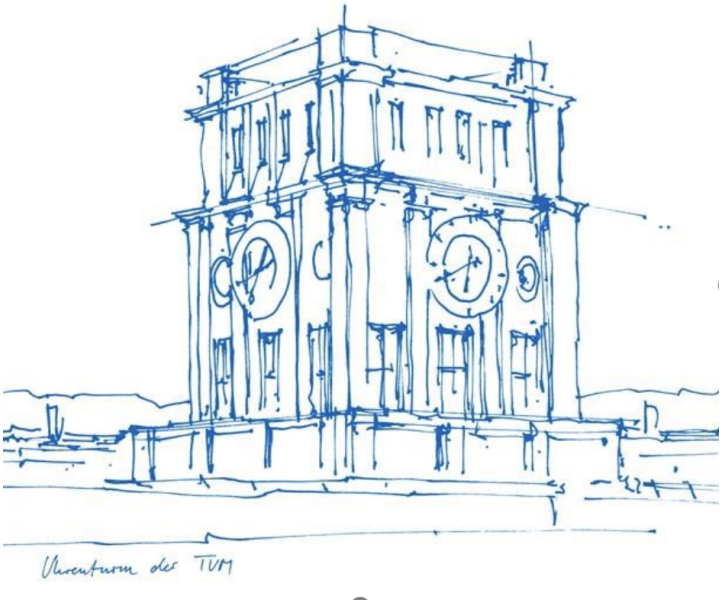
SCAN ME

DuckieTown Practical Course

Mohamed Amine Jradi, Abdelmoniem Abdelkhalek

Project Report: Practical Course
Bachelor of Information Engineering
Technical University of Munich



Professor
Amr Alanwar

Submitted on
18.03.2025

Contents

Abstract—

This project, developed for the Practical course at the Technical University of Munich, focuses on creating an autonomous driving vehicle. Autonomous navigation in constrained environments is a fundamental challenge in robotics, requiring precise perception, control, and decision-making. This project explores Duckietown, a scaled-down autonomous driving ecosystem designed for experimentation and education in robotics and AI. We developed and implemented a robust perception and control pipeline enabling a Duckiebot to autonomously navigate a lane, detect obstacles, and perform dynamic maneuvers such as lane switching and U-turns.

The system integrates computer vision techniques for lane detection, vehicle identification, and distance estimation, utilizing OpenCV and ROS. A reactive collision avoidance mechanism was implemented, allowing the Duckiebot to dynamically adjust speed and position based on detected objects. Additionally, service-based architecture was used to toggle between lane following and avoidance behaviors.

This work contributes to the field of autonomous robotics by showcasing a modular and extensible approach to self-driving behavior in miniature city environments.

1. Problem Statement

Autonomous vehicles rely on robust perception and control mechanisms to navigate safely in dynamic environments. In the Duckietown project, a miniature autonomous vehicle (Duckiebot) must follow lanes, avoid obstacles, and comply with traffic rules in a structured urban setting. However, achieving reliable lane following and navigation presents several challenges:

- Variability in environmental conditions (e.g., lighting changes, shadows, and occlusions) can affect lane detection accuracy.
- Hardware limitations in processing power and sensor accuracy constrain real-time decision-making.
- Dynamic obstacles such as other Duckiebots or unexpected objects require quick adaptations in trajectory.
- Noise in sensor data can lead to errors in perception, causing the Duckiebot to deviate from its intended path.

This project aims to develop a vision-based lane-following system that ensures the Duckiebot can navigate autonomously within its environment. By optimizing computer vision algorithms, adjusting control thresholds, and integrating real-time feedback mechanisms, we seek to improve the robustness and adaptability of the Duckiebot's navigation.

2. Literature Review

The field of autonomous driving and robotics education has seen significant advancements, with simulation-based and large-scale testing environments often being perceived as inaccessible or complex for beginners and researchers in academia. This has led to the development of platforms like Duckietown, which provide a low-cost, hands-on environment for experimenting with autonomous navigation and reinforcement learning techniques. Existing literature on Duckietown highlights its role in bridging the gap between theoretical machine learning models and real-world implementation, allowing students and researchers to test perception, planning, and control algorithms on miniature autonomous vehicles. While computer vision-based lane-following systems demonstrate reliable performance in structured environments, challenges arise in handling dynamic obstacles, lighting variations, and real-time decision-making. The integration of ROS (Robot Operating System), reinforcement learning, and classical control techniques forms the backbone of Duckietown's approach, enabling robust lane-keeping and multi-agent coordination. The modular nature of Duckietown further enhances its flexibility, allowing for extensions in areas such as swarm robotics, multi-agent cooperation, and human-robot interaction. As autonomous driving technology continues to evolve, Duckietown serves as a valuable tool for education and research, fostering innovation in the field of self-driving vehicles.

