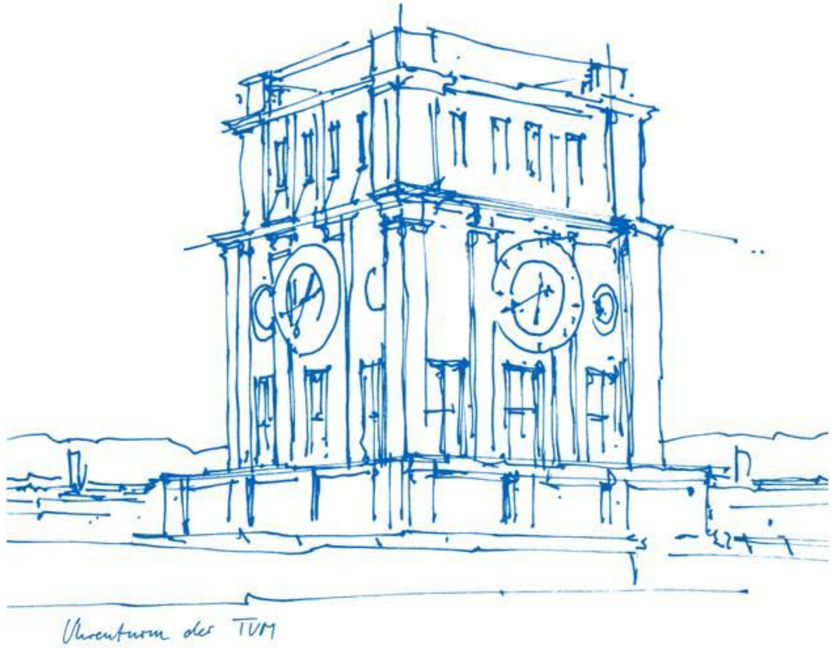
Practical Course: Duckie-Town

Group D
Mohamed Amine Jradi, Abdelmoniem Abdelkhalek

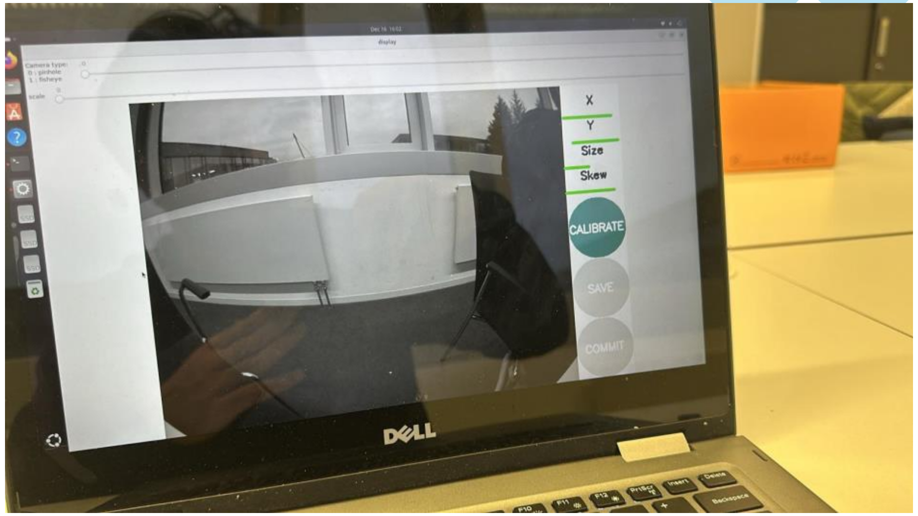
Technical University of Munich
TUM School of Computation, Information and Technology
Campus Heilbronn, 18. March 2025

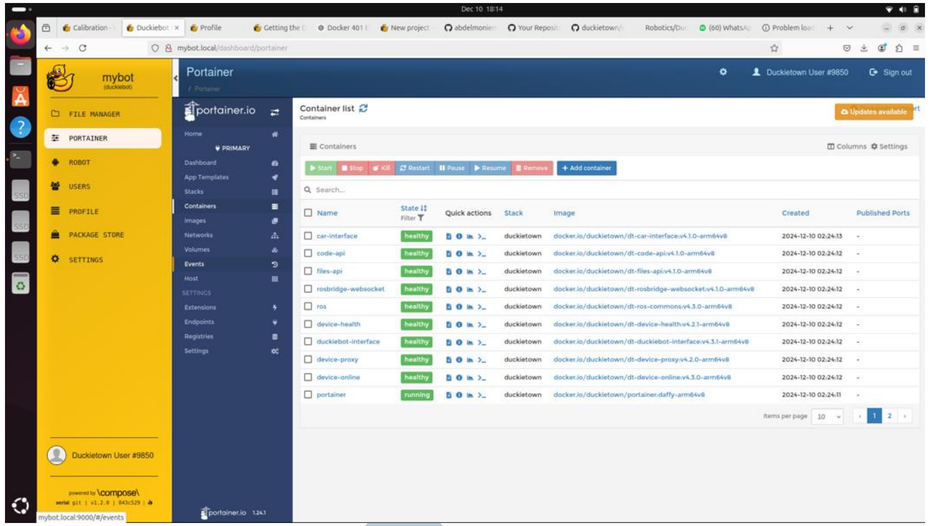


- Setup
- Lane Following
- AprilTag detection
- Collision Avoidance
- Challenges

Setup

- **Build the Bot**
- **Set up the SD Card**
- **Calibrate the wheels and camera**
- **Get familiar with how the bot works**





Lane Following

Methodologies

First Approach

- **Predefined Movement (UP, LEFT, RIGHT)** :

- Instead of PID control, this approach uses **predefined movement commands** based on **lane detection**.

