# Project Parking
# Autonomous Mobility on Demand
# ETH Zürich

Trevor Phillips, Vincenzo Polizzi, Linus Lingg

December 2019

## Abstract

There is currently no existing solution for autonomous parking in Duckietown. The goal of our project was to implement such a solution, in order to provide functionality for Duckiebots to get out of the way and potentially in the future, to charge themselves automatically. This includes entering and exiting a parking area, parking, exiting a parking spot, and avoiding collision with other Duckiebots during the entire process. We hope our work provides a basis for future students to implement various parking maneuvers and precise coordination between Duckiebots during parking. See [2] for a video demonstration of our work.

# Contents

## 1 Mission and Scope

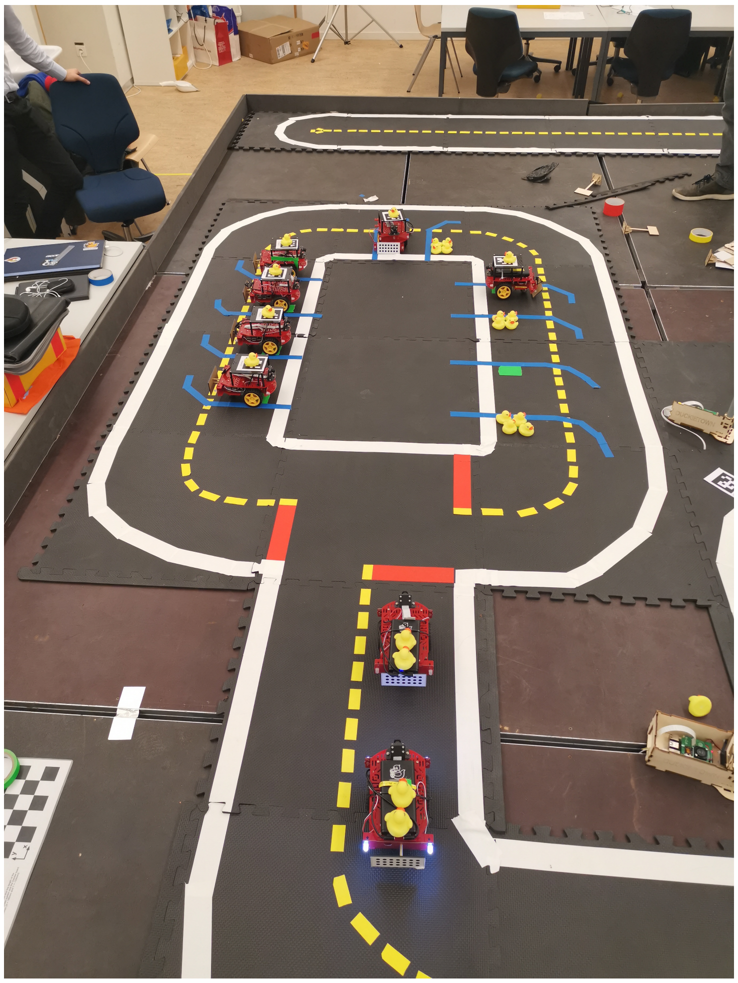

Figure 1: An example parking area configuration

### 1.1 Motivation

For this project, our primary goal was to implement autonomous parking for Duckiebots within Duckietown. One of the main reasons why this is being addressed, is to provide a function such that Duckiebots can get out of the way and stop in parking spaces for a few minutes when desired, or to dock for charging.

## 1.2 Existing Solution

There have been some prior projects which attempted to tackle this challenge, however we decided to start from scratch and did not base our project on any prior work, in order to gain a full understanding of the system and to have ownership over our code. Nevertheless, before starting our implementation, we researched some papers on autonomous parking and used this as a basis and for inspiration.

## 1.3 Opportunity

The realization of this project would add some useful features to Duckietown, as it enables the Duckiebots to exit Duckietown for a brief rest and potentially recharge, if our project is later extended to integrate with wireless charging.

## 1.4 Preliminaries

*Crash avoidance*: To reduce the crash probability, we use a time slotted algorithm. The idea comes from telecommunication protocols such as Aloha [3]. Indeed, we can draw a comparison between the parking area and a network. In the latter we have sources and a common medium on which these sources want to send messages, and respectively in the former, we have parking spots and a lane.

*General*: The reader should have general knowledge about ROS and the overall Duckietown infrastructure (both software and hardware). Moreover, if the reader is interested in improving our code we strongly encourage him or her to have general knowledge about Docker and Python.

