Autonomous Driving Duckiebot

GROUP V

SUBMITTED BY

Xinran Shen, Ruilai Ma, Weiyuan Du, Jing Cheng,
Yijie Gao, Yingtong Tian

SUPERVISOR

Prof. Dr. Amr Alanwar

Heilbronn, 02.09.2025

1 Abstract

This project focuses on developing a self-driving vehicle system that combines lane-keeping, and road environment detection to mimic core autonomous driving behaviors. An opencv-based lane detection module keeps the vehicle on course using PID control. Road sign recognition adds to the autonomous capabilities of the vehicle, allowing it to respond appropriately to road signage. The perception modules feed into a decision layer that keeps track of various states of the vehicle to plan safe and efficient trajectories. Finally, a control layer translates high-level decisions into low-level motor commands for steering and speed. The autonomous driving system is written in python, and tested in the gym-duckietown simulator, ensuring maximum similarities with a real Duckietown environment. This project provides hands-on experience with computer vision and cyber-physical systems, building a foundation for more advanced autonomous vehicle research.

2 Introduction

This project implements fundamental autonomous driving functions on the Duckiebot DB21J platform, with development and validation primarily conducted in the Gym-Duckietown simulator.

The implemented functions include:

- Open-loop and feedback-based PID control for motion execution
- Lane keeping to maintain stable trajectory within marked lanes
- Traffic light and sign recognition and decision-making for compliance with signals
- Reinforcement Learning for Obstacle Avoidance

3 Hardware Setup

- **Platform**: Duckiebot DB21J
- **Processing Unit**: Jetson Nano 4GB
- **Notable Features**:
  - Front-facing camera
  - Wheel encoders
  - Inertia measurement unit
  - Time-of-Flight sensor

4 Simulator Environment

Due to limitations with experimenting on a physical robot, we chose to implement most of our work in the gym-duckietown simulator. Using Docker, we have successfully set up a stable testing environment with a variety of maps, in order to make sure our algorithms would function well when translated to the physical robot. Mounting the container onto a desktop location would also allow ease of development, since all files can be moved to-and-from the container directly on the host.

5 Software Architecture

The software architecture of our Duckiebot system is organized into three main layers. These layers build upon each other, starting from low-level motor control, to mid-level perception and navigation, and finally to high-level semantic decision-making. This modular design ensures clarity, extensibility, and alignment with the principles of autonomous driving systems.

5.1 Color Recognition and LED

This section integrates computer vision and color recognition by receiving compressed images from the robot's camera node and converting them to HSV color space. And then the Duckiebot's LEDs are dynamically controlled according to the color the camera recongized. The results are visualized in real time and fully integrated with ROS topics and LED publishers.

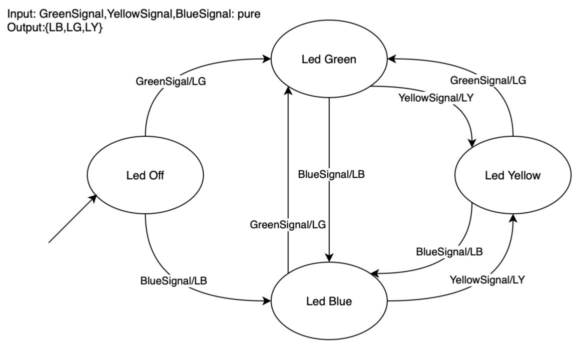

Figure 1: State machine of the color recognition system

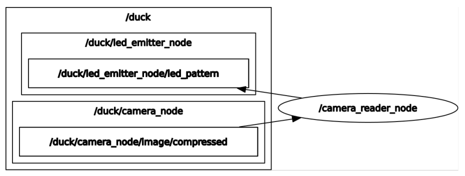
/duck
/duck/led_emitter_node
/duck/led_emitter_node/led_pattern
/duck/camera_node
/duck/camera_node/image/compressed
/camera_reader_node

Figure 2: ROS graph of Camera_reader

Camera_reader first subscribes to the `/camera_node/image/compressed` topic via ROS to receive compressed image data from the onboard camera in real time. Upon receiving each new image message, the node uses OpenCV and `cv_bridge` to decode the ROS image message into OpenCV format and convert it to the HSV color space to recongize different colors we pre-defined in the code (Green, Blue, Yellow).

