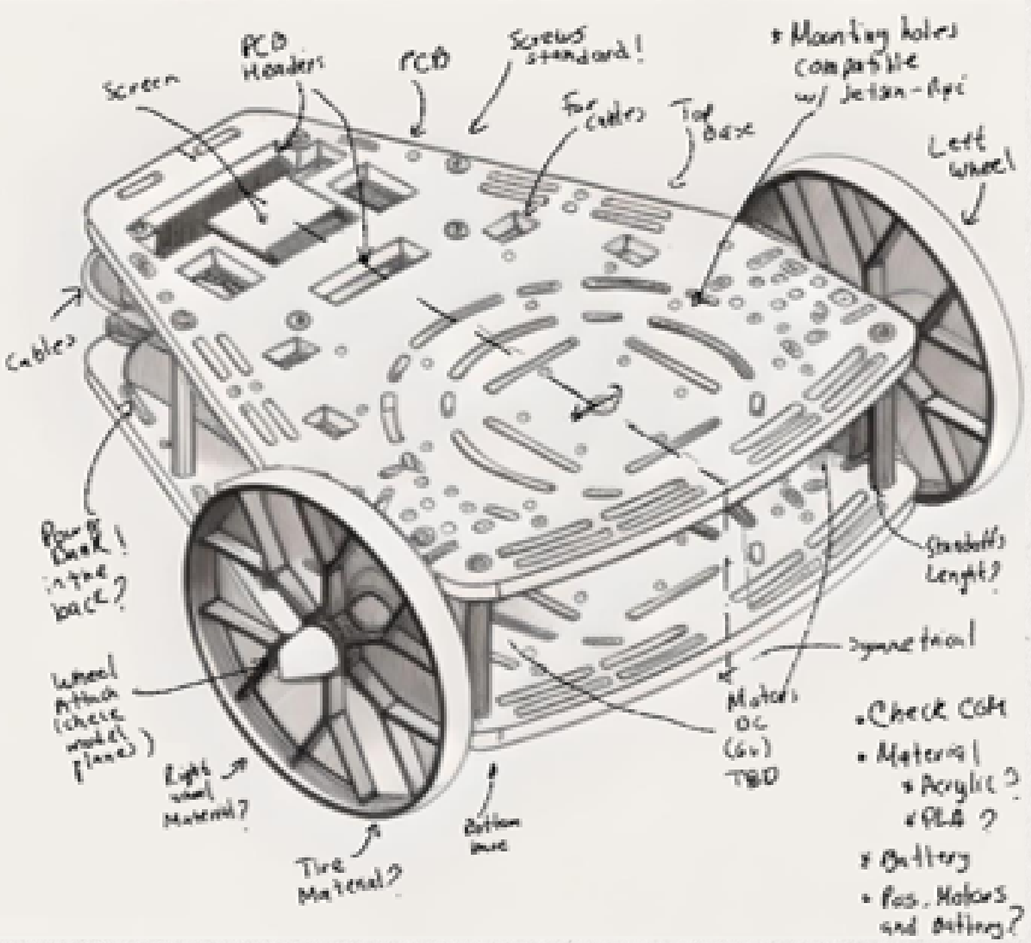
1:10 acrylic chassis, 6V brushless DC motors with 11 PPR encoders; 65 mm wheels.

