Duckietown Report

Julien-Alexandre Bertin Klein, Andrea Pellegrin, Fathia Ismail

18.03.2025

Abstract — This report presents the work of our team within the Duckietown project, focusing on the development of the Duckiebot's perception, navigation, and control systems. Through extensive testing and optimization, we have established a framework that enables the Duckiebot to navigate autonomously with precision and reliability.

1 Introduction

The Duckietown project is an open-source platform for research and education in autonomous systems, offering a cost-effective and scalable environment for developing and testing self-driving algorithms. At its core is the Duckiebot, a small autonomous robot that navigates a structured urban-like setting using only onboard sensors and computer vision.
This report presents the work carried out over the past six months to enhance the Duckiebot's perception, navigation, and decision-making capabilities. The primary objectives were to improve its autonomous navigation by implementing algorithms that ensure safe and efficient movement within the city, as well as to enhance its visual perception through advanced computer vision techniques.
The report is structured as follows: first, a brief overview of the hardware components is provided, followed by an examination of the preconfigured system, including Ubuntu, ROS, and the foundational framework provided by Duckietown. Next, we detail the features implemented during development and discuss the emphasis placed on autonomous driving safety, a key focus throughout the project. Finally, we conclude by summarizing the progress made, outlining the challenges encountered, and proposing potential improvements for future development.

2 Hardware Overview

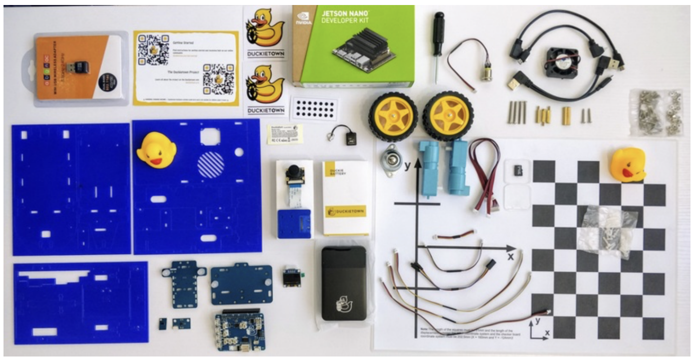


Figure 1 components provided in the duckiebox

2.1 Components

Our bot, the DB21J Model, provides as components:

- **NVIDIA Jetson Nano:** a small AI computer responsible for processing sensor data, and controlling the robot.
- **Battery with Diagnostics Sensor:** prsents the powerbank of the Duckiebot, that monitors the battery status.
- **HUT (Hardware Utility Board):** manages power distribution and communication between different parts, ensuring coordinated operation.
- **Display (128x32 pixels I2C):** a small screen used for showing system status, such as "Health", "Robot Info", and "Network Info".
- **DC Motor with Wheel Encoder:** a wheel motor that measures rotation to help with speed and position tracking.
- **Camera (8 Megapixels):** the primary vision sensor that captures images for lane detection, object recognition, allowing autonomous navigation.
- **Wi-Fi Adapter:** a USB device that enables wireless communication for remote control, and data transmission.

- **IMU (Inertial Measurement Unit):** a sensor that contains a gyroscope, accelerometer, and magnetometer to track the bot's orientation, movement, and stability.
- **Time of Flight Sensor:** a sensor that measures distances to objects using infrared light, helping with obstacle detection and thus, collision avoidance.
- **RGB LEDs:** provide visual feedback and signal different system states, as well as enhance user interaction.

2.2 Assembly

During assembly, components are interconnected as follows:

- The **NVIDIA Jetson Nano** connects to the **HUT**, which interfaces with the motors, IMU and ToF sensors, and fan.
- The **camera** attaches to the front bumper of the Jetson Nano.
- The **battery** connects to the HUT, providing power to the entire system
- The **Display** module connects directly to the HUT.
- The **Wi-Fi Adapter** is connected to one of the Jetson Nano's USB ports to enable wireless networking capabilities.