After completing color recognition, the node dynamically determines the LED display color based on the largest detected color region in the current frame. It publishes `LEDPattern` messages to the `/led_emitter_node/led_pattern` topic via ROS, setting all LEDs to the RGB values matching the dominant color,providing immediate confirmation of the recognition performance.

The state machine is shown in Figure 1. Which transfer between different colors according to the color signal it recievied. And to be noted, this state machine does not have an exit, which means it only shutdown when it recieve and interruption from the keyboard.

5.2 Open Loop Control

To better understand Duckiebot's motion control, we designed an Open Loop Square Task and its corresponding control system. The open loop control system executes a time-based square trajectory without feedback correction. The robot follows pre-calculated motion sequences based on kinematic models and timing constraints.

The square trajectory consists of four equal sides of length $L = 1.0$ meter and four $90°$ turns. Figure 3 illustrates the entire process, starting from the lower right corner of the square. After moving forward 1 meter, the bot turns left and changes its LED color to green. This pattern repeats, with the LED changing colors according to the current phase. Upon completing the task, the bot stops at the initial start point and turns off the LED.

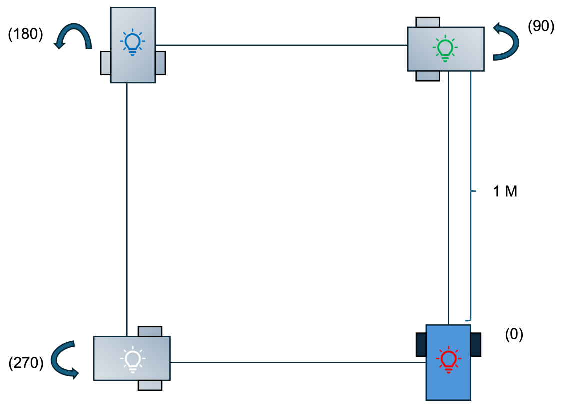

Figure 3: Open-loop square task execution

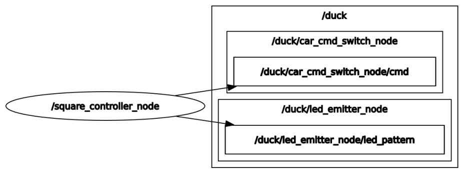
/duck
/duck/car_cmd_switch_node
/duck/car_cmd_switch_node/cmd
/square_controller_node
/duck/led_emitter_node
/duck/led_emitter_node/led_pattern

Figure 4: ROS graph of the square controller

If we define the initial start point as $(0, 0)$ in the x-y plane, the target positions for the task are sequentially $(0, 1)$, $(-1, 1)$, $(-1, 0)$, and finally returning to $(0, 0)$, which is the origin of our coordinate system.

In our experiment, the bot is configured with the following motion parameters:

$$\begin{cases} \text{Linear velocity:} & v = 0.3 \text{ m/s} \\ \text{Angular velocity:} & \omega = 4.0 \text{ rad/s} \\ \text{Move time per edge:} & t_{\text{move}} = \frac{L}{v} = \frac{1.0}{0.3} = 3.33 \text{ seconds} \\ \text{Turn time per corner:} & t_{\text{turn}} = \frac{\pi/2}{\omega} = \frac{\pi/2}{4.0} = 0.393 \text{ seconds} \end{cases}$$

The control algorithm publishes **Twist2DStamped** messages to the motor control topic, using the kinematic model:

$$\begin{bmatrix} \dot{x} & \dot{y} & \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos(\theta) & v\sin(\theta) & \omega \end{bmatrix} \tag{1}$$

Each corner is associated with a specific LED color (Red, Green, Blue, White) to provide visual feedback. The state machine operates with a timer sequence as follows:

1. Set corner LED color
2. Move forward for the calculated duration
3. Turn left for the calculated duration
4. Repeat for the next corner

Below is the visualization of the system's state machine. Using an internal ticker, the system measures elapsed time and controls the bot's state by setting time thresholds and resetting the timer.