- There is a certain threshold that should not be exceeded.
- The bot **divides the lane into three zones**:
  - **Centered** → Moves **straight (UP)**.
  - **Lane detected more on the right → Turns left (LEFT).**

  - **Lane detected more on the left → Turns right (RIGHT).**

→ It didn't work accurately

```
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]
```

Second Approach

- **PID-Controlled Lane Following:**

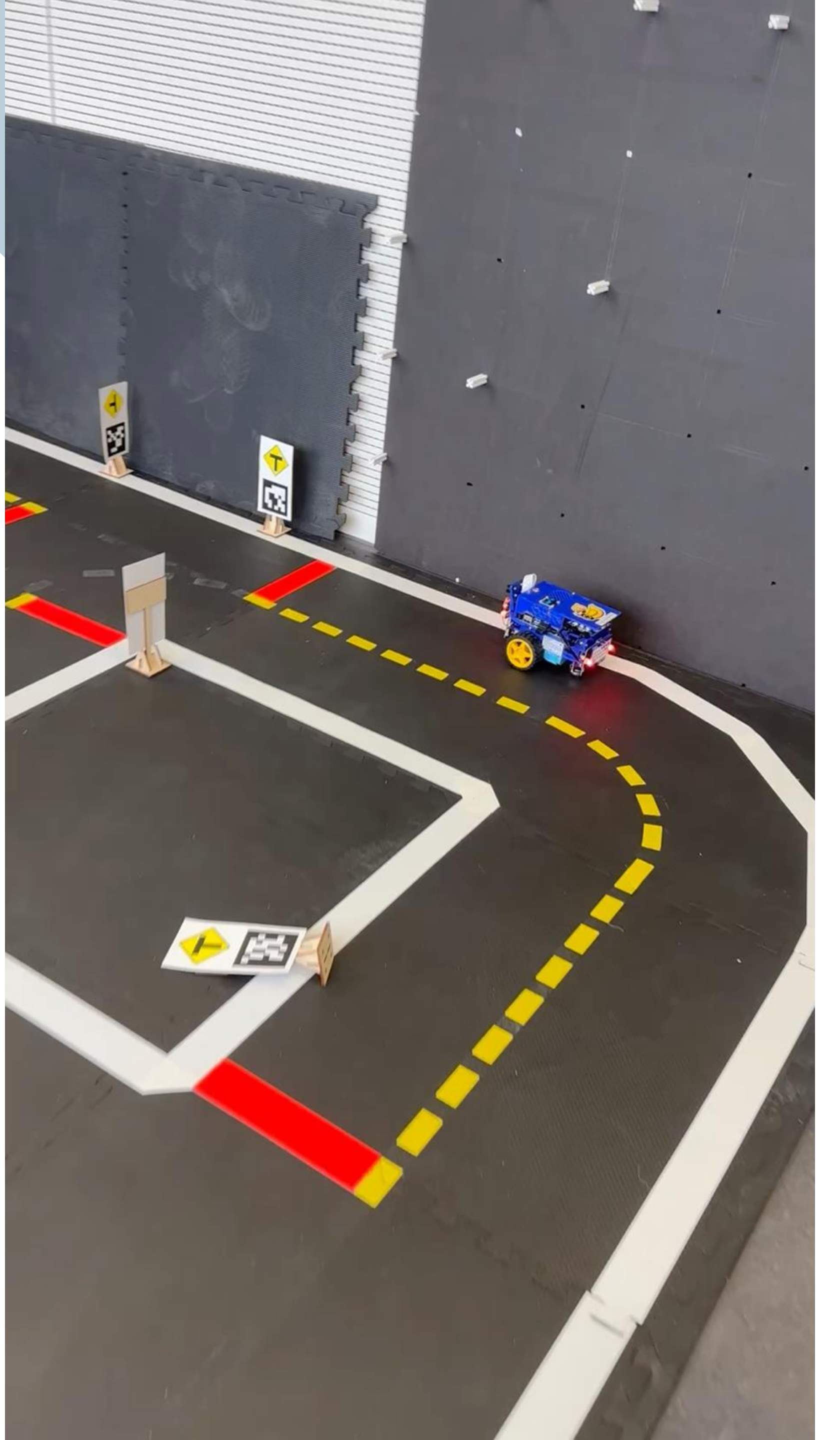
•The Duckiebot **continuously detects the yellow lane** using **HSV color filtering**.
•It calculates **how far off-center it is** (error value).

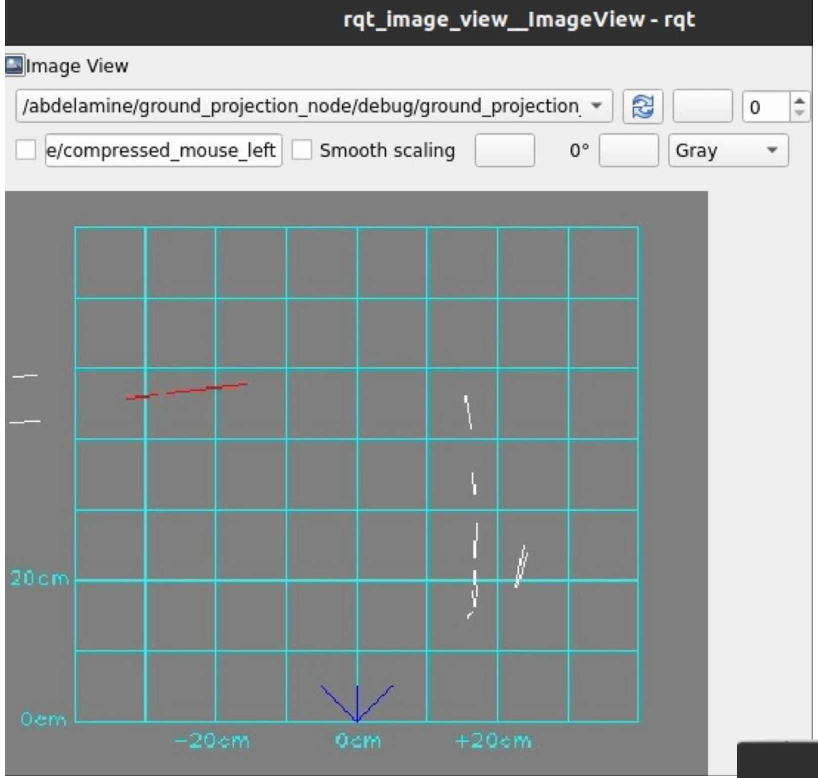
A **PID controller** adjusts the **steering angle** (**omega**) to bring the bot back to the center.

```
diff = ((x + int((self.size_ratio*y))) - goal) * scale_for_pixel_area
```