HackerBoard

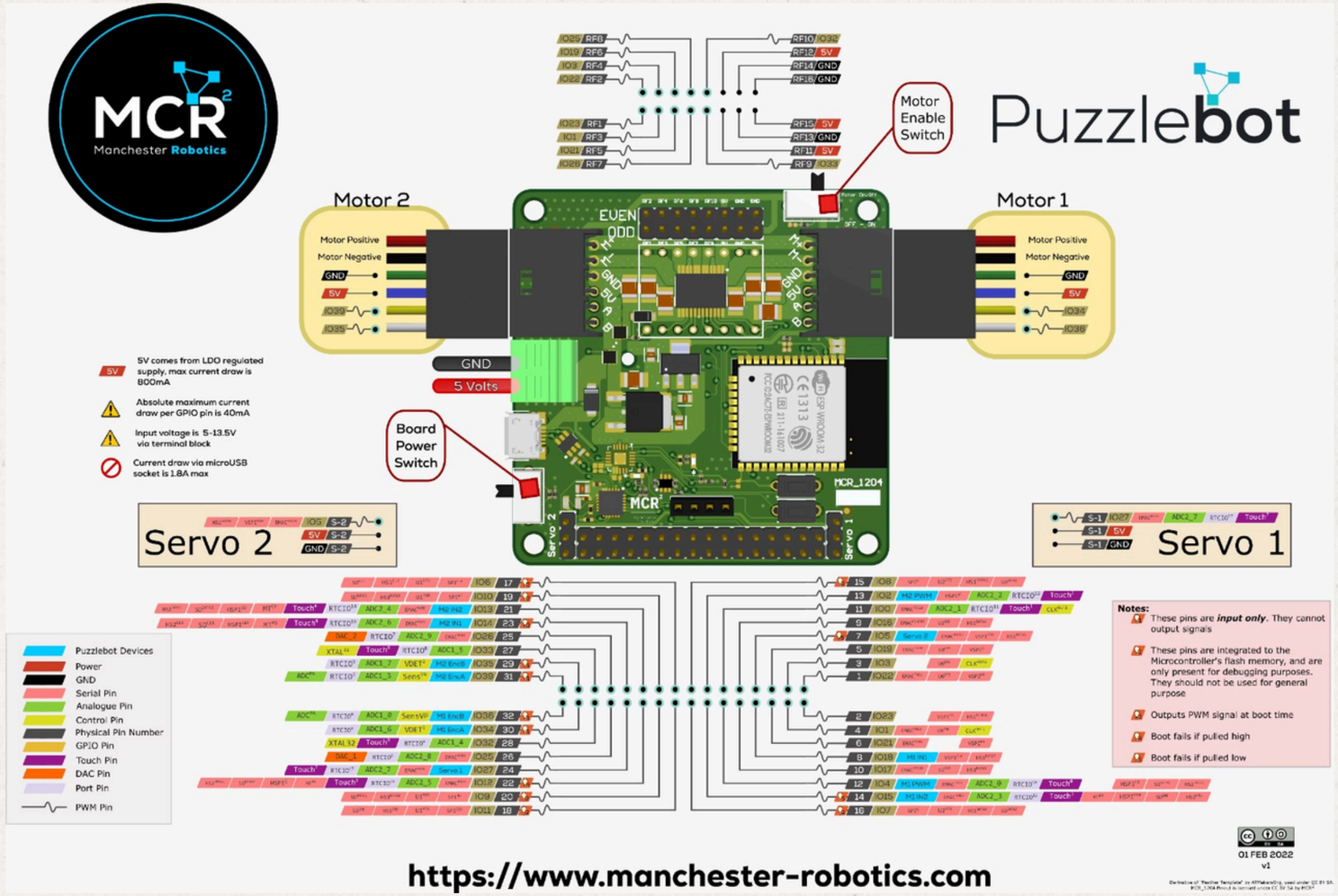
2×15 A PWM, dedicated ESC, I²C telemetry.

NVIDIA Jetson Nano 4 GB

Quad-A57 CPU @ 1.4 GHz + 128 Maxwell CUDA cores; 5 GHz USB Wi-Fi.