2.3 System Calibration

Proper calibration of the Duckiebot's components is essential for accurate autonomous operation. By fine-tuning key parameters, the Duckiebot can follow lanes more accurately, detect stop lines reliably, and maintain proper localization.

2.3.1 Camera Calibration

Camera Calibration ensures that the Duckiebot accurately interprets visual data for lane detection, obstacle recognition, and localization. The calibration process involves two parameters:

- **intrinsic calibration:** determines the camera's internal characteristics, such as focal length, optical center, and distortion coefficients and corrects distortions caused by the camera lens.
- **extrinsic calibration:** determines the camera's exact position and orientation relative to the Duckiebot's frame of reference. This ensures that objects detected in the camera's image, such as lane markings or obstacles, correspond accurately to their real-world positions.

After calibration, the calculated parameters are saved in:

```
/home/duckiebot/calibrations/
   camera_intrinsic.yaml
/home/duckiebot/calibrations/
   camera_extrinsic.yaml
```

2.3.2 Wheels Calibration

Wheels Calibration: the process of adjusting the Duckiebot's wheel parameters to ensure accurate and balanced movement. We used an expression that relates the **trim**, that adjusts the speed difference between the left and right wheels to maintain a straight trajectory, **gain**, that is a scaling factor that adjusts the overall speed of both wheels, and **tic**, encoder readings that measure wheel rotations, allowing accurate velocity estimation.
The wheel velocities are computed as follows:

$$V_{\text{left}} = \text{gain} \times (1 - \text{trim}) \times V_{\text{command}} \quad (1)$$

$$V_{\text{right}} = \text{gain} \times (1 + \text{trim}) \times V_{\text{command}} \quad (2)$$

```
duckie@duckie:~$ rosparam set /
   duckie/kinematics_node/trim
   0.05
```

```
duckie@duckie:~$ rosparam set /
   duckie/kinematics
```

3 Preconfigured System

Following the Duckietown guide, the SD card was set up with the preconfigured Duckietown software, which is built on **ROS** and powered by **Ubuntu**.
This system serves as the foundation for enabling autonomous navigation, integrating sensors, and managing robot control within the Duckiebot ecosystem.

3.1 Duckietown Framework

The Duckietown framework is an open-source software platform designed for developing and testing autonomous robotics in the Duckietown environment. It uses **Docker containers** to organize functionalities into separate environments. The provided containers are:

- **ros:** provides the core ROS environment, enabling communication between different nodes and containers.
- **code-api:** exposes an API for interacting with the Duckiebot's codebase, useful for remote access, development, and testing.
- **duckiebot-interface:** manages hardware interactions, ensuring communication between ROS and the Duckiebot's sensors and actuators.

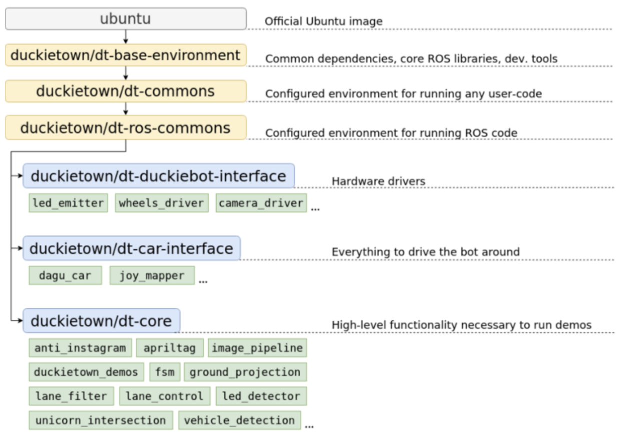
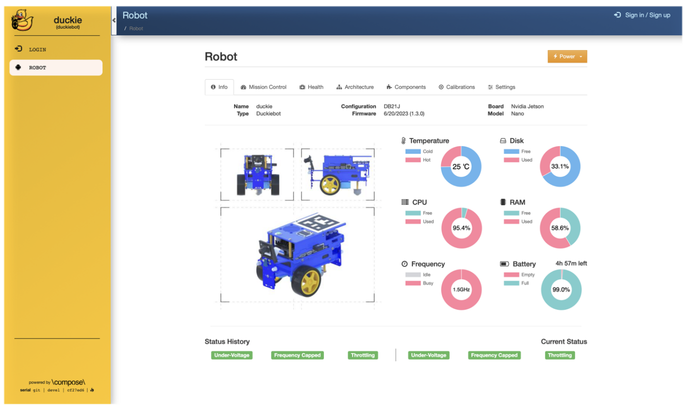