At initialization, two variables are introduced: c, which records the current corner position, and Time, which tracks the elapsed time. The variable p is a pure signal indicating the system's status; when p is present, the bot is considered to be executing the task sequence normally, otherwise not. The system ticker runs at 10 Hz, and the timer resets upon every state transition. By alternating left turns and forward movements, the defined task is accomplished.

Although this design roughly achieves the intended task, the process is very unstable. It relies heavily on a reliable environment and is affected by factors such as the bot's battery level. After multiple experiments, we found that this implementation does not robustly meet our goals. This approach yields predictable motion but lacks error correction, making it vulnerable to wheel slip, calibration errors, and environmental disturbances.

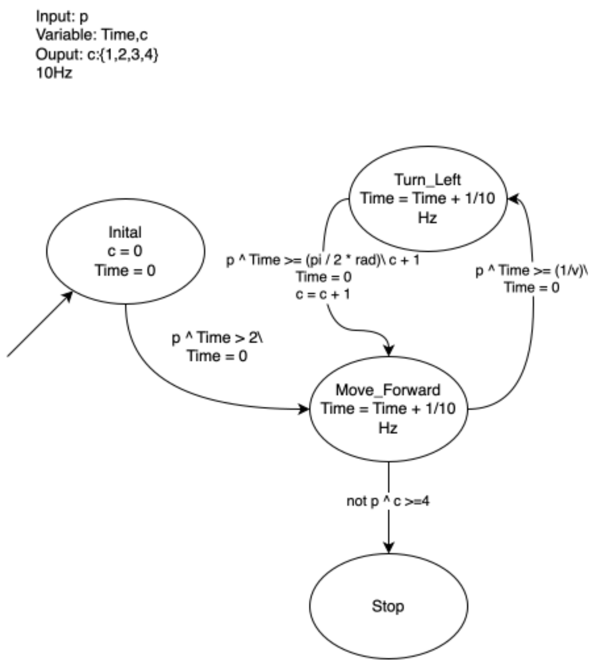

Figure 5: State machine of Open Loop Control

5.3 Odometry

odometry_node and **odometry_activity** together form the foundational odometry system for robot. This system enables real-time pose estimation and publishing based on wheel encoder data. It uses core kinematic algorithms implemented in odometry_activity to estimate the robot's motion.

To be more specific, the odometry activity encapsulates the kinematic formulas of a differential-drive robot. It includes functions for calculating the rotational angle increment of each wheel (δ_ϕ) and for estimating the robot's new pose (estimate_pose) based on the movement of both wheels.

$$\Delta_{\text{wheel}} = \frac{2\pi R N_{\text{ticks}}}{N_{\text{total}}} \tag{2}$$

where R is wheel radius, b is baseline (wheel separation), $\Delta\phi_{L,R}$ are wheel rotations, and dA is distance traveled.

The pose update is performed in two steps. First, compute the incremental motion over a small time interval based on wheel encoder readings:

$$\begin{aligned}\Delta x &= dA\cos(\theta^{(t)})\\ \Delta y &= dA\sin(\theta^{(t)})\\ \Delta\theta &= \frac{dr-dl}{2L}\\ dA &= \frac{dr+dl}{2}\end{aligned} \tag{3}$$

Then, update the global pose of the robot as:

$$\begin{aligned}x_w^{(t+1)} &= x_w^{(t)} + \Delta x\\ y_w^{(t+1)} &= y_w^{(t)} + \Delta y\\ \theta_w^{(t+1)} &= \theta_w^{(t)} + \Delta\theta\end{aligned} \tag{4}$$

This design allows odometry node to focus solely on data flow and ROS message publishing, while all mathematical modeling and computations related to physical motion are handled within odomety activity.