# 2 Definition of the Problem

## 2.1 Goal

The goal of this project is, as previously mentioned, to create a fully autonomously working parking area. This includes Duckiebots entering the parking area, parking, exiting the parking area, and avoiding collisions during the entire process.

## 2.2 Assumptions

To realize the above goal we assume that the existing Duckietown infrastructure, specifically Lane Following and Indefinite Navigation, is functioning. Furthermore, we assume that there is Duckie placed on top of each Duckiebot, parking April Tags are correctly placed, and the following Docker images are up-to-date:

- `dt-core`
- `dt-ros-commons`
- `dt-duckiebot-interface`

- `dt-car-interface`

## 2.3 Performance Metrics

To determine the quality of the parking system, we utilized four distinct metrics to evaluate performance:

- # Duckiebots within the parking area at the same time
- % of complete, successful parking maneuvers
- Time needed for an entire maneuver
- Precision of the parking maneuver (angle and distance from desired pose)

# 3 Contribution and Added Functionality

## 3.1 Area Design

Figure 2 depicts one possible parking area design. To build this, we use six `DT17_tile_straight` tiles, four `DT17_tile_curve_left`, one `DT17_tile_three_way_center` tile and two `DT17_tile_empty` tiles. These are all standardized tiles from Duckietown. All of the straight tiles can be easily transformed into parking tiles by following the instructions found in the project's README file [1]. The resulting parking tile should look as shown in Figure 3. This implementation makes the parking area completely modular, meaning the user can adapt the design depending on the space available. The only restriction is, that the T-intersection must be placed in a way such that the Duckiebot must do a right turn to enter the area, turn right to exit the area, and go straight to stay within the area as these parts are hard-coded into our implementation.

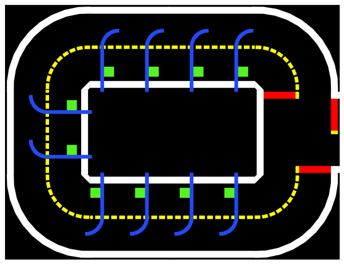

Figure 2: Example parking area

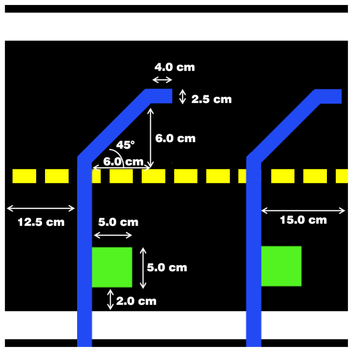

Figure 3: Parking Tile

## 3.2 System Interfaces: Software

The packages we modified and added for this project are as follows:

- Lane Following $\rightarrow$ used for navigating around the parking area and parking itself
- Vehicle Avoidance $\rightarrow$ avoid other Duckiebots while searching for parking and exiting the parking area
- Line Detector / Lane Filter $\rightarrow$ detect blue lines
- Lane Controller $\rightarrow$ backwards movement and manual, hard-coded commands in certain situations
- April Tag Detection $\rightarrow$ for detecting special parking April Tags and triggering a switch in the Finite State Machine (FSM)
- FSM / Nav $\rightarrow$ for switching between parking and normal operation

The new files are:

- Parking States $\rightarrow$ defines logical flow during parking
- Parking Spot Detection $\rightarrow$ determine whether a parking spot is free
- Lite Red Line Detection $\rightarrow$ indicates if there is a red line in front of the Duckiebot
- White Line Detection $\rightarrow$ indicates if there is a white line in front of the Duckiebot
- LED Detection $\rightarrow$ determines if there is a Duckiebot with LEDs turned on, within a region of interest

## 3.3 Contributions

**Backward Lane Following**

Our main problem since the beginning of the project was to drive backward in closed-loop control. To achieve this goal, we started looking at the model of the Duckiebot to design an ad-hoc controller. Actually the problem that arises is that when going backward, the center of mass is in front of the actuators and so the system is much harder to control. However the model does not change much.
After that, we thought about the pose estimation and the calculation of the error to be passed to the controller. The pose estimation remains the same as normal lane following, but the error calculation is instead slightly different. Since we want to go backwards, the pose, namely the relative distance and angle between the Duckiebot and the middle dashed line (where the Duckiebot has to be) is the opposite of the forward driving direction. What we did was to change the sign of the pose and re-tune a PID controller to achieve stable backwards lane following.