3. Setup

- Build the bot
- Flash the SD Card
- Calibrate the Camera
- Calibrate the wheels

4. Lane Following

The fundamental approach for the bot to follow the road is by following the lane. In our case we found two approaches and we chose the one that worked better. Both approaches work by first processing the image then detecting the yellow line, then correcting the path of the bot so it says on the road but the difference is in the error correcting.

4.1. image processing

The robot subscribes to a compressed image topic to receive frames from the camera. It uses OpenCV to process the images, detect yellow and red lines, and calculate errors based on the position of the yellow lane.

```
# Subscribers & Publishers
    self.sub_image = rospy.Subscriber("
        csc22938/camera_node/image/
        compressed", CompressedImage, self.
        process_image)
    self.pub_cmd = rospy.Publisher("
        csc22938/joy_mapper_node/car_cmd",
        Twist2DStamped, queue_size=1)
```

```
def process_image(self, msg):
        img = self.bridge.
            compressed_imgmsg_to_cv2(msg)
```

4.1.1. Yellow Line detection. The robot searches for yellow lanes using color thresholding in the HSV color space. If the robot detects a yellow lane, it calculates the center of mass of the detected lane and computes the error as the deviation from the center of the image.

```
    # Lane Following Parameters
    self.yellow_lower_bound = np.array([20,
        45, 25])
    self.yellow_upper_bound = np.array([35,
        255, 255])
```

4.1.2. Red Line detection. When the robot detects a red line, it triggers the stopping behavior, halting the robot's movement. The red lines are detected using color thresholds and contour detection. Second page progress

```
    # Red line Parameters
    self.red_lower_bound1 = np.array([0,
        100, 100])
    self.red_upper_bound1 = np.array([10,
        255, 255])
    self.red_lower_bound2 = np.array([160,
        100, 100])
    self.red_upper_bound2 = np.array([180,
        255, 255])
```

4.2. Methodologies

To adjust the bot movement while following the yellow line, we encountered two solutions and we implemented with what worked best.

4.2.1. First approach. This approach directly adjusts the movement (speed and angular velocity) based on the simple error calculation. This approach is more reactive and simpler, without any accumulated error or time-based adjustments.

```
# Calculate the errors based on the
    processed image
        error = self.calculate_error(img)
        if(abs(error) < 0.4):
            self.target_v = UP[0]
            self.target_omega = UP[1]
        elif(error > 0):
            self.target_v = LEFT[0]
            self.target_omega = LEFT[1]
        elif(error < 0):
            self.target_v = RIGHT[0]
            self.target_omega = RIGHT[1]
```

4.2.2. Second approach. A PID controller with proportional (K_p), integral (K_i), and derivative (K_d) gains can be used to control the robot's movement more smoothly. It adjusts both the speed (v_{error}) and angular velocity (ω_{error}) based on error values over time. The control output is given by:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d \frac{d}{dt} e(t)$$

Where: - $u(t)$ is the control output (either speed or angular velocity), - $e(t)$ is the error at time t (difference between the desired and actual values), - K_p, K_i, and K_d are the proportional, integral, and derivative gains, respectively.

```
    error = self.calculate_error(img)
    omega = self.omega_controller.control(
        error)
    v = self.base_speed
    self.control_wheels(v, omega)
```

5. AprilTag detection

In this project, we use AprilTag Detection to help our Duckiebot recognize traffic signs like STOP, LEFT TURN, and RIGHT TURN. The robot detects an AprilTag, interprets its ID, and adjusts its movement accordingly. AprilTags are like QR codes for robots. They provide unique IDs that the Duckiebot can recognize and use for decision-making Each AprilTag has a unique ID linked to a specific action.