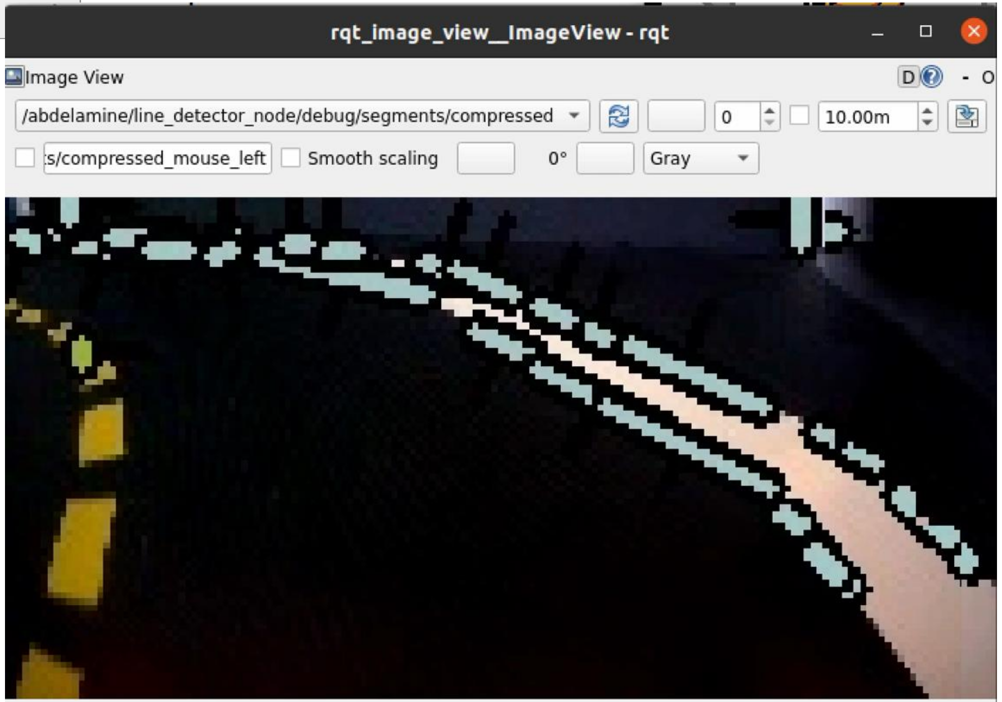
```
self.omega = -self.PID[0] * diff
```

```
def cb_img(self, msg):
    # get the image from camera and mask over the hsv range set in init
    data_arr = np.fromstring(msg.data, np.uint8)
    col_img = cv2.imdecode(data_arr, cv2.IMREAD_COLOR)
    crop = [len(col_img) // 3, -1]
    hsv = cv2.cvtColor(col_img, cv2.COLOR_BGR2HSV)
    imagemask = np.asarray(cv2.inRange(hsv[crop[0] : crop[1]], self.lower_bound, self.upper_bound))
```





rqt_image_view__ImageView - rqt
Image View
/abdelamine/ground_projection_node/debug/ground_projection
0
e/compressed_mouse_left
Smooth scaling
0°
Gray
20cm
0cm
-20cm
0cm
+20cm



rqt_image_view__ImageView - rqt
Image View
/abdelamine/line_detector_node/debug/segments/compressed
0
10.00m
s/compressed_mouse_left
Smooth scaling
0°
Gray

AprilTag detection

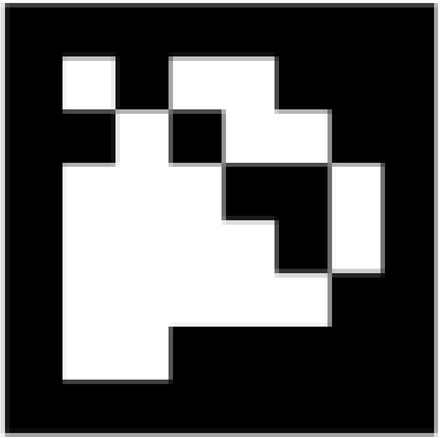
AprilTag

- 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.



Camera Data Processing

```
def cb_img(self, msg):
    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)
```

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.