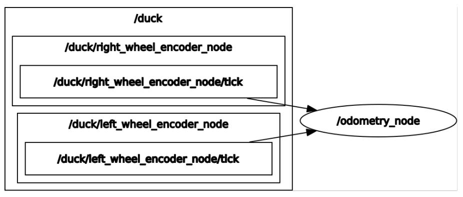
/duck
/duck/right_wheel_encoder_node
/duck/right_wheel_encoder_node/tick
/duck/left_wheel_encoder_node
/duck/left_wheel_encoder_node/tick
/odometry_node

Figure 6: ROS graph of the odometry node

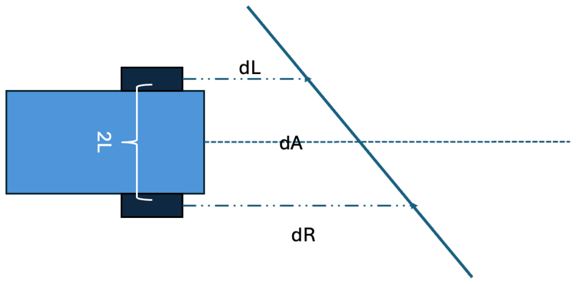

Figure 7: Odometry computation diagram based on wheel encoders

During actual execution, odometry node first initializes and retrieves the robot's physical parameters, such as wheel radius and wheelbase. It then continuously listens for encoder messages from both the left and right wheels. Once new encoder data has been received for both wheels, it computes the incremental angular displacement for each wheel and accumulates these values for the current cycle. It then estimates positions of the robot.

The node calculates the linear and angular velocity based on the time interval and publishes the updated pose and velocity information in the standard ROS Odometry and Pose2DStamped message formats. It also broadcasts the TF coordinate transform to support spatial localization and visualization by other ROS nodes.

5.4 Closed Loop Control

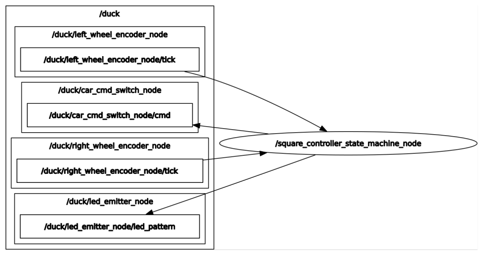
/duck
/duck/left_wheel_encoder_node
/duck/left_wheel_encoder_node/tick
/duck/car_cmd_switch_node
/duck/car_cmd_switch_node/cmd
/square_controller_state_machine_node
/duck/right_wheel_encoder_node
/duck/right_wheel_encoder_node/tick
/duck/led_emitter_node
/duck/led_emitter_node/led_pattern

Figure 8: ROS graph of Closed Loop Control

The closed loop control system implements feedback-based navigation using wheel encoder odometry for precise pose estimation and trajectory following with real-time error correction. It use the information from the Odometry node to control the system.

During the bot's execution of the square trajectory task, we defined two types of errors: distance error and angular error. The distance error is defined as the distance between the bot's stopping position and the target point at the end of the current task. The angular error is defined as the difference between the bot's heading direction and the desired target orientation each time before starting the next task. We set a tolerance threshold for these errors, not to completely eliminate them, but to maintain the errors within an acceptable range through feedback control.

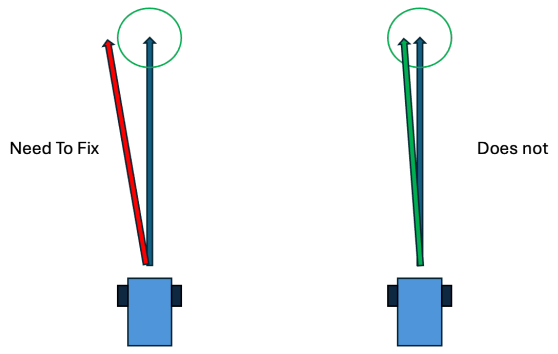

Figure 9:

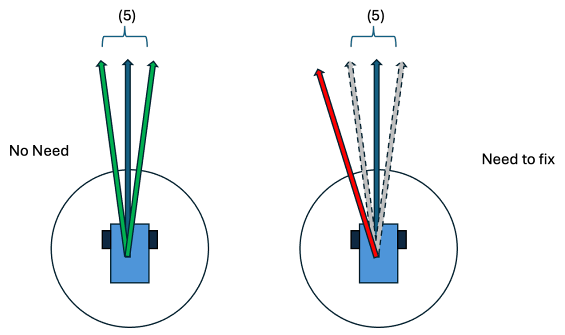

Figure 10:

The two types of errors used in the position control task.