ubuntu
Official Ubuntu image
duckietown/dt-base-environment
Common dependencies, core ROS libraries, dev. tools
duckietown/dt-commons
Configured environment for running any user-code
duckietown/dt-ros-commons
Configured environment for running ROS code
duckietown/dt-duckiebot-interface
Hardware drivers
led_emitter
wheels_driver
camera_driver
...
duckietown/dt-car-interface
Everything to drive the bot around
dagu_car
joy_mapper
...
duckietown/dt-core
High-level functionality necessary to run demos
anti_instagram
apriltag
image_pipeline
duckietown_demos
fsm
ground_projection
lane_filter
lane_control
led_detector
unicorn_intersection
vehicle_detection
...

Figure 2 The Docker image hierarchy

- **device-online:** checks and maintains the online status of the Duckiebot's connected devices.
- **device-health:** monitors the health and status of Duckiebot's components (e.g., battery, temperature, connectivity).
- **device-proxy:** acts as a bridge between different devices, ensuring smooth data transfer and communication.
- **rosbridge-websocket:** enables web-based communication with ROS, allowing external applications (e.g., web dashboards) to interact with the Duckiebot.
- **files-api:** provides an interface for managing files, useful for logging, uploading, or modifying configurations.
- **dashboard:** runs a web-based dashboard for monitoring the Duckiebot.



Name duckie
Type Duckiebot
Configuration DB21J
Firmware 6/20/2023 (1.3.0)
Board Nvidia Jetson
Model Nano

Figure 3 duckietown dashboard

Within the framework, we did also use **dt-core**, the main software repository for the Duckiebot, that did help us integrate and expand within the Duckietown ecosystem. It contains the essential packages for navigation, and control, enabling the Duckiebot to operate autonomously. **dt-core** has mainly two important parts: **launchers** that help manage and start multiple ROS nodes, ensuring that all required components run smoothly and **packages** . Our team used the following packages (the package marked **UNCHANGED** are directly taken from Duckietown repositories, the others are modified):

- **anti_instagram [UNCHANGED]:** adjusts image colors and lighting conditions to improve computer vision performance by reducing the effects of extreme brightness, contrast, or saturation variations.
- **apriltag [UNCHANGED]:** is responsible for detecting and processing AprilTags, which serve as visual markers for localization and navigation. It identifies tags in images, extracts their position and orientation, and refines the data for accurate robot localization and environment mapping.
- **deadreckoning:** estimates the Duckiebot's position based on **wheel encoder data** and **motion history**, without relying on external localization systems. It continuously updates the

robot's estimated trajectory by integrating velocity and direction changes over time. While useful for short-term navigation, it may accumulate errors due to drift, requiring corrections from additional sensors like cameras or AprilTags.