Detecting AprilTags

```
def detect_tag(self):

    # convert the img to greyscale
    img =  self.col_img
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

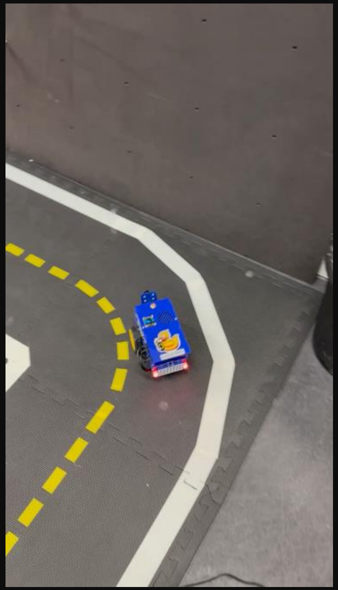
This gives us list of tags which the bot detected

Code Execution When a Left Turn Sign (AprilTag) is Detected

- The AprilTag detector scans the grayscale image for AprilTags
- The **detected tag's ID** is retrieved.
- If the tag matches the **left** turn ID, execution will proceed accordingly

- **Lane Following Node Reacts to the Left Turn Sign**

- Increases **omega (steering angle)** to make a left turn.



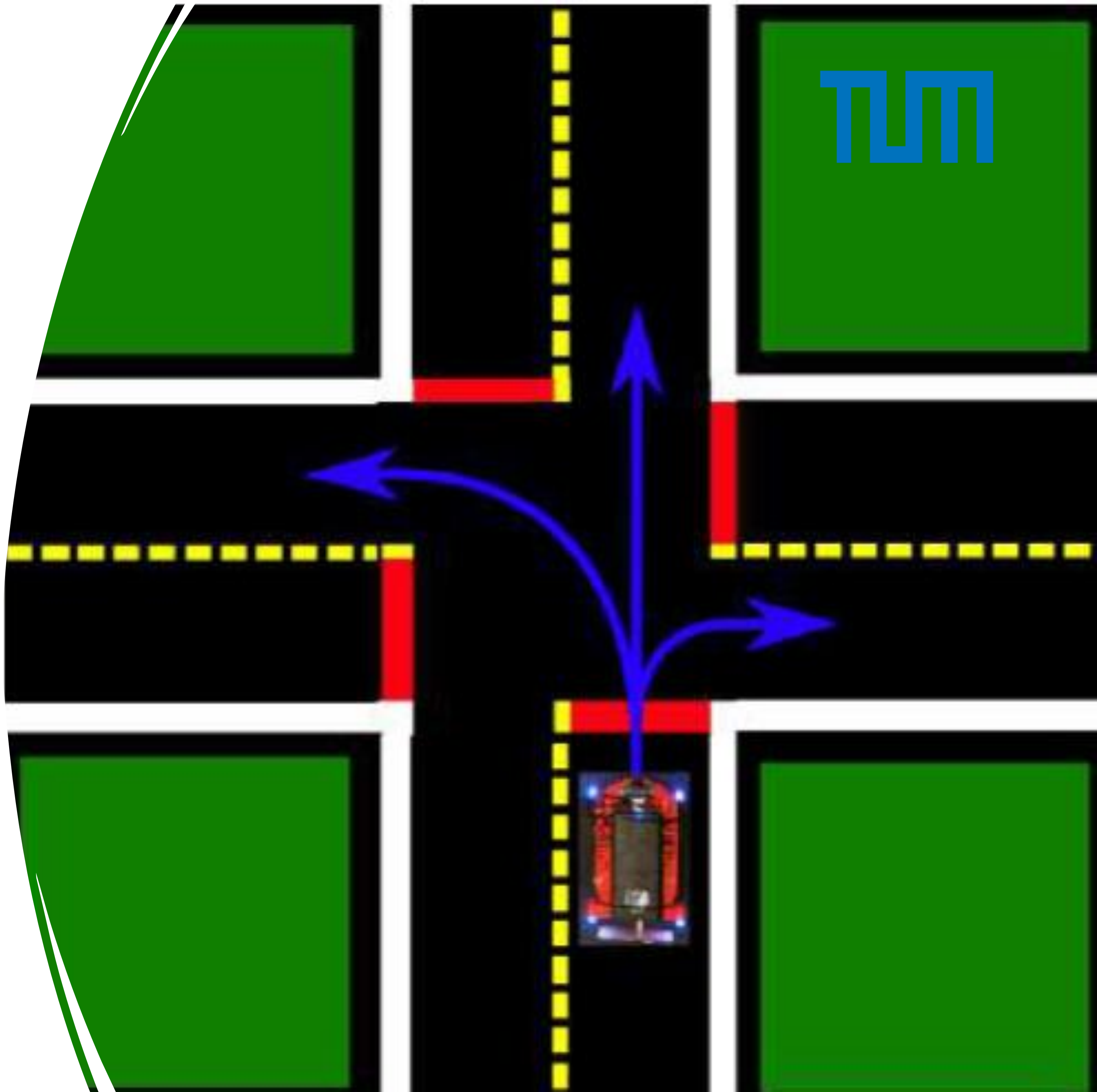
```python
if tag_color == "LEFT":
    print("Turning left!")
    self.send_drive_command(0.2, 2.0)  # Adjust speed and omega for turning left
    rospy.sleep(1.5)  # Sleep to allow turn
    self.send_drive_command(0.2, 0.0)  # Move forward after turn