### Dynamic Color Adaption for Lane Following

We adapted the `lane_detector` such that the Duckiebot looks for blue lines as well as yellow, red and white ones. This means that the `lane_filter` can now handle blue lines as well. When you publish 'blue' to `/parking/lane_color`, the lane filter will ignore the white and yellow lines and only filter blue lines, which it expects to be at the same relative position that the yellow lines used to be.

### Time Slot Coordination

Our working principle (shown in Figure 4) is to divide the time in slots (of 20 seconds for the parked Duckiebots and 5 seconds for the Duckiebots driving around the parking area). The global time is the same for all Duckiebots, so each of them knows exactly in which time slots it is. When a Duckiebot wants to exit from the parking spot, it chooses a time slot randomly and switches on its red LEDs. When the Duckiebots that are driving around detect red LEDs, they check in which time slot they are, and if they are in in the first or the last 5 seconds of the time slot they stop. Otherwise if they are not in this interval, they continue driving. With this we minimize the risk to have a Duckiebot backing out into another Duckiebot that waits for a third one which is exiting. This probability could be reduced even more if more time slots for the parked Duckiebots are added. However, this will also slow down the whole parking procedure as in the worst case the Duckiebot waits until the last slot. The trade off between collision avoidance and waiting time is in a nice balance using four time slots.

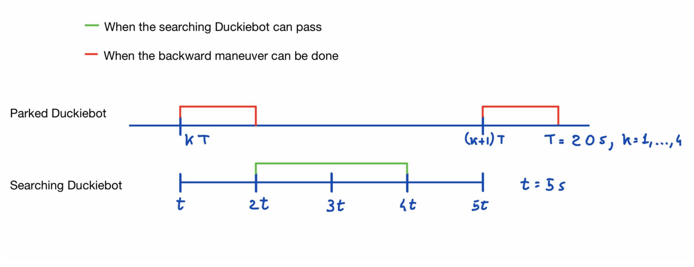

Figure 4: Time slot working idea

## 3.4 Logic

### Parking States

- INACTIVE
- ENTERING_PARKING_LOT
- SEARCHING
- IS_PARKING

- IS_PARKED
- EXITING_PARKING_SPOT
- EXITING_PARKING_LOT

**Indefinite Navigation → Parking**

When running indefinite navigation we can tell the Duckiebot to start responding to parking April Tags when at an intersection by setting the boolean `/BOTNAME/parking_on` to `True`. Each time the Duckiebot is then at an intersection within Duckietown, it will try to find a parking April Tag. If it sees such an April Tag, the boolean `/BOTNAME/intersection_navigation_node/parking_intersection` is set to `True`. After the intersection turn is finished (as always within indefinite navigation) the boolean `/BOTNAME/intersection_navigation_node/intersection_done` is set to `True`. All of these three booleans being `True` triggers then the FSM (Finite State Machine) to switch from Indefinite Navigation to Parking, because now the Duckiebot is approaching the entrance of the parking area.

**ENTERING_PARKING_LOT → SEARCHING**

After entering the entrance, our parking node becomes active, which in the beginning does normal lane following. This means the Duckiebot then just normally follows the lane until it sees a red line in front of it. This is done by the Lite Red Line Detection Node which looks at the lower part of the image and runs red color detection. When it arrives at the red line, it will turn on its LEDs to red and will fall asleep for 3 seconds. After these 3 seconds, it will look for red LEDs within the left horizontally centered part of the image. If it sees some red LEDs, it will fall asleep for 10 seconds and redo the same procedure again. If not, it will do the right turn and change to the SEARCHING state.

**SEARCHING → IS_PARKING**

Now the Duckiebot is actually within the parking area and starts looking for free parking spots. For this it will do normal lane following and at the same time look for square green areas within the left bottom part of its image. Parallel to this, it runs the normal back-bumper detection as well as red led detection. The back-bumper detection is ignored if the back-bumper detected is at an angle of around 90 degrees. In this case, we ignore the message and keep on driving.

If there is a LED detected within the vertically centered left hand side of the image, the Duckiebot checks where within the current time slot it is. The Duckiebots that are driving around the parking area have slots of length 5 seconds each. If they are within the first or last 5 seconds of one of the 20 seconds slots of the parked Duckiebots, they will wait for 5 seconds, otherwise they ignore the detection and drive past the Duckiebot that wants to exit, because they have enough time to do so. All this is done to minimize the probability of crashing.