HARDWARE



Raspberry Pi V2 Camera

8 MP, f=3.6 mm lens, 62° FOV.

Steren 20k +10k mAh powerbanks

5 V-3 A + 9 V-2 A; ~1 hour continuous autonomy.

Wiring

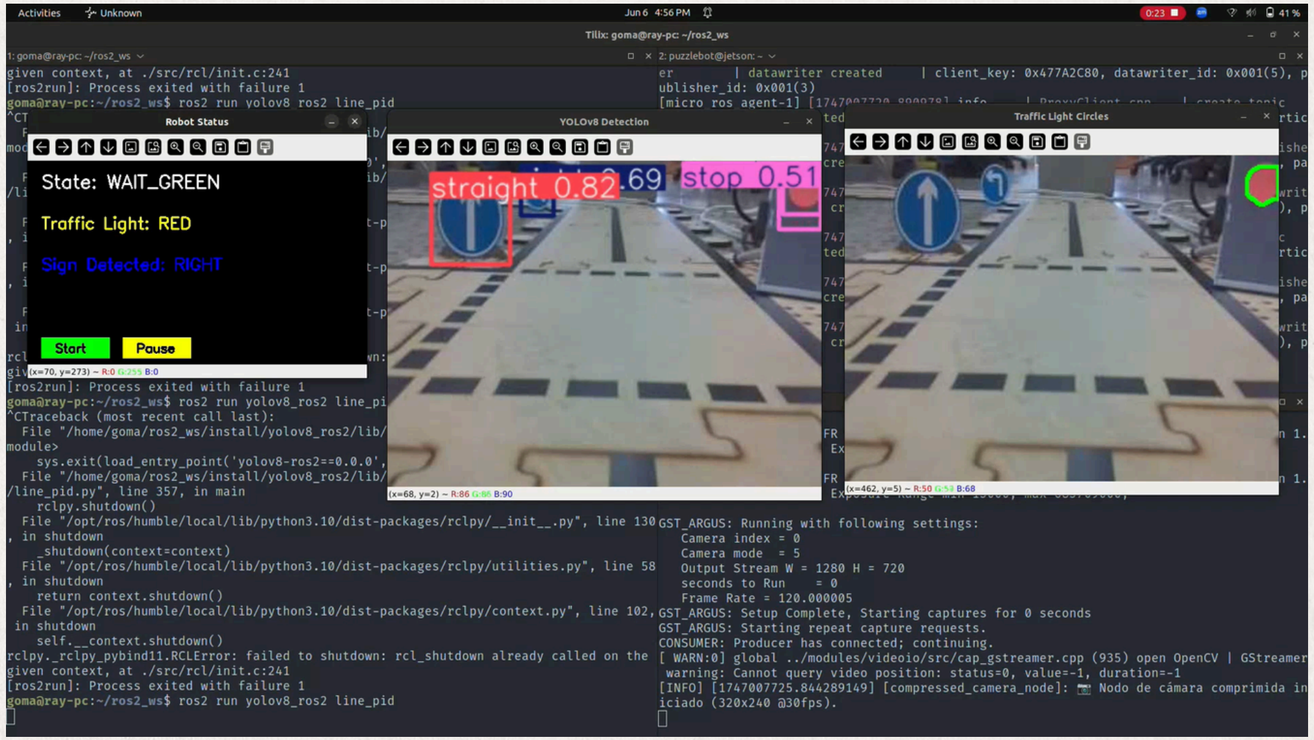
Dupont 22 AWG, 5 A fuse, printed PCB for 5 V/12 V distribution.

SOFTWARE

Jetson
Ubuntu 20.04.6 LTS-based, ROS 2 Humble, micro-ROS, Python 3.10 base of scrpts

Local Station
Ubuntu 22.04, ROS 2 Humble, Roboflow anot and training model Yolo

Package Management
colcon/rosdep, pipenv clean env

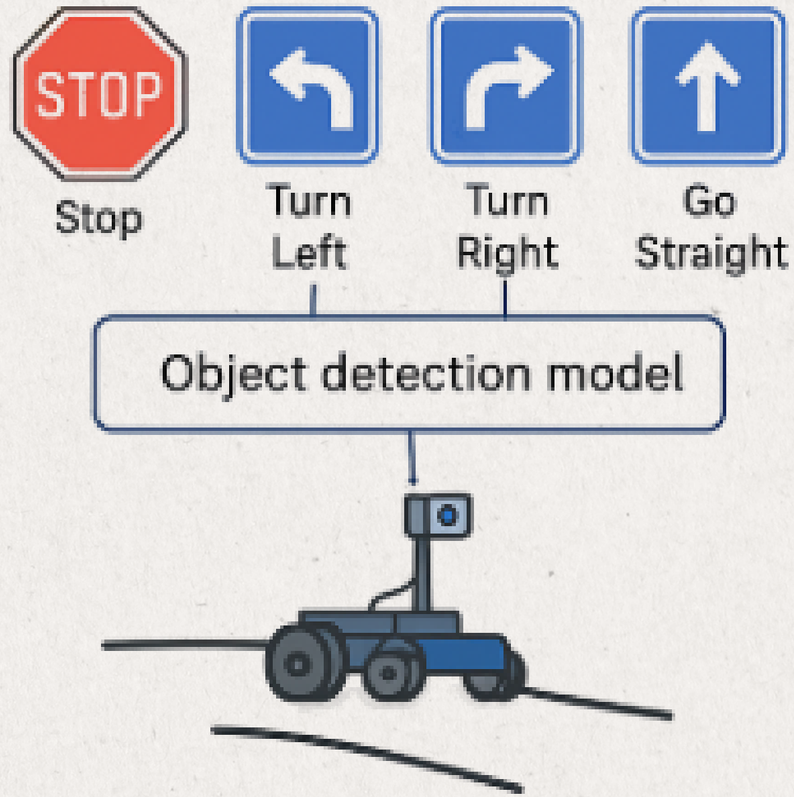


MACHINE LEARNING

What is Machine Learning?
It is a branch of artificial intelligence that allows computers to learn from data and improve performance without being explicitly programmed for each task.

How did we apply it in our project?
We trained a custom object detection model to recognize road signs like "Stop", "Turn Left", "Turn Right", and "Go Straight".