elif tag_color == "RIGHT":
    print("Turning right!")
    self.send_drive_command(0.2, -2.0)  # Adjust speed and omega for turning right
    rospy.sleep(1.5)  # Sleep to allow turn
    self.send_drive_command(0.2, 0.0)  # Move forward after turn
```

Our Approach

red line detection and AprilTag-based decision-making

1. Stop at Red Line

- When the Duckiebot detects a **red line at an intersection**, it **stops for 3 seconds**.



Our Approach

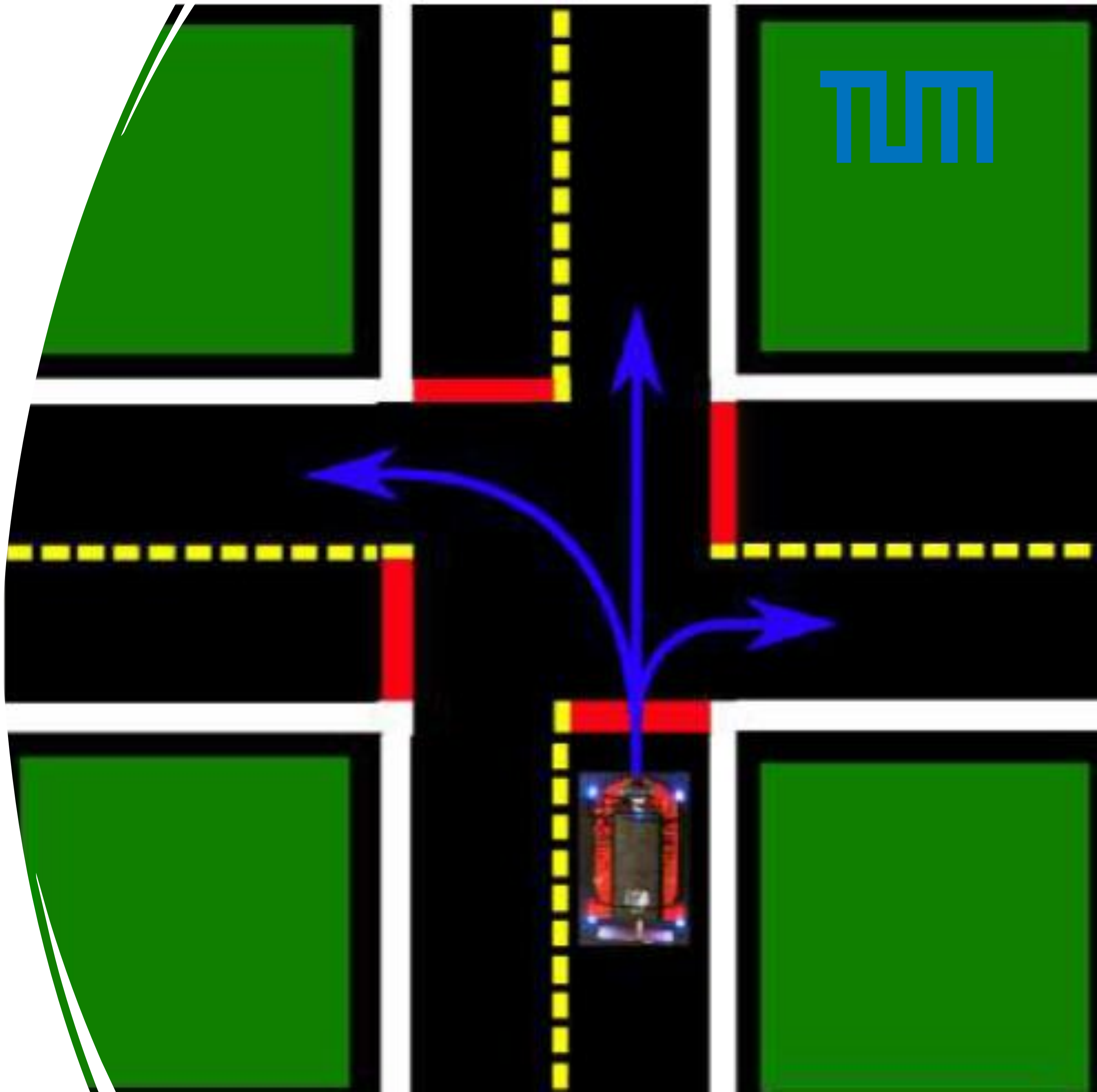
red line detection and AprilTag-based decision-making

1. Stop at Red Line

- When the Duckiebot detects a **red line at an intersection**, it **stops for 3 seconds**.

2. Detect & Analyze AprilTag

- After stopping, the bot **activates AprilTag detection** to determine the next move.



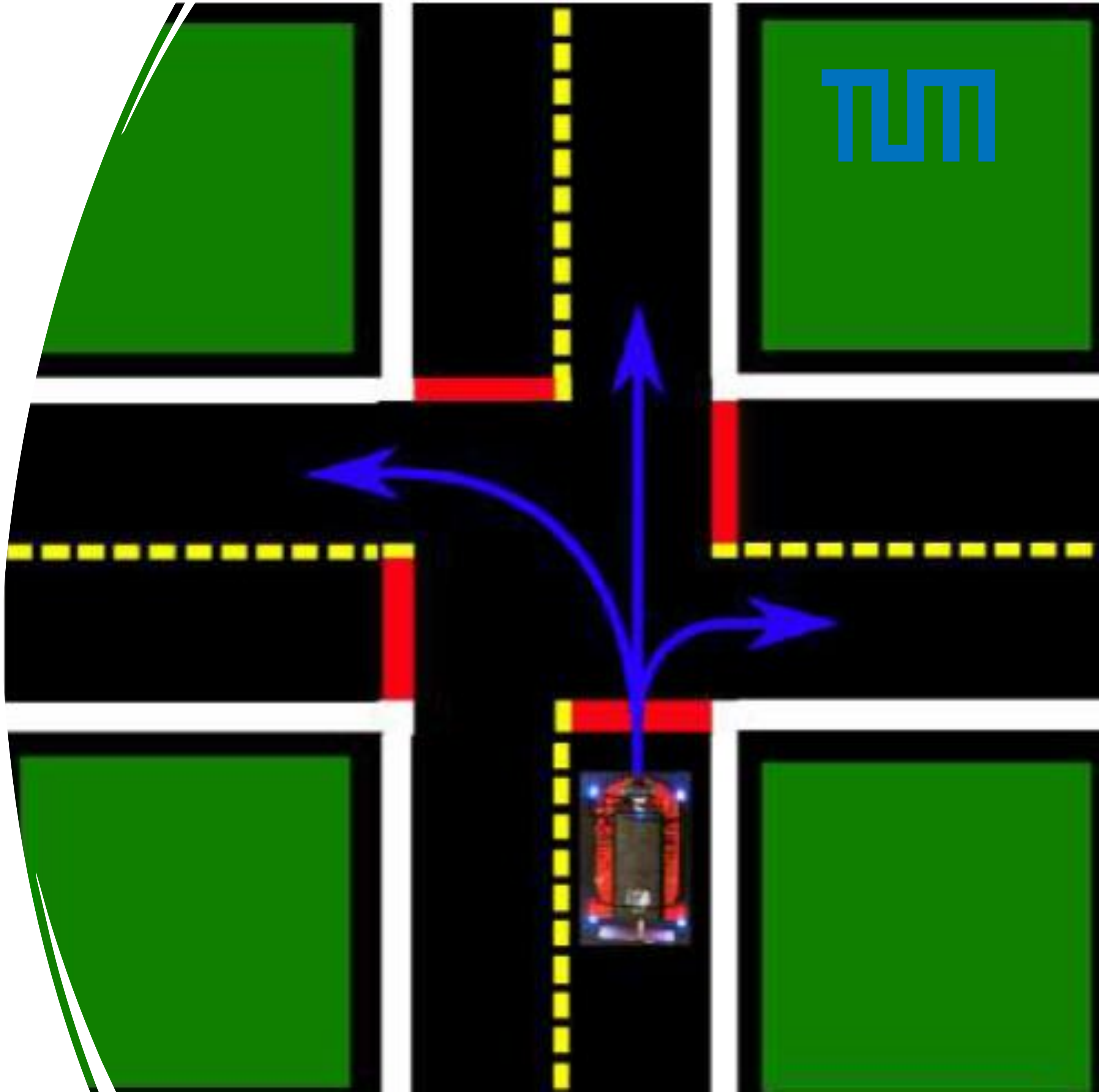
Our Approach

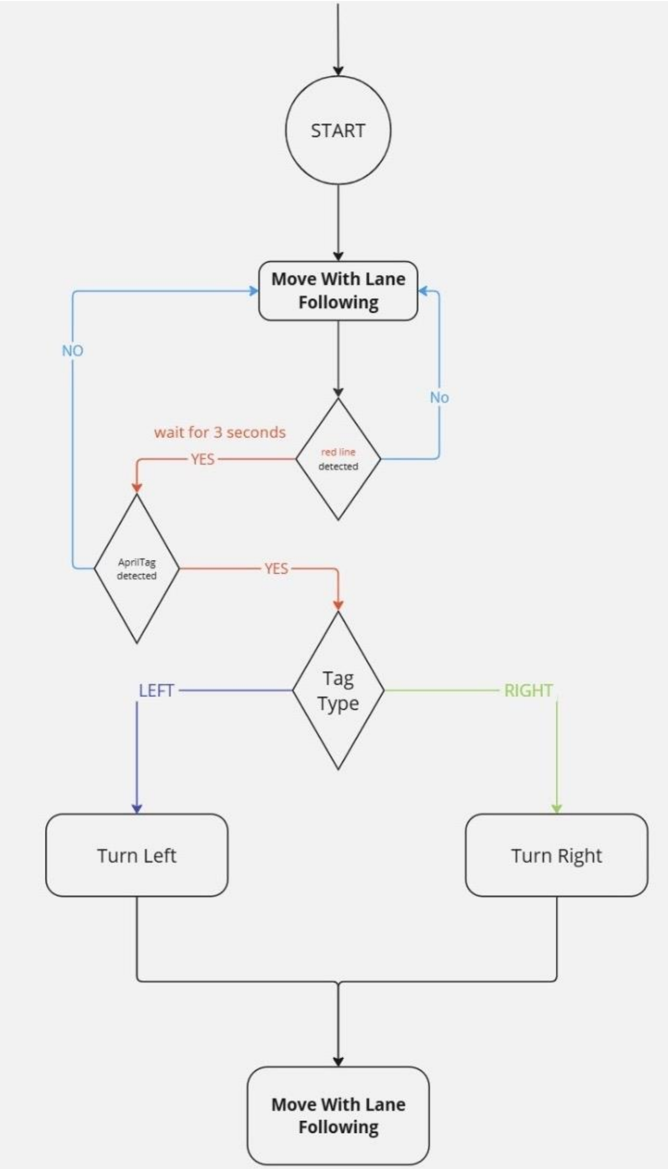
red line detection and AprilTag-based decision-making

3. Turn Based on AprilTag ID

- If the detected **AprilTag indicates a turn**, the bot **executes the precise turn**, if not it continue with the lane following.