5.4.1 Position Control

The position controller in Figure 9 employs proportional control to navigate between waypoints. Given a target position (x_t, y_t) and the current position (x_c, y_c):

$$\begin{aligned} \text{Distance error:} \quad & e_d = \sqrt{(x_t - x_c)^2 + (y_t - y_c)^2} & (5) \\ \text{Desired heading:} \quad & \theta_d = \arctan 2(y_t - y_c, x_t - x_c) & (6) \\ \text{Angular error:} \quad & e_\theta = \text{normalize}(\theta_d - \theta_c) \quad \text{with } e_\theta \in [-\pi, \pi] & (7) \end{aligned}$$

The control outputs are computed as:

$$\begin{cases} v = K_v \cdot \text{sign}(e_d), & \text{if } |e_d| > \epsilon_d \\ \omega = K_\omega \cdot e_\theta \end{cases} \tag{8}$$

5.4.2 Orientation Control

Precise control of the robot's heading direction is crucial, especially when the robot needs to rotate in place before moving forward. For pure rotational control like in Figure 10, the objective is to align the robot's current orientation θ_c with the target orientation θ_t without any linear displacement or with a minor difference.

The controller first calculates the angular error e_θ, which represents the smallest signed difference between the target and current orientations. This error is normalized to the range $[-\pi, \pi]$ to ensure smooth rotational behavior:

$$e_\theta = \text{normalize}(\theta_t - \theta_c) \tag{9}$$

To achieve pure rotation, the linear velocity is set to zero:

$$v = 0 \tag{10}$$

The angular velocity ω is then controlled proportionally to the angular error using a gain K_ω:

$$\omega = K_\omega \cdot e_\theta \tag{11}$$

This proportional control law drives the robot to minimize the orientation error smoothly and efficiently. The gain K_ω is tuned to balance responsiveness and stability, avoiding oscillations or overshoot in the robot's rotation.

By focusing solely on orientation adjustment in this mode, the controller ensures that the robot achieves the desired heading before proceeding with translational movements, which is essential for accurate waypoint navigation and overall path following.

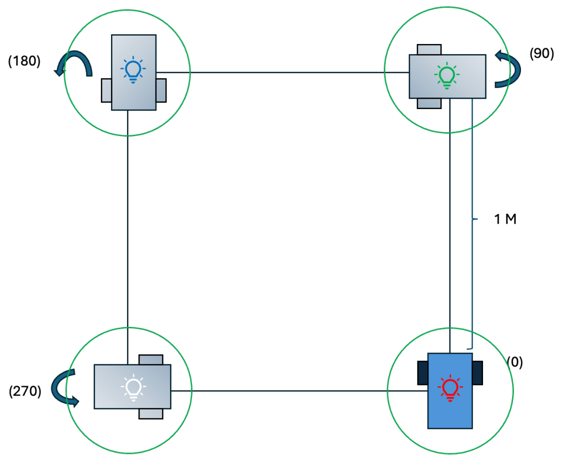

Figure 11: Closed-loop control square task

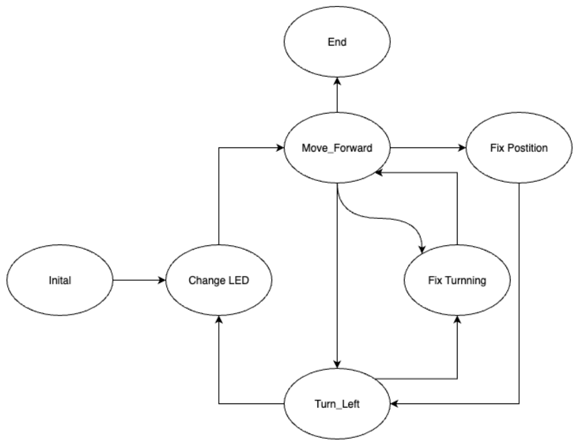

Figure 12: Ideal state machine of the closed-loop control (simplified)

5.4.3 Ideal State Machine

An ideal closed-loop controller is implemented as a finite state machine that precisely navigates through the following waypoints and target orientations:

The controller continuously estimates the robot's pose, computes control errors, applies proportional control laws, and publishes velocity commands, ensuring robust trajectory tracking despite disturbances and model uncertainties.