The robot used the trained model to interpret its environment and act accordingly, such as stopping at a stop sign or turning as directed.



YOLOV8

Image Preprocessing

A ROS 2 node applies a letterbox resize to keep the aspect ratio and uses a light Gaussian blur to reduce noise before inference.

Backbone (CSPDarknet)

- Extracts key textures and edges of the traffic signs while keeping GPU usage low on the Jetson Nano.

Neck (PANet + FPN)

- Detects both small signs (e.g., a distant “Turn Left” sign) and large ones (e.g., a close-up “Stop” sign) with equal precision.

Head (Detection Head)

- A convolutional network that, for each grid cell, predicts:
  - Class (Stop, Turn Left, Turn Right, Straight)
  - Bounding box (position, width, height)
  - Objectness confidence
- We filter out anything with confidence < 0.7 and ignore detections outside our four trained classes.

Post-procesamiento (NMS)

Non-Maximum Suppression removes overlapping boxes for the same sign, keeping only the most confident detection.

Integación

- YOLOv8 runs in a vision_detector node and publishes each detection as a BoundingBox + Label message.
- A separate state_machine node subscribes to those messages and converts each label into the corresponding robot command.

METHODOLOGY

Methodology

Traffic signal detetcion

- Recorded 7 IRL videos along the track.



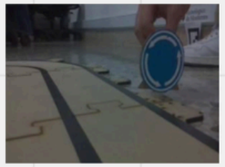




- Split videos into images
- 400 images per signal
- Data Augmentation













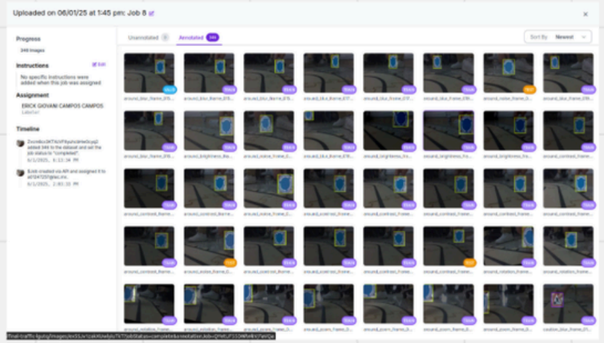




Data Augmentation

1. Blur
2. Brightness
3. noise
4. Original
5. Rotation
6. Translation
7. Zoom
8. Contrast

- Roboflow to anotate



- Train in Google Colab
- Use CUDA (T4 GPU)

```python
# Train the model
model.train(
    data=dataset.location + "/data.yaml",
    epochs=200,
    patience=15,
    imgsz=640,
    batch=16,
    project="traffic-signs",
    name="yolov8m_final",
    pretrained=True,
    optimizer="SGD",
    augment=True,
    cache=True,
    device=0,
    save=True,
    save_period=1,
    val=True
)
```

```
71 epochs completed in 1.641 hours.
Optimizer stripped from traffic-signs/yolov8m_final/weights/last.pt, 52.0MB
Optimizer stripped from traffic-signs/yolov8m_final/weights/best.pt, 52.0MB

Validating traffic-signs/yolov8m_final/weights/best.pt...
Ultralytics 8.3.148 🚀 Python-3.11.12 torch-2.6.0+cu124 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 92 layers, 25,843,813 parameters, 0 gradients, 78.7 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 13/13 [00:12<00:00,  1.06it/s]
                   all        416        416      0.996      0.995      0.994      0.707
                around         69         69          1      0.984      0.995      0.737
               caution         63         63      0.997          1      0.995      0.715
              giveaway         46         46      0.998          1      0.995      0.732
                  left         61         61      0.982      0.984      0.986      0.675
                 right         57         57      0.998          1      0.995       0.64
                  stop         62         62      0.999          1      0.995      0.707
              straight         58         58      0.995          1      0.995      0.744
Speed: 0.3ms preprocess, 23.6ms inference, 0.0ms loss, 1.9ms postprocess per image
Results saved to traffic-signs/yolov8m_final
```

Methodology

Traffic light detection

1. Prepare Image

Blurs the image slightly to reduce noise.