- Since our city layout consists of **only 90-degree intersections**, we can confidently apply a **fixed-angle turning strategy** to ensure smooth and accurate navigation.




START
Move With Lane Following
NO
No
wait for 3 seconds
YES
red line detected
AprilTag detected
YES
Tag Type
LEFT
RIGHT
Turn Left
Turn Right
Move With Lane Following

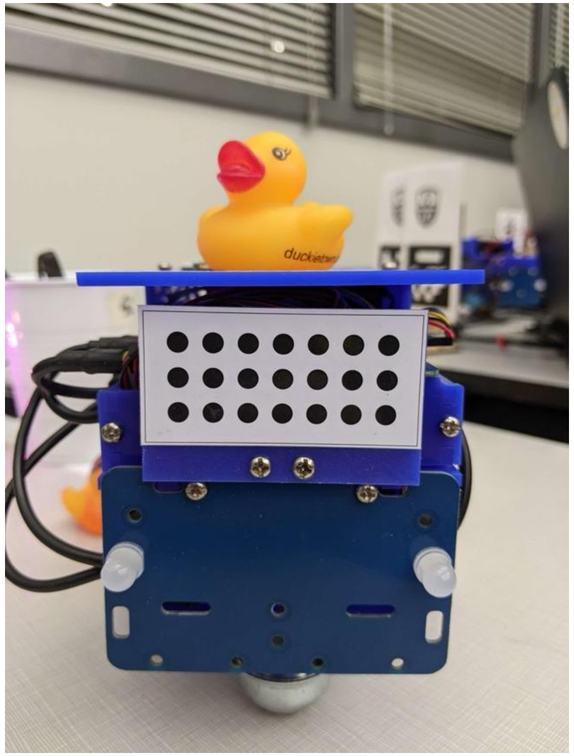
Collision Avoidance

Duckiebot Collision Avoidance System

- **Detects Duckiebots** using a **7x3 circle grid pattern** with **OpenCV's** `findCirclesGrid()`.

- **Estimates distance** using `solvePnP()`, calculating the position and orientation relative to the camera.

- If a Duckiebot is directly in front and too close, the system **executes a U-Turn**, allowing smooth movement **around obstacles while staying on course**.



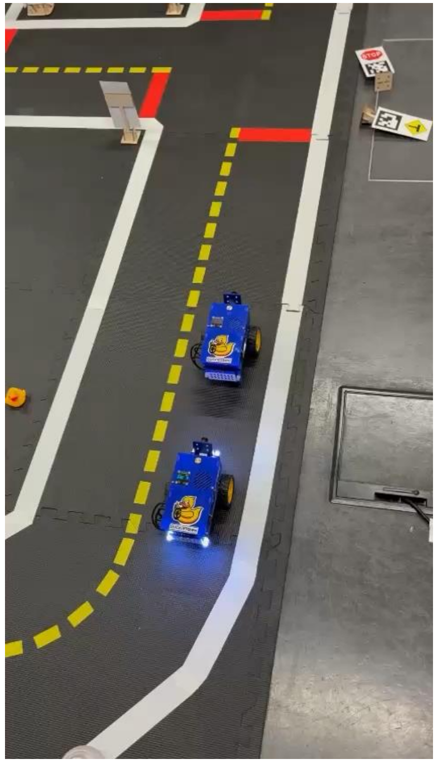
Duckiebot Collision Avoidance System

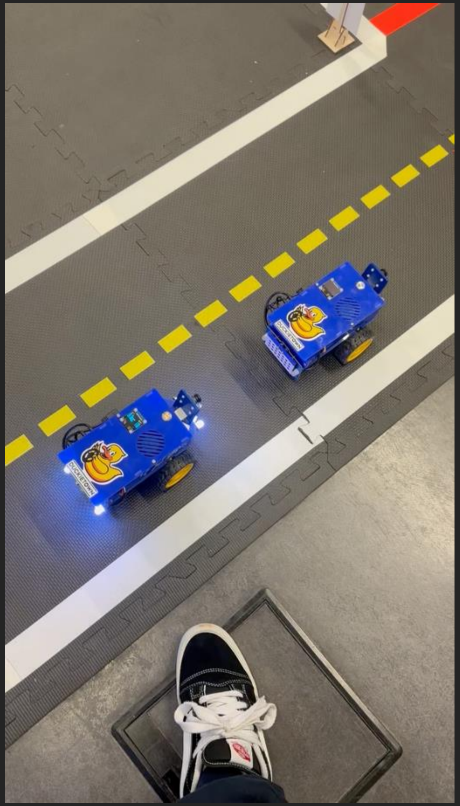
```
if distance is None:
    #No robot in sight, full speed!
    self.control_wheels(0.2, 0)