- **duckietown_demos:** preconfigured **launch files** that start specific functionalities for autonomous navigation, perception, and system management.
- **fsm (Finite State Machine):** manages the robot's decision-making by transitioning between different states based on sensor inputs and predefined rules. It controls high-level behaviors such as stopping, turning, and lane following, ensuring smooth and reactive navigation.
- **ground_projection[UNCHANGED]:** transforms image coordinates from the camera into real-world ground coordinates, enabling accurate perception of lane markings and obstacles for navigation.
- **image_processing[UNCHANGED]:** processes camera images by first decoding raw image data and then rectifying distortions, ensuring accurate perception for tasks like lane detection and object recognition.
- **lane_control:** regulates the Duckiebot's movement to ensure it follows the lane accurately. It processes camera input, detects lane markings, and adjusts wheel speeds to maintain a stable trajectory. The **custom version** likely includes modifications for improved control, adapting to different environments or specific testing scenarios.
- **lane_filter[UNCHANGED]:** estimates the Duckiebot's position within the lane by processing visual data.
- **led_emitter:** controls the activation of LEDs to send signals or convey status information for communication and interaction.
- **stop_line_filter:** detects stop lines using camera data to assist in autonomous stopping at intersections, and provides a graphical representation of the detected stop line for debugging and analysis.
- **vehicle_detection:** identifies nearby vehicles using camera data.

3.2 ROS: Robot Operating System

ROS is a flexible framework designed for developing robotic applications. It provides tools, libraries, and communication mechanisms that enable robots to process data, make decisions, and interact with their environment efficiently. Running on **Ubuntu**, it simplifies robotic systems by managing hardware, and communication between components, that are:

- **Node:** individual process in ROS that performs specific tasks (e.g. controlling motors)
- **Topic:** a channel where nodes communicate between each other by send (publishing) and receive (subscribing to) data streams.
- **Service:** a **request-response mechanism** where a node can send a **request** to another node, which processes it and returns a **response**.
- **Package:** contains everything needed to run a ROS-based application:
  - **config**/ $\rightarrow$ stores configuration parameters
  - **launch**/ $\rightarrow$ includes launch files to start multiple nodes simultaneously.
  - **src**/ $\rightarrow$ contains the Nodes source code.
  - **include**/ $\rightarrow$ stores header files and necessary dependencies.

3.3 General Workflow

Duckietown provides a shell utility called *dts (duckietown shell)* to interact, build and launch programs on the Duckiebot. Despite providing essential tools for configuring the Duckiebot and debugging, the installation of *dts* remains cumbersome and OS-restricted to a specific version of Ubuntu. As we are working from different machines and different OS, this situation is less than ideal and pushed us to put in place a better workflow. SSH commands enabled remote access to the Duckiebot from any computer. From there we can directly go into to the docker containers and modified the code there. A git repository allows us to work locally and to push our changes from there. We also create some bash scripts to help us automatized repetitive tasks like `ros` to connect to the ros container and go into our main work folder and `rossync` to pull the code from our repository and copy it to the container.

4 Implemented Features

4.1 Operational Modes

In Duckietown, the Duckiebot operates in different modes to manage its behavior based on the environment and user commands. These operational modes define how the robot processes sensor data, controls movement, and responds to external inputs. The Finite State Machine (FSM) manages transitions between these modes, ensuring smooth operation and safety.

- **IDLE mode:** used when the robot is not driving, either before activation or after being stopped. It switches to **LANE_FOLLOWING** when idle_mode_off is triggered, or to **NORMAL_JOYSTICK_CONTROL** when joystick override is activated, but also it can switch to **RECOVERY_MODE** in case of an emergency stop.
- **Lane_following mode:** used for self-driving within a Duckietown environment, allowing the bot to navigate streets and intersections while staying within the lanes. It switches to **IDLE_MODE** when idle_mode_on is triggered, and to **RECOVERY_MODE** in case of an emergency stop.
- **Recovery Mode:** activated when an emergency stop is triggered, ensuring the Duckiebot halts safely.
- **Joystick Mode:** used to manually control the Duckiebot, overriding autonomous behaviors like lane following.