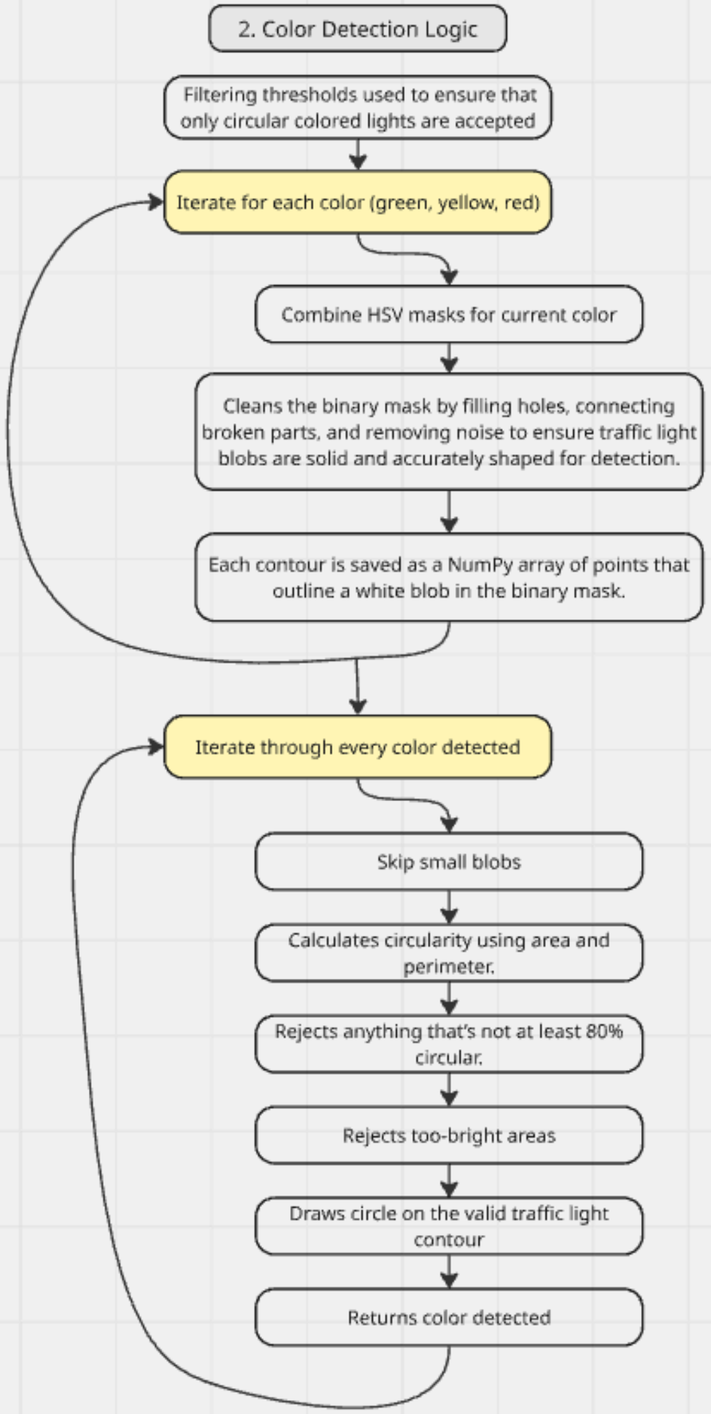
Converts the image from BGR to HSV, which is better for color detection.

→ Separates the HSV image into its three components
→ Applies CLAHE to brightness channel to improve lighting contrast.
→ Recombines the adjusted HSV image.

→ Each color maps to a list of masks with for different HSV ranges.
→ Red has 4 ranges due to its position on both ends of the hue circle.

```python
def detect_traffic_light_color(self, frame):
    self.circle_frame = frame.copy()
    frame_blur = cv2.GaussianBlur(frame, (9, 9), sigmaX=2, sigmaY=2)
    hsv = cv2.cvtColor(frame_blur, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    v_eq = clahe.apply(v)
    hsv = cv2.merge([h, s, v_eq])

    rangos = {
        'red': [
            cv2.inRange(hsv, (0,  70,  50), (10, 255, 255)),
            cv2.inRange(hsv, (0,  50, 200), (10, 255, 255)),
            cv2.inRange(hsv, (170, 70,  50), (180, 255, 255)),
            cv2.inRange(hsv, (170, 50, 200), (180, 255, 255))
        ],
        'yellow': [
            cv2.inRange(hsv, (20,  60,  60), (35, 255, 255)),
            cv2.inRange(hsv, (20,  40, 200), (35, 255, 255))
        ],
        'green': [
            cv2.inRange(hsv, (45,  80,  60), (75, 255, 255)),
            cv2.inRange(hsv, (45,  60, 200), (75, 255, 255))
        ]
    }
```