elif distance <= self.stop_distance:
    #Too close, stop and perform obstacle avoidance maneuver.
    rospy.logwarn("Too close! Stopping and initiating obstacle avoidance.")
    self.control_wheels(0, 0)

    if not self.obstacle_avoidance_active: #Only start avoidance once
        self.obstacle_avoidance_active = True #Set flag
        self.avoid_obstacle() # Call the lane switching maneuver

elif distance <= self.min_distance_to_react:
    #Approaching, slow down.
    new_speed = 0.05#self.base_speed * self.slowdown_factor
    rospy.logwarn(f"Slowing down: Distance = {distance:.2f}, New Speed = {new_speed:.2f}")
    self.control_wheels(new_speed, 0) #slow

else:
    #Robot far away, proceed as planned.
    self.control_wheels(0.2, 0) #full speed!
    rospy.logwarn("No bot close by")
```

Duckiebot Collision Avoidance System

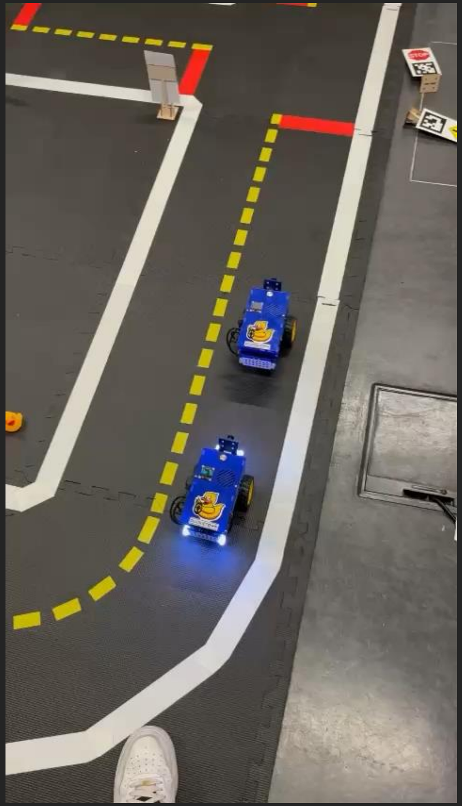
```python
def avoid_obstacle(self):
    """
    Performs a predefined lane-switching maneuver to avoid an obstacle.
    This is a simplified example using fixed durations and speeds.
    """
    rospy.loginfo("Obstacle avoidance maneuver started...")