If the Duckiebot comes up to a red line while in the SEARCHING state, it will turn on its LEDs on red and fall asleep for 3 seconds. Afterwards it looks for red LEDs within the vertically centered right hand side of the image (same procedure as when entering the area, but looking at the right part of the image this time). As soon as there is no Duckiebot in sight it will go straight such that it can continue circling around and looking for free parking spots.

If there is a green square detected in the lower part of the picture, it signals by turning on LEDs and stopping. We then adapt lane following such that white lines are ignored, and blue lines are observed instead of yellow lines. Also we adapt the lane controller's `d_offset` such that the Duckiebot covers the green square when parked.

#### IS_PARKING → IS_PARKED

Lane following is run while looking only at blue lines. The Duckiebot follows the blue line until it sees a white line in front of it (same principle as the red line detection) which will trigger it to stop lane following, turn of its LEDs, and fall asleep.

#### IS_PARKED → EXITING_PARKING_SPOT

This is triggered by setting `/parking/time_exiting_parking_spot` to `True`.

#### EXITING_PARKING_SPOT → EXITING_PARKING_LOT

As soon as the boolean mentioned in section 3.4 is set to `True`, the Duckiebot switches its LEDs on and randomly chooses one out of 4 slots of 20 seconds length each.

As soon as his slot has arrived, he starts backward blue lane following until he sees within the center/lower part of the image the green square again. When the green square is seen, lane following is stopped and the Duckiebot does a hard-coded open loop backward left turn to be aligned with the lane again. After this turn, normal lane following is restarted.

#### Parking → Indefinite Navigation

We return to Infinite Navigation directly after the Duckiebot sees a red line when in the EXITING_PARKING_LOT state. After the red line is seen, the Duckiebot turns right and normal Infinite Navigation starts.

## 3.5 Further and Failed Ideas

- Improve lane following using a Smith Predictor (delay compensation). We were not able to implement it, but it could have a strong impact on lane following, especially for the curves.

- Implement backwards parking (potentially with closed-loop control by using April Tags for localization).
- Make parking spots unique by placing April Tags on the ground instead of identical green squares.
- We discovered that detecting April Tag on the lane and running lane following is hard

## 4 Performance Evaluation and Results

### 4.1 Results

As shown in our video [2], the final outcome is quite satisfying.

### 4.2 Evaluation

- # Duckiebots within area at the same time: **4** (should work with many more, but not tested due to limited number of available Duckiebots)
- % of complete, successful parking maneuvers: **75%**
- Time needed for one entire maneuver: **55 seconds**
- Time neededfor entering the parking spot: **4.8 seconds**
- Time needed for exiting the parking spot: **7.5 seconds**
- Precision of parking maneuver: **+/- 5 cm, +/- 10 degrees**
- % of Parking Spot recognition: **90%**
- % of Red Line detection: **95%**
- % of White Line detection: **95%**
- % of successfully entering spot after detection: **95%**
- % of successfully exiting spot: **90%**
- % of exiting Duckiebots detection: **85%**
- % of Duckiebot recognition at intersection: **80%**

### 4.3 Conclusion

Our biggest issue was with Lane Following. For example, the intersection behaviour of our code is strongly dependant on lane following functioning, as the Duckiebot needs to arrive more or less perpendicular at the red line of the intersection such that open-loop right turns and going straight work correctly. This problem can be avoided when the indefinite navigation is run instead of normal lane following, as the intersection behaviour then should be more stable.

Also our LED detection could certainly be improved to remove the few false positives we get at the moment. There are barely any false negatives however we do have some false positives which is certainly preferred to having false negatives. However, this can be avoided by looking for patterns and letting the LEDs blink whenever we want to exit or whenever we are at an intersection. This would be more reliable and would certainly fix the issue.

## 5 Future Avenues of Development

There are still some tasks that can be done to stabilize and improve the overall performance. Below we list some of the possible (and certainly useful) improvements:

- Integrate Indefinite Navigation into the parking intersection
- Improve efficiency of time-slotted coordination
- Improve red LED detection (OpenCV blob detection parameters)
- Improve intersection navigation (more stable turning)

# Bibliography

[1] GitHub Project Parking. https://github.com/duckietown-ethz/proj-parking/tree/v1.

[2] Proj-Parking Demo Video. https://vimeo.com/380507214.

[3] Slotted Aloha. https://en.wikipedia.org/wiki/ALOHAnet#Slotted_ALOHA.