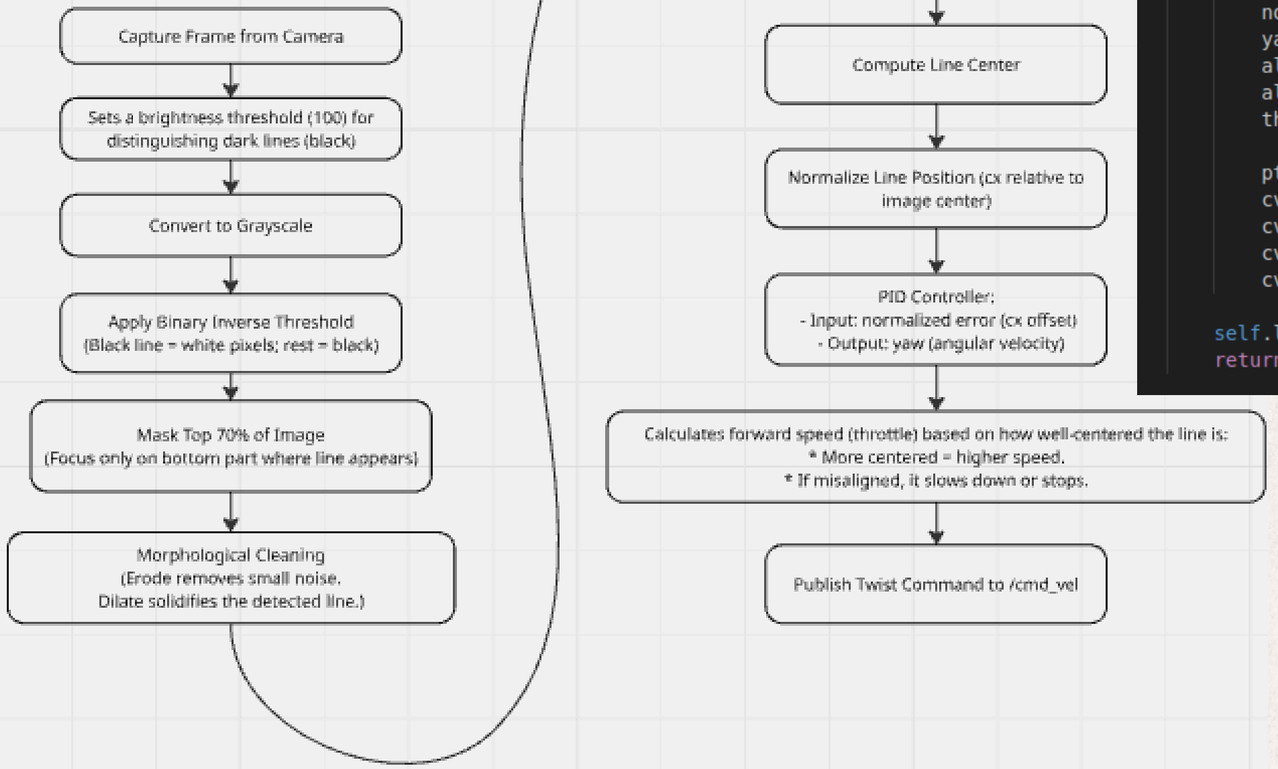



```python
min_area = 200
circularity_th = 0.8

for color, masks in rangos.items():
    mask_full = masks[0].copy()
    for m in masks[1:]:
        mask_full = cv2.bitwise_or(mask_full, m)
    kernel_close = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7, 7))
    mask_full = cv2.morphologyEx(mask_full, cv2.MORPH_CLOSE, kernel_close, iterations=2)
    mask_full = cv2.medianBlur(mask_full, 5)
    contours, _ = cv2.findContours(mask_full, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    for cnt in contours:
        area = cv2.contourArea(cnt)
        if area < min_area:
            continue
        perimetro = cv2.arcLength(cnt, True)
        if perimetro == 0:
            continue
        circularity = (4 * np.pi * area) / (perimetro * perimetro)
        if circularity < circularity_th:
            continue
        mask_cnt = np.zeros(frame.shape[:2], dtype=np.uint8)
        cv2.drawContours(mask_cnt, [cnt], -1, 255, -1)
        _, _, v_channel = cv2.split(hsv)
        total_pixels = cv2.countNonZero(mask_cnt)
        if total_pixels == 0:
            continue
        mean_valor = cv2.sumElems(cv2.bitwise_and(v_channel, v_channel, mask=mask_cnt))[0] / total_pixels
        if mean_valor > 240:
            continue
        cv2.drawContours(self.circle_frame, [cnt], -1, (0, 255, 0), 2)
        return color, None
return None, None
```