4.2 Lane Following

To follow the lanes, we first used lane filtering system that estimates the robot's position in the lane by processing detected lane segments, providing a stable estimate of lateral deviation and heading, and then, lane controller uses this information to compute steering and velocity commands, ensuring the robot stays centered and follows the lane smoothly. We did make changes when it comes to the parameters presented by the default lane_control, such as:

- **Base Speed (v_bar):** Base speed of the robot.
- **Lateral Deviation Correction (k_d):** a parameter that adjusts how aggressively it corrects deviations from the center of a lane. We did decrease the value to have stronger lateral corrections.
- **Heading Correction (k_theta):** a parameter that controls how strongly the robot adjusts its orientation to stay aligned with the lane direction. Our custom version reduces heading correction strength to adjust its heading more slowly, reducing overcorrection.
- **Integral Gains for Lateral and Heading Correction (k_Id, k_Iphi):** We did change the K_Id to have less accumulation of past errors, preventing overshoot in long turns or curved paths.
- **Heading Thresholds (theta_thres):** Narrows the allowed heading error range, so the robot will start correcting sooner, making lane-following more precise.

Our custom version lane_controller_custom_node allows finer control of heading errors.

4.3 Intersection Navigation

The current intersection navigation system in Duckiebot, called *indefinite_navigation* relies on a **Finite State Machine (FSM)** to manage state transitions, activating and deactivating nodes based on predefined conditions. While this approach allows for a fast and easy implementation of new states, it also introduces several issues that affect the normal execution of the program and create unexpected behavior.

The main problem comes with how the FSMNode is handling the activation/deactivation of the nodes. As the FSMNode cannot directly switch on and off other nodes, it relies on publishing on a unique topic for each node if it should be active or not. It also try to call some services to change the behavior of specific nodes. If some services are not reachable, the FSM node will wait for maximum 10 seconds, halting the program execution. As a result, some nodes that should be activated aren't and some others that should

stopped continue to run. For example, when transitioning from **INTERSECTION_CONTROL** to **LANE_FOLLOWING**, `lane_controller_node` and `stop_line_filter_node` needs to be activated while `unicorn_intersection_node` should be deactivated. If the FSM node tries to switch off `unicorn_intersection_node` first it will halt for 10 seconds as the related service is unavailable (most likely due to bad renaming on a launch file)[1]. This cause the Duckiebot to continue moving without properly restoring essential nodes, here the possibility to follow the line and stop at red lines.
With some research, it looks like that we were not the only one to have issues with the *indefinite_navigation* of Duckietown. This is why we decided to completely change our approach to solve the problem of the intersection navigation.
As a result, our work was based on the implementation of Samuel Neumann, PhD student at the University of Alberta. His approach roughly consist of merging `unicorn_intersection_node` with `lane_controller_node` removing the need for changing state with the FSM node.
The `unicorn_intersection_node` was in charged to change the parameter of the lane following to make the Duckiebot deviate from its normal position depending on the turn the bot should do. This forces the `lane_controller_node` to wait for the new parameter to be updated without the possibility to check if the current parameters are the ones expected or not. Samuel Neumann solves this by directly publishing specific *v* and *omega* value to the car controller for a fixed period of time depending of the type of turns (going straight included). This also creates by the same occasion a more sequential execution of the intersection navigation, as `lane_controller_node` becomes the only file to publish commands to the wheels.

4.3.1 Overall Execution

By only using one file for the intersection navigation, we can simplify and optimize the execution of our program. The `lane_controller_node` keeps track of the situation the Duckiebot is in with two flag variables `at_stop_line` and `is_turning`. The main loop `drive()` is executed every tenth of a second and depending of the state of this two variables, it calls the respective functions `lane_following()` (default case), `at_intersection()` and `turns()` always in this order.