Figure 1: AprilTag

5.1. image processing

Once the camera captures an image, we:

1) Convert it from compressed format to an OpenCV image.
2) Convert it to grayscale for easier AprilTag detection.

```
data_arr = np.frombuffer(msg.data, np.
    uint8)
col_img = cv2.imdecode(data_arr, cv2.
    IMREAD_COLOR)
grey_img = cv2.cvtColor(col_img, cv2.
    COLOR_BGR2GRAY)
```

5.2. AprilTag detection

Detects AprilTags using an OpenCV-based detector, enabling the robot to identify and localize itself based on the tags' positions.

```
tags = self.detector.detect(gray,
    True, self.camera_parameters,
    self.tag_size)
```

Once a tag is detected, its ID is used to classify the color (or type) of the tag (e.g., blue for intersections, red for stop signs). The LED color is updated and a command will be sent to the wheels to move based on the recognized tag ID.

```
self.twist_publisher.publish(msg)
```

If tags are detected, the node processes their data and controls the robot's behavior accordingly. If no tags are found, the robot continues its previous behavior (e.g., lane following). The system uses LED color feedback to indicate the detected tags to the user.

```
def sign_to_col(self, id):
    if(id in self.sign_col_map):
        return self.sign_col_map[id]
    else:
        print("[INFO] Recognized
            AprilTag - but the ID is
            not valid.")
        return "WHITE"
```

5.3. Our approach

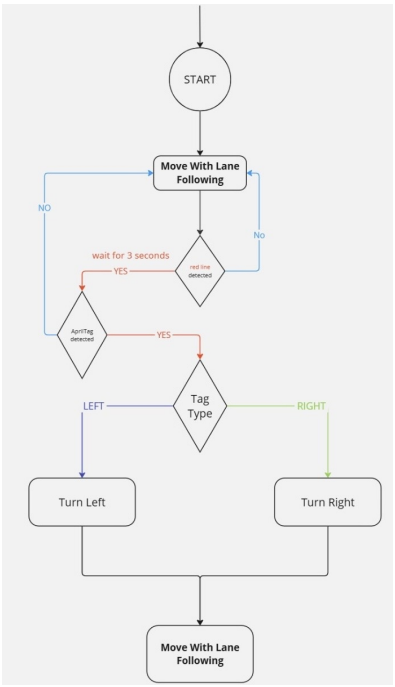


Figure 2: scenario

Using a state machine, we switch between the states, so the bot can know which node to activate and which action to proceed.

6. Collision Avoidance

The Collision Avoidance System in our Duckiebot works by detecting obstacles (other Duckiebots), estimating their distances, and executing an avoidance . The system processes camera images, recognizes predefined patterns, and dynamically adjusts the bot's movement to prevent collisions.

6.1. Duckiebot detection

The Duckiebot must identify other robots on the road to avoid collisions. Since it operates in an environment with multiple robots, detecting nearby bots is essential for safe navigation.

The detection relies on OpenCV's findCirclesGrid(), which locates a 7×3 circle grid pattern commonly found on Duckiebots.

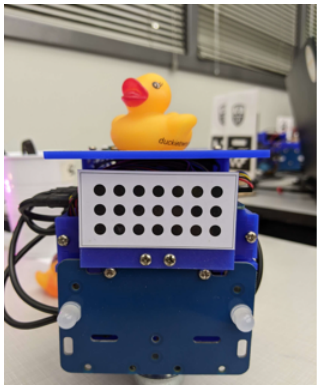


Figure 3: grid pattern

6.2. Distance estimation

Once a Duckiebot is detected, the system estimates how far it is. The bot adjusts its speed based on this distance. We use SolvePnP() method that estimates the position and orientation of the detected object relative to the camera.

```
    def estimate_distance(self, image_msg,
        image_cv, centers):
        try:
            H = 3
            W = 7
            self.calc_circle_pattern(H, W)
            points = np.zeros((H * W, 2),
                dtype=np.float32)
            for i in range(len(centers)):
                points[i] = np.array([
                    centers[i][0][0],
                    centers[i][0][1]])

            success, rotation_vector,
                translation_vector = cv2.
                solvePnP(
                objectPoints=self.
                    circlepattern,
                imagePoints=points,
                cameraMatrix=self.pcm.
                    intrinsicMatrix(),
                        distCoeffs=self.
                    pcm.distortionCoeffs(),
```

6.3. the U-turn

When a detected obstacle is too close, the bot executes a side-step maneuver instead of stopping completely. In our current implementation, it does a 90-degree lane-switching U-turn, which is a hardcoded solution.

1) If an obstacle is far away, the bot slows down.
2) If an obstacle is too close, the bot performs a U-turn.
3) This maneuver consists of predefined turns and forward movements.