Methodology

Black line detection




```python
def follow_line(self, frame):
    display_frame = frame.copy()
    height, width = frame.shape[:2]
    dark_thres = 100

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    _, mask = cv2.threshold(gray, dark_thres, 255, cv2.THRESH_BINARY_INV)
    mask[:int(height * 0.7), :] = 0
    mask = cv2.erode(mask, np.ones((3, 3), np.uint8), iterations=3)
    mask = cv2.dilate(mask, np.ones((5, 5), np.uint8), iterations=5)

    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contours = [c for c in contours if cv2.contourArea(c) > 1000]

    throttle, yaw = 0.0, 0.0
    if contours:
        best_contour = sorted(contours, key=cv2.contourArea, reverse=True)[0]
        x, y, w, h = cv2.boundingRect(best_contour)
        cx = x + w // 2
        normalized_x = (cx - (width / 2)) / (width / 2)
        yaw = self.yaw_pid(normalized_x)
        alignment = 1 - abs(normalized_x)
        align_thres = 0.3
        throttle = self.max_thr * ((alignment - align_thres) / (1 - align_thres)) if alignment >= align_thres else 0

        pt1, pt2, angle, line_cx, line_cy = self.get_contour_line(best_contour)
        cv2.drawContours(display_frame, [best_contour], -1, (0, 255, 0), 2)
        cv2.line(display_frame, pt1, pt2, (255, 0, 0), 2)
        cv2.circle(display_frame, (int(line_cx), int(line_cy)), 5, (0, 0, 255), -1)
        cv2.line(display_frame, (int(width // 2), height), (int(line_cx), int(line_cy)), (0, 255, 255), 2)

    self.line_display_frame = display_frame.copy()
    return throttle, yaw, None
```

Methodology

State Machine

States

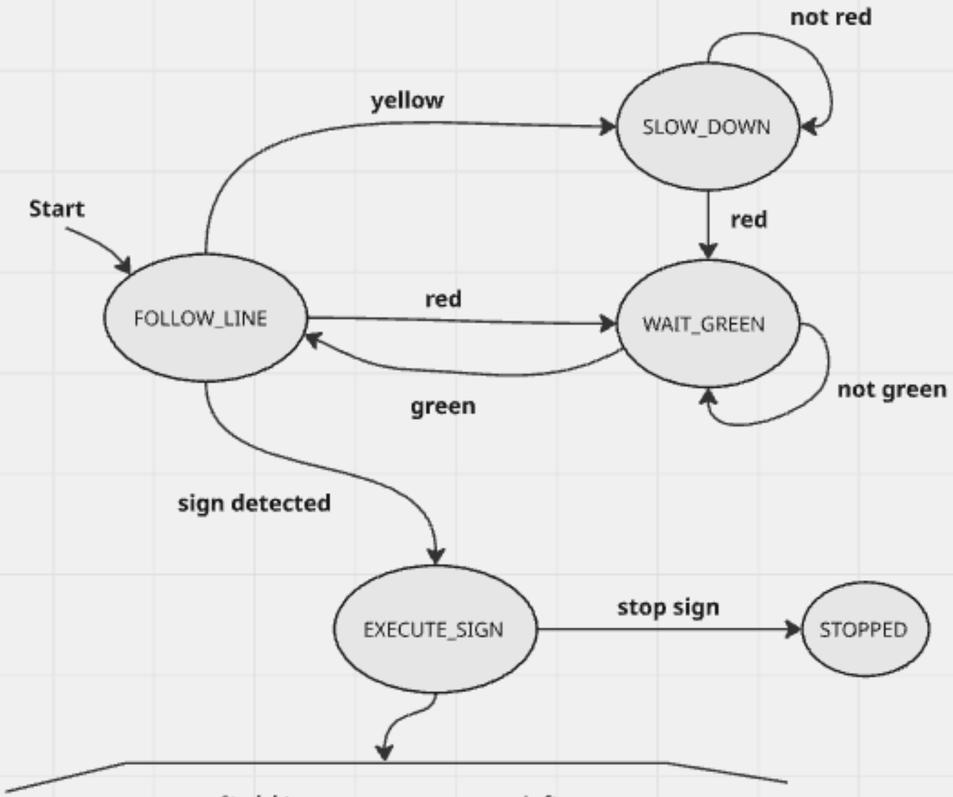
1. FOLLOW_LINE
Robot follows the black line using PID and computer vision.

2. WAIT_GREEN
Stops

3. SLOW_DOWN
Changes velocity to 50%

4. EXECUTE_SIGN
Identifies which sign is detected and follows the corresponding instructions.

5. STOPPED
Stops



Straight

Continue forward until the black line is reached, then return to **FOLLOW_LINE**.