[1]This part of the code is not documented at all, so this is only an assumption

4.3.2 Stopping at red lines

The first step in the intersection navigation is of course being able to stop at the red lines. The detection of the red lines is done in the *stop line filter* package and the original one works relatively well except for one major design flaw, the Duckiebot always stops around 10 centimeters before the red line. The reason for this is simple, the `stop_line_filter_node` detects all the red segments and compute their average x and y distance to the center of the Duckiebot, when a specified threshold has been reach the node publish on a topic that the bot has reach a stop line. If the threshold distance is too small, the red line will be out of the camera field before the desired distance is reach. And with no red line in sight, there is no need for the bot to stop! In a real life situation, it is obviously not ideal for a car to stop so much before a stop line (bad visibility, not detected by the red light trigger, etc.). Even in Duckietown, this situation makes it harder to implement reliable turns as this requires to go first a bit straight then turn for example. To counter act this, we first implemented a timer to delay the stop. This approach was working fine but the too big speed variation makes it too unreliable, causing the bot to stop either too soon or too late. The better solution we came up with was a distance-based solution. For computing the odometry of the Duckiebot, the `deadreckoning_node` computes first the average linear distance traveled by the wheels, we can use this information to know the total (absolute) distance travel by our Duckiebot over time. Now instead of delaying the stop by a fixed amount of time, we can delay it by the distance the bot needs to cover to get to the red line from the moment we reach the `should_stop` threshold.

4.3.3 Turning

For turning, we decided to use the same approach as for the red lines stops and to modify the time-based approach into a distance-based one. The implementation remains similar, we keep track of the distance traveled by the Duckiebot during the turn and stop it when a predefined distance threshold has been reach. In general, this allows for more reliable turns and fixed most of the issues we were facing so far. For example in the time-based approach, a very slow speed would cause the bot to stop its turn in the middle of the intersection, using the distance ensure the full execution of the intersection.

For the both the red lines stops and the turns, the distance-based approach uses real feedback to compute and deliver the expected outcome. Thus the Duckiebot relies less on arbitrary value that are often affected by its physical limitation. This improves the overall reliability of the Duckiebot and therefore also its safety.

4.4 Navigation using Dijkstra's Algorithm

We implemented Dijkstra's Algorithm in order to ensure efficiency and safety, by determining the optimal path from a starting point to a destination.
We started first by assigning the actual map of the city to a graph, where its **nodes** correspond to individual tiles.
As a first step, we computed the shortest path from a starting point to a destination using Dijkstra's algorithm. It initializes all distances to infinity except for the starting node, set to 0, and iteratively updates the shortest paths by expanding the node with the smallest known distance. A predecessor dictionary allows path reconstruction once the goal is reached. By systematically evaluating nodes, the **Dijkstra class** efficiently finds the shortest route, making it ideal for structured environments like Duckietown.
Next, we implemented Dijkstra within the Duckietown environment. We defined a **get_direction()** function to determine whether the robot should go straight, left, or right at intersections based on node coordinates. The graph is instantiated, populated with map data, and a shortest path is computed. The testing framework verifies the pathfinding logic before integrating it into navigation nodes like **DijkstraTurnsNode**.
Finally, we integrated Dijkstra with the **Finite State Machine (FSM)** to track the robot's operational mode and execute turn decisions only when necessary. The node subscribes to route and position updates, allowing it to determine the correct movement based on the computed shortest path. It interacts with the graph-based map representation **DUCKIETOWN_CITY**, the route planner **Route**, and the **Dijkstra algorithm** to ensure the robot follows an optimal trajectory. Additionally, it publishes turn commands and manages the robot's idle mode, ensuring smooth transitions between navigation states. This step was the bridge between high-level path planning real-time execution, allowing the robot to make precise and informed decisions while navigating intersections.

4.5 Custom Web Dashboard

React.js is a JavaScript library for building interactive user interfaces. It has a component-based approach that offers a fast and efficient way to update the content of a web page.
In combination with the rosbridge package (also used by Duckietown) and the ROS library for JavaScript (roslibjs), we can communicate with the duckiebot via a WebSocket and create a fully interactive web interface that can subscribe/publish to topics but also call services. This interface was created in a way to be easily implemented yet powerful and greatly extendable.
Each of the different modules is a React component that can all access the same ROS instance. This allows every component to subscribe to any topics it might use and to unsubscribe from the topic when it's not rendered. If any information sent on a topic needs to be accessed by multiple components, for example for the position, the FSM state or the emergency break status. The subscription/publishing can be handled by the main component (App.js) or any parent of the components requesting this information. The value can be then passed down by the parents with the use of props.
In general, there is no need for a child component to update its parent, as this can result in unsynchronized or outdated data. If this is the case, it is important to ensure the value cannot be modified by another component or another system on the duckiebot. Here is a list of the different components from the interface:

- Joystick controller
- Map
- Path finder (to start the Dijkstra navigation)
- Navigation Information (when navigating with Dijkstra)
- Camera Feedback

Some of the components, such as the camera or the joystick, are direct copies of tools provided by

Duckietown, but adapted for web usage. The main benefit is being able to use these tools (very useful for debugging) without any OS restrictions.
Creating this web interface might look a bit out of scope with the main idea of the project but is a great asset both on the debugging side and on the user side. Indeed, a friendly UI not only improves comfort, but also makes safety measures and procedures more understandable and accessible to the users.

5 Autonomous Driving Safety

Ensuring safety has been a central focus throughout our work, as it is essential in the Duckietown ecosystem. As an autonomous system, the Duckiebot must operate with high reliability to prevent collisions and ensure smooth navigation. This is achieved through key safety mechanisms, such as **emergency stop**s and **intersection safety protocols**, which help mitigate risks and enhance overall system stability. As when it comes to the **emergency stop**, we tried to determine when an emergency stop is necessary, by monitoring sensor inputs including StopLineReading messages. Upon detecting a stop condition, the node immediately sets the robot's linear velocity to zero, halting its movement. As for **the lane_following**, to not have a safe navigation, so we kept changing the parameters, so the bot can follow the lanes steadily.
We wanted also to ensure a **safe intersection**, by monitoring stop lines, intersection lane poses, and obstacle distances to prevent collisions. Using stop_line_reading that provides the distance from the stop line, If the Duckiebot is approaching a stop line, it gradually reduces speed.
Since **traffic lights** provide an effective key to ensure the safety goal, we tried to work on implementing them, but, due to some technical issues related to our TUM Duckietown city, we found out that some traffic lights were mounted but not on, so the bot cannot learn about the intersection presented through them.

6 conclusion

Throughout this project, we implemented and enhanced various autonomous driving functionalities within the Duckietown ecosystem. By focusing on perception, navigation, and control, we developed a more robust and reliable system capable of autonomous lane following, intersection navigation, and emergency stops. We tried to work, as well, on safety mechanisms, such as real-time stop line detection and intersection awareness to minimizing risks.
However, our team did face several challenges throughout the work. At a first level, we faced hardware issues such as battery limitations, as well as, physical limitations from the wheels due to a to high grip on the tiles surface.
At a second level, many essential parts of the system are either poorly documented or have had their documentation removed entirely. This significantly did slow down debugging and development.
As a last point, we observed a lack of consistency in coding practices, such as no naming conventions, either no comments or too many unnecessary comments, making it difficult to integrate or modify existing functionalities.
As future enhancements, we want to focus on apriltags detection, smarter obstacle avoidance, traffic light awareness, so the focus will be on better perception, smarter decision-making, real-time adaptability, and a stronger software foundation.

References

[1] Duckietown Assembly
https://docs.duckietown.com/daffy/opmanual-duckiebot/assembly/db21j/index.html
[2] Code Hierarchy
https://docs.duckietown.com/daffy/devmanual-software/beginner/code-hierarchy/index.html
[3] Docker
https://docs.duckietown.com/daffy/devmanual-software/basics/development/docker.html
[4] ROS Tutorials
https://wiki.ros.org/ROS/Tutorials
[5] dt-core code
https://github.com/duckietown/dt-core
[6] Wheels calibration
https://docs.duckietown.com/ente/opmanual-duckiebot/operations/calibration_wheels/index.html
[7] Samuel Neumann - Intersection Navigation
https://samuelfneumann.github.io/posts/duckie_4/