Although both distance and angular errors were defined, our experiments primarily focused on orientation control. This decision is based on observations from open-loop control experiments, where the bot's linear displacement errors remained within the acceptable tolerance. Therefore, orientation regulation was emphasized, as maintaining correct heading is generally more critical than speed control.

5.5 Analyze the Performance

After completing the closed-loop control task, we can visualize the bot's trajectory throughout the task by extracting and plotting data from the log output. By comparing this actual trajectory with the ideal path, as shown in the figure 13, we observe that the bot repeatedly attempts to adjust its orientation during movement. Each time the bot deviates from the target position or orientation, it performs corresponding yaw corrections to counteract errors caused by mechanical inaccuracies.

For instance, in our experiment, even after straight-line correction, the bot tended to drift left if the distance was long enough — a phenomenon clearly visible in the plotted trajectory. Our closed-loop control system continuously attempts to correct such deviations. In comparison, the bot's distance prediction proved to be more accurate, with a final position error of only 0.135 meter from the target.

This highlights the impact of systematic errors and the necessity of feedback regulation to achieve ongoing correction and improved control accuracy.

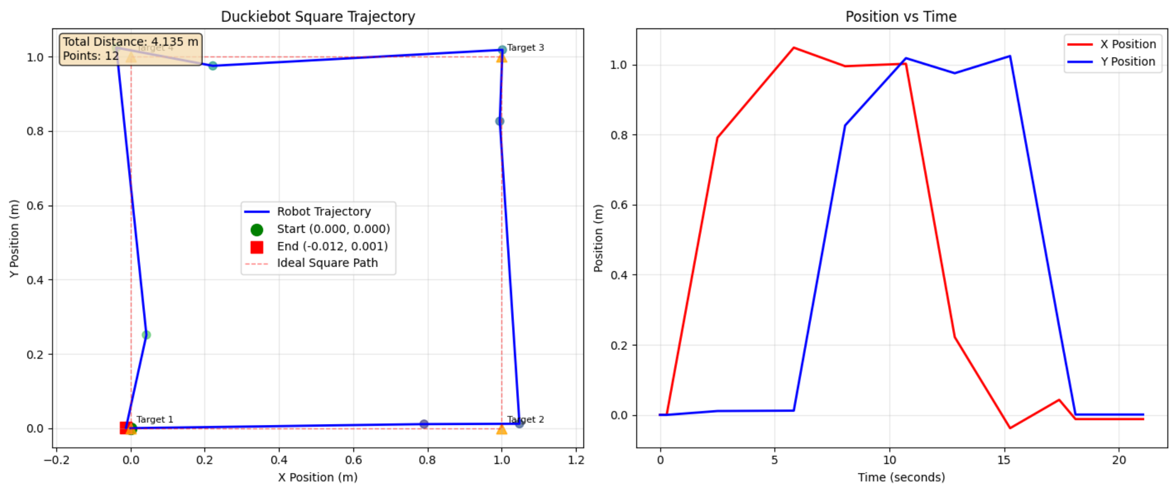

Figure 13: The position change of Robot

5.6 Lane Keeping

For lane keeping, we devised two approaches: PID Control and Lane Servoing, each with its own unique drawbacks and advantages.

5.6.1 PID Control

First, lane boundaries were extracted from the onboard camera image using computer vision techniques:

- Preprocessing: Images are enhanced via CLAHE for better contrast and converted to the LAB and HSV color spaces.
- Yellow line detection: Adaptive HSV filtering resistant to changes in illumination.
- White line detection: Combined HSV and LAB thresholds restricted to the right side of the image, with morphological cleanup. A histogram-based fallback was used when no strong white lines were visible.

From these masks, lane centers were estimated using horizontal slicing and Hough line transforms, enabling computation of lane position and curvature.
In order to accomplish lane keeping, we implemented an enhanced PID controller. The PID controller performs the standard corrective steps:

- Proportional adjustment of the steering force based on the error from the desired direction

- Integral adjustment of small biases, preventing them from building up over time
- Derivative adjustment reduces the tendency to over-correct from the proportional adjustment

Additionally, we also implemented turn-aware control:
Normal PID control only works when lines on both sides are present. However, this is not true when turning. Our algorithm accounts for turns and completes the maneuver, despite the missing reference line. In the event that both reference lines are lost, it searches for reference lines in-place and reacquires its lane-keeping target.