Left

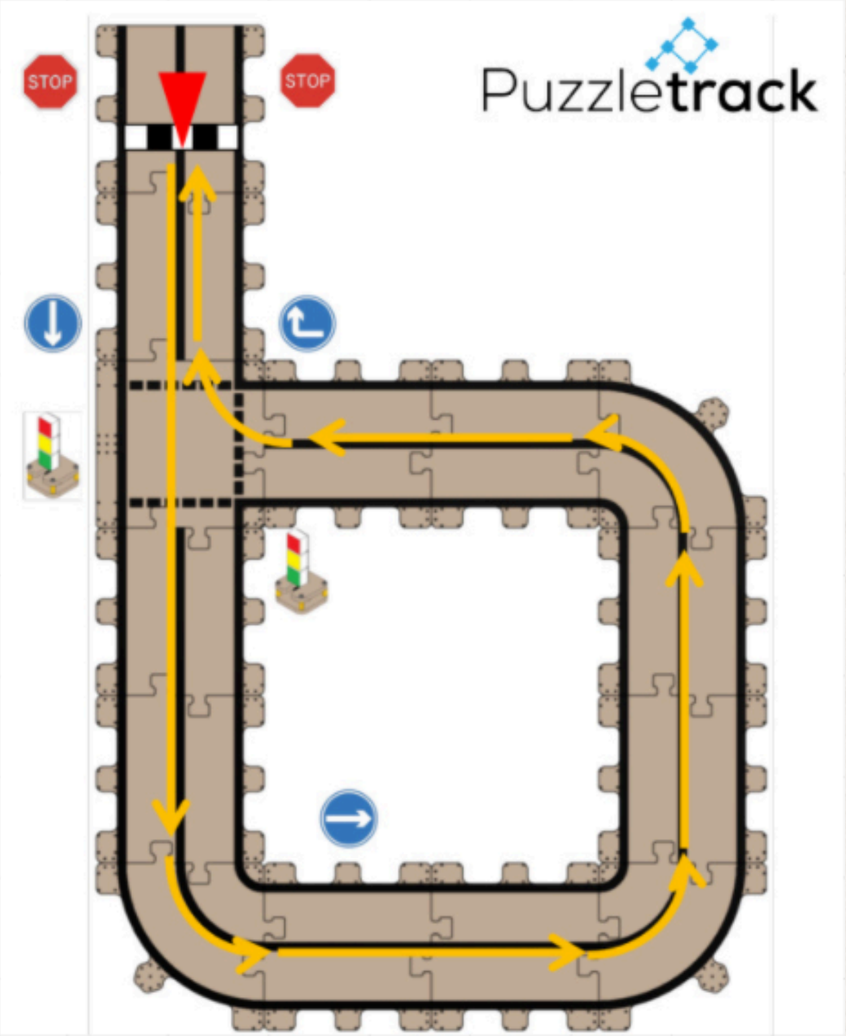
Return to **FOLLOW_LINE**.

Right

Move forward
Make right turn
Move forward
Return to **FOLLOW_LINE**.

Stop

Go to **STOPPED**.



RESULTS



https://youtu.be/NXRFa3HvvB4

CONCLUSION

CHALLENGES & SOLUTIONS

Challenge	Solution
Inconsistent lighting affecting traffic light color detection	Applied CLAHE + HSV filtering + contour shape + brightness filtering to make detection robust under varying conditions
YOLOv8 misclassifying sings occasionally	Filtered detections by confidence threshold ($\geqslant 0.7$) and limited to known classes (stop, right, left, straight)
Black line partially occluded of faded	Used morphological cleaning and selected only the largest contour to stay locked onto the line
Robot drifting or over-correcting	Tuned PID controller parameters for smooth real-time adjustment
One Motor was stronger/faster than the other, causing the robot to drift	Manually compensated in the code by adding slightly more velocity to the slower motor
Synchronizing traffic light logic with state machine	Designed a state-based architecture using enums to handle transitions cleanly

CONCLUSION

To conclude, this project combined computer vision, machine learning, robotics, and ROS 2 to create a fully autonomous robot capable of detecting traffic signals, following a black line, and executing commands like turning and stopping.

We overcame real-world challenges like uneven lighting and motor imbalance with smart software solutions. The result is a robust, adaptable system that performs reliably in dynamic environments.

Throughout this project and the semester, we also significantly improved both our technical and soft skills. The abilities developed during this experience will be of great value in our future professional careers.