    # Step 1: Move diagonally left to enter the parallel lane
    rospy.loginfo("Step 1: Moving diagonally left...")
    self.control_wheels(0.2, 0.7)  # Move slightly forward while turning left
    rospy.sleep(1.5)
    self.control_wheels(0.2, 0)  # Drive straight briefly
    rospy.sleep(1.5)

    # Step 2: Adjust to the lane by slightly turning right
    rospy.loginfo("Step 2: Adjusting to the lane...")
    self.control_wheels(0, -0.7)
    rospy.sleep(1.2)

    rospy.loginfo("Lane switch complete. Following new lane for a bit...")
    rospy.sleep(5)  # Follow lane for a while (in the 'other' lane)

    # Step 3: Perform a U-turn (right turn → left turn) to return
    rospy.loginfo("Step 3: Initiating U-turn...")
    self.control_wheels(0, -0.7)  # Turn right
    rospy.sleep(1.5)
    self.control_wheels(0, 0.7)  # Turn left
    rospy.sleep(1.5)

    # Step 4: Move diagonally right to return to original lane
    rospy.loginfo("Step 4: Moving diagonally right to return...")
    self.control_wheels(0.2, -0.7)  # Move slightly forward while turning right
    rospy.sleep(1.5)
    self.control_wheels(0.2, 0)  # Drive straight briefly
    rospy.sleep(1.5)

    # Step 5: Adjust to align with the lane
    rospy.loginfo("Step 5: Adjusting to align with original lane...")
    self.control_wheels(0, 0.7)
    rospy.sleep(1.2)
```

Failures





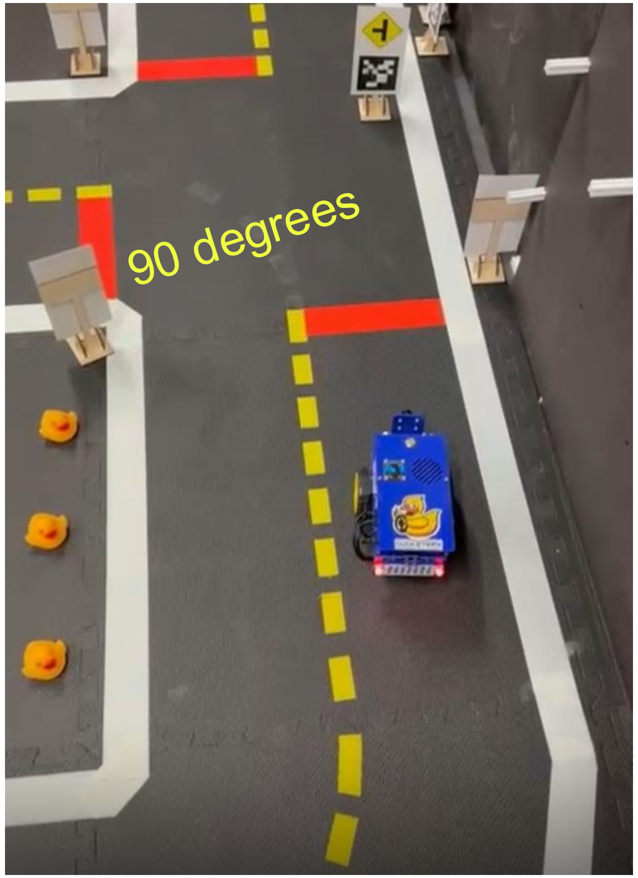
Success



Challenges

1- Our current collision avoidance system uses a hardcoded avoiding system which if any of the environment variables changes, avoiding obstacles will lead to failure.

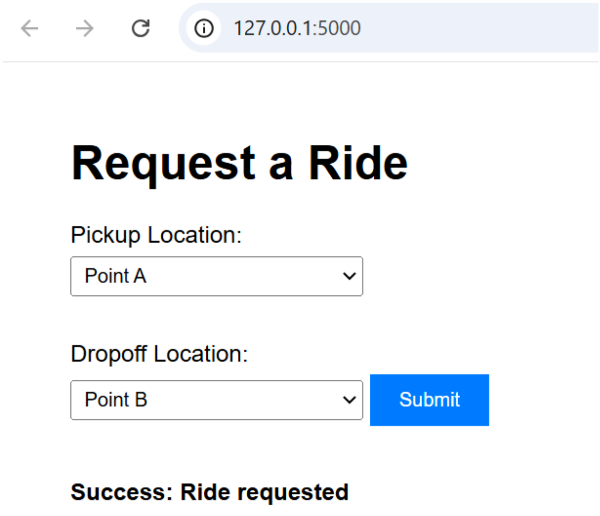
2- Our current system assumes all intersections are 90-degree turns as it is in the our duckiebot city map . The challenge is to make the Duckiebot dynamically adjust its turn angle based on real-time perception



Future Improvements

While we have successfully developed an interface that allows a client-server system to move from **Point to Point** , a critical challenge remains: **how does a memoryless bot efficiently store, recall, and process waypoints and destinations**

Graph-based real-time navigation: Treat the city as a graph where nodes (intersections) and edges (streets) are constantly updated through a server



Our system **only detects Duckiebots** using **SolvePnP**, which works with a **fixed 7x3 circle grid pattern**.

•This means **other obstacles (walls, pedestrians, random objects) are not detected**, limiting the bot's ability to avoid obstacles effectively.

So by using YOLO-Based Object Detection

•It can detect multiple objects of different types

(vehicles, pedestrians, obstacles) in a single image frame.

References

https://github.com/ekhumbata/Adventures-in-Duckietown

https://docs.duckietown.com/daffy/opmanual-duckiebot/setup/setup_laptop/index.html

SCAN ME

Thank you !