To combine the functionality above, a hybrid automaton was implemented that switches between discrete operational modes depending on lane visibility:

- **Normal Lane Following:** Both lane lines visible, PID controller active.
- **Turn Mode (Left/Right):** Triggered when one line disappears for several consecutive frames. The vehicle biases toward the visible line, applies a steer bias, reduces speed, and ramps steering effort.
- **Recovery/Search Mode:** Activated when both lines are missing for too long. The vehicle first steers toward the last known side, then performs sweeping searches at low speed until a line is reacquired.

The PID Control approach was able to effectively navigate simple roads and keep the bot centered on its lane. However, it is less reliable when additional environmental noise is introduced, and it does not handle intersections well.

5.6.2 Lane Servoing

As with PID Control, first, lane boundaries were extracted from the onboard camera image using computer vision techniques:

- Preprocessing: Images are converted to the HSV color space.
- Yellow line detection: HSV filtering resistant to changes in illumination.
- White line detection: HSV thresholds restricted to the right side of the image, with morphological cleanup.

From these masks, lane centers were estimated using a weighted mean calculation, differentiating it from the PID Control approach, enabling computation of lane position and curvature.
Then, the lane servoing controller kicks in:

- The lane marking masks are multiplied by a pre-set weighting matrix, giving a value for each pixel in the masks

- The weighted results are converted into a steering vector and passed to the wheels.
- Essentially, this is a variation of the ”P” portion of PID Control

This lane keeping controller effectively navigates simple roads as well as intersections. However, its shortcomings show as it stays in the middle of the entire road, often on top of the yellow lane, due to flaws in its perception and control logic. This is where PID Control performs better.

5.7 Traffic Light Recognition

A computer vision pipeline was implemented to detect and interpret traffic lights in the simulated environment. It once again uses a finite-state machine model of operation:

- Normal state: The algorithm follows lane-keeping behavior
- Red light state: This state is triggered when the robot encounters a red light at an intersection. The robot stops and waits for a green light.
- Green light state: This state is triggered when the robot encounters a green light at an intersection. The robot proceeds straight through the intersection, and resumes lane-keeping behavior afterwards.

5.8 Traffic Sign Recognition

Another computer vision pipeline was implemented to detect and interpret traffic signs in the simulated environment. It can effectively detect caution and stop signs, and form an appropriate response before crossing the intersection.

5.9 Reinforcement Learning for Obstacle Avoidance

Reinforcement Learning (RL) was applied to improve obstacle avoidance beyond classical PID methods. A PPO agent was trained in the Duckietown simulator with randomized obstacles.

- Environment: Duckietown simulator (`loop_obstacles.yaml`) with varying obstacle layouts.
- Input: RGB camera images ($84 \times 84 \times 3$).
- Output: Continuous actions $[v, \omega]$ (linear and angular velocity).
- Reward: Reward function is defined as follows:
  $+1$ for forward progress.
  -1 for collisions or leaving lane.
  Small penalty for sharp steering.
- Training: PPO with CNN-based policy (Stable-Baselines3), updated after collecting rollouts (`duckie_avoid_env.py`).
- Results: Agent avoided obstacles and maintained lane-following, showing better adaptability than PID in unseen cases. Training required long runtimes.
- Limitations: High variance, long training time, and challenges in sim-to-real transfer.

5.10 Reinforcement Learning: Experimental Analysis

We trained a PPO agent in the Duckietown simulator for about 20 iterations ($\sim$20k timesteps, 378s, 54 FPS).
The following observations were made:

Table 1: Key Training Metrics (PPO, 20 iterations)

Metric	**Value**
KL Approximation	0.071
Clip Fraction	0.391
Entropy Loss	-2.6
Explained Variance	-0.362
Learning Rate	0.0003
Policy Gradient Loss	-0.0673
Value Loss	0.0109
Std (performance variance)	0.885

- KL Approximation and clip fraction currently indicate large policy updates, reflecting instability in early training.