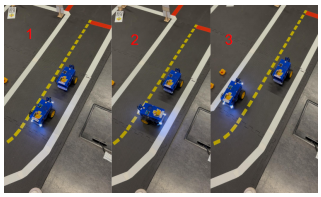


Figure 4: 90DegreeCollisionAvoidence

7. Challenges

7.1. FirstChallenge

In our current collision avoidance system, we implemented a hardcoded 90-degree lane-switching solution when a Duckiebot or obstacle is detected too close. However, this approach is not optimal for real-world navigation, as not all obstacles require a full 90-degree turn. Instead, the Duckiebot should slow down and go with an angle rather than making turns, similar to real-life driving, where vehicles adjust their trajectory smoothly. To improve adaptability, the bot should dynamically adjust its path based on the obstacle's position rather than following a fixed path. By integrating depth estimation (SolvePnP) with real-time object detection (YOLO) for future improvements, the system can make smarter decisions based on the size, distance, and type of obstacles. This enhancement allows for a more efficient, flexible, and realistic collision avoidance strategy, enabling the Duckiebot to navigate complex environments safely.

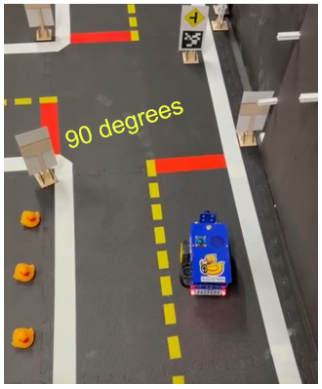


Figure 5: 90degree turn

7.2. SecondChallenge

In our current system, all intersections in the Duckiebot city map are designed as predefined 90-degree turns. This structured approach allows the bot to follow a fixed movement pattern, ensuring it executes precise left or right turns at intersections. However, in real-world navigation, intersections are not always perfect 90-degree angles, and roads can have irregular shapes or curves. Relying solely on fixed-angle turns limits adaptability and may cause inefficient navigation in more complex environments. To enhance flexibility, the system should incorporate dynamic path planning, allowing the Duckiebot to adjust its turning angle based on the road layout and detected intersections. This would enable smoother transitions and more efficient movement in real-world driving condition

8. Future improvements

8.1. Intelligent Graph-Based Navigation for Autonomous Ride Requests

A significant future improvement to our system is enabling autonomous ride-hailing similar to services like Uber, where a Duckiebot can efficiently navigate between pickup and drop-off locations.

Currently, our implementation establishes a client-server model, allowing a user to request a ride from Point A to Point B. However, the key challenge lies in making a memoryless Duckiebot effectively store, recall, and process waypoints and destinations dynamically.

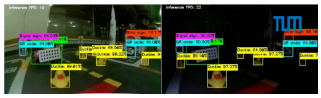


Figure 6: YOLO object detection

To address this, we will need propose a graph-based real-time navigation system, where the city is represented as a graph of nodes (intersections) and edges (streets). The server will continuously update this graph, providing the Duckiebot with optimized routes. By integrating real-time location tracking and dynamic path planning, this system could pave the way for an autonomous ride-sharing network, allowing multiple bots to operate seamlessly within a smart city.

8.2. Enhancing Obstacle Detection with YOLO-Based Object Recognition

Currently, our system detects only Duckiebots using SolvePnP, which relies on a fixed 7x3 circle grid pattern. This approach limits its detection to specific objects while ignoring other obstacles such as walls, pedestrians, and random objects, reducing the bot's ability to navigate safely. To improve this, we propose integrating YOLO-based object detection, which allows the system to identify multiple object types (vehicles, pedestrians, obstacles) in real time, enhance collision avoidance by recognizing and responding to a broader range of obstacles, and improve navigation adaptability, making the Duckiebot more efficient in complex environments. By incorporating YOLO, our bot can transition from pattern-based detection to general object recognition, ensuring a more intelligent and versatile obstacle avoidance system.

9. Conclusion

In this project, we developed an autonomous Duckiebot capable of lane following, AprilTag-based navigation, and collision avoidance. The bot efficiently follows lanes using image processing and PID control, stopping at red lines to detect AprilTags for directional decisions. Our system currently detects only Duckiebots using SolvePnP with a fixed 7x3 grid, limiting obstacle detection.

To improve adaptability, we propose integrating YOLO-based object detection to recognize multiple obstacles and enhance navigation. Additionally, transitioning from hard-coded 90-degree turns to dynamic, angle-based avoidance will make the bot more realistic. Future enhancements include memory-based path planning for improved autonomy, creating a foundation for real-world robotic navigation. This project demonstrates the potential for scalable and intelligent autonomous systems.