- Entropy loss shows that exploration is reducing, but further tuning is required for stable convergence.
- Negative explained variance highlights that the value function is not yet reliable, showing the limits of the current simulator-trained model.
- High variance indicates the policy is still highly exploratory and not yet stable, which results in partial success in lane following but frequent failures. Future work will focus on achieving more consistent policies, reducing learning rates, and extending training to improve robustness.

Overall, the agent demonstrated emerging lane-following behavior but lacked consistency. Future improvements will include lowering the learning rate, tuning clipping parameters, extending training, and refining the reward function.

5.11 Training Stability Analysis

After the initial 20k-step experiment, we extended the PPO training with different learning rates to study their effect on stability and convergence. The following figures show training curves for two runs:

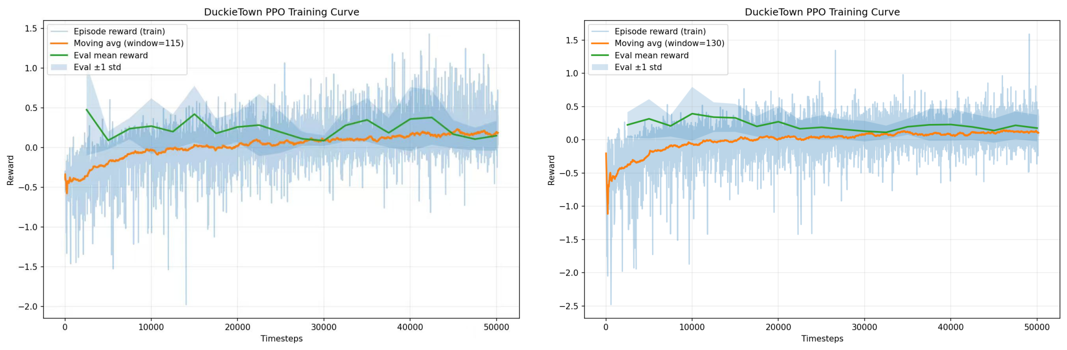

Figure 14: PPO training reward curves with learning rate 3*e*-4 (left) and 1*e*-4 (right).

The plots show four curves:

- **Episode reward (blue line)**: Raw reward per episode, showing fluctuations due to exploration.
- **Moving average (orange line)**: Smoothed reward over time, indicating long-term performance trend.
- **Evaluation reward (green line)**: Agent performance in a fixed environment without randomness.
- **Shaded area**: ±1 standard deviation of evaluation reward, reflecting policy stability.

Findings:

- With higher learning rate (3e-4), the agent learns faster but suffers from instability and reward spikes.
- With reduced learning rate (1e-4), the training is slower but smoother, and evaluation rewards are more consistent.

These results confirm that adjusting hyperparameters such as learning rate significantly impacts training stability and final performance. Our next steps will include longer training durations with the stable configuration, further reward shaping, and transfer to real Duckiebot.

6 Experimental Results

- PID-based lane keeping: Lane following works reliably on straight roads and smooth curves. Turn awareness improves stability in sharp corners.
- Traffic light recognition: Traffic light recognition successfully stops at red and proceeds at green.
- Traffic sign recognition: Traffic signs at intersections are correctly detected and interpreted, with appropriate velocity changes for each variant.

7 Challenges and Limitations

During the development of autonomous driving functions, several challenges arose. Lane markings were sometimes only partially visible or distorted under varying conditions, making stable lane keeping difficult. Moreover, all modules needed to operate in real time within a modular framework to allow smooth integration of perception, control, and decision layers. Addressing these issues required robust computer vision pipelines, carefully tuned controllers, and an integrated framework that allows real-time coordination.
In particular, obstacles on the track were especially difficult to perceive correctly due to their similarities to many other environment elements, and the inability to perceive 3d space made obstacle avoidance difficult. In the future, we hope to correct this problem by implementing reinforcement learning, so that the obstacle avoidance model can eventually learn to correctly and reliably identify obstacles.

Although our PPO-based reinforcement learning agent has shown a preliminary trend toward convergence in simulation, it is not yet capable of consistent and reliable obstacle avoidance. The learned policy still lacks stability and robustness. Further training iterations, hyperparameter tuning, and environment randomization are required to improve generalization and real-world transferability. This represents an early but promising step toward data-driven decision making for obstacle avoidance.