Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Dynamic Systems and Control
Institut für Dynamische Systeme und Regelungstechnik

Sebastian Nicolas Giles
Christian Leopoldseder
Matthias Wieland

Project Lane Following Intersection (proj-lfi)

Final Report

Autonomous Mobility on Demand (Duckietown)
Institute for Dynamic Systems and Control
Swiss Federal Institute of Technology (ETH) Zurich

Supervision:
Amaury Camus
Merlin Hosner
Dr. Jacopo Tani

December 2019

Table of Content

List of Tables

List of Figures

Acronyms and Abbreviations

- LFI Lane following intersection
- EKF Extended Kalman Filter
- DB Duckiebot
- DBSCAN Density-based spatial clustering of applications with noise

Mission and Scope

Our mission is to enable any Duckiebot to reliably cross an intersection on a smooth trajectory and give control back to lane following when appropriate to allow for continuous navigation.
A successful intersection crossing means that the Duckiebot, starting from a position in front of a stopline, reaches the target lane without invading other lanes or crashing into other Duckiebots.
“Reliably” means that it should be able to navigate despite other Duckiebots waiting at other stoplines with a reasonable rate of success.

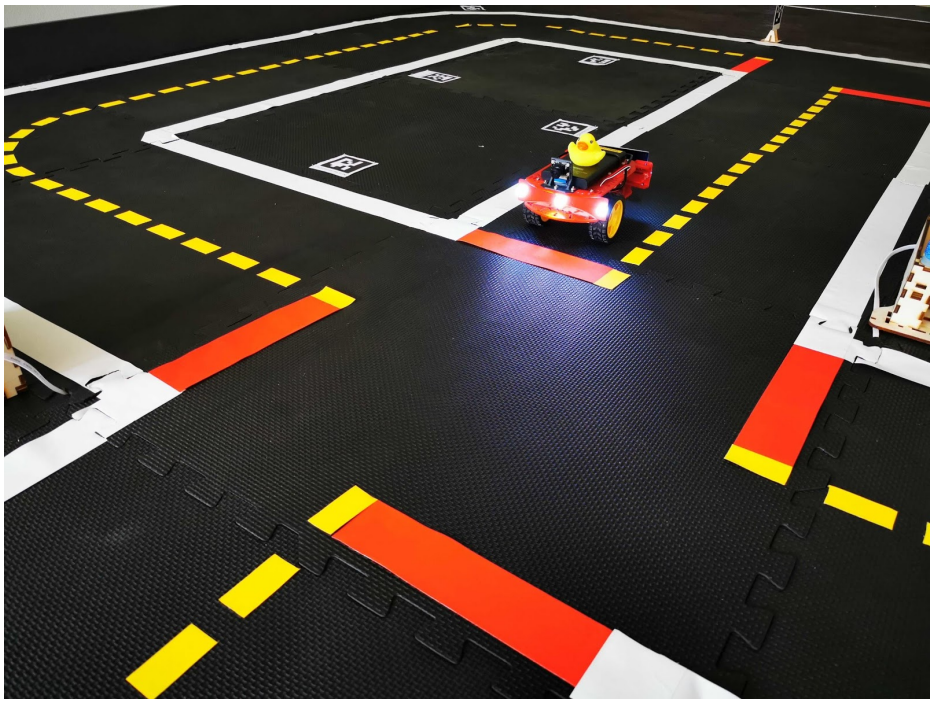


Figure 1: Intersection navigation options

Motivation

Intersections are an essential part of duckietowns, since without them, Duckiebots can only drive autonomously on straight roads or loops. So far, however, no good solution for Duckiebots to navigate intersections quickly and reliably has been implemented. Since this is a core functionality of every autonomous vehicle that interacts with street-like infrastructure, the approach proposed and implemented in this project tries to fill this hole.

Existing Solutions

Currently Used in Indefinite Navigation

The currently used method for intersection navigation is a simple open loop approach, where timings and angles for turns are pre-configured. When the Duckiebot come to an intersection it will execute these pre-configured instructions without gathering feedback from its environment. Whether the Duckiebot will cross the intersection successfully largely depends on luck since no correction can be made in case of unpredictable robot behavior (e.g. slippage). This implementation also does not take the initial position of the Duckiebot into account, which in itself shows that this solution is very lacking.

2018 Project LFI Implementation

Last year's intersection navigation team based their approach on the april tag detection library [1], which provides position and orientation. These measurements were then filtered using an Extended Kalman Filter and sent to a pure pursuit controller, which computed the appropriate motor commands.
The problem with this approach, however, is that the april tag detection library is rather slow, because of its high computational complexity. It can only provide an unsatisfactory update rate of approximately 5 Hz. In addition, The april tag library does not give accurate orientation information when the Duckiebot is facing an april tag head-on. Furthermore, the april tag's position on intersections in Duckietown are not exactly specified, and from experience we can say they are often moved when people or Duckiebots bump into them. That means that even if the library gave very accurate measurements, they most likely cannot be fully relied upon. All these problems combined lead to a high failure rate and make this idea, in our opinion, infeasible.

2017 Project LFI Implementation

The Navigators team implemented a template matching based approach, based on the paper Edge-Based Markerless 3D Tracking of Rigid Objects [2].
Due to the computational heaviness of the approach, a fast navigation could not be achieved, and due to slow control update cycles, the success rate of the standard implementation could not be improved.

Opportunity

So far the Duckietown platform is still lacking a robust intersection coordination package. All the previously developed solutions were either to slow or didn't use measurements at all. Therefore we decided to tackle this problem with a completely new approach, which detects all the existing red stoplines of the intersection, clusters them and calculates the pose of the Duckiebot based

on the relative position to the stoplines. The Duckiebot then navigates through the intersection with an offline precomputed optimal trajectory.

Preliminaries

The heart of the state estimation is a DBSCAN clustering algorithm [3]. The DBSCAN algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped. DBSCAN also doesn't assume a fixed number of clusters.

Definition of the Problem

Scope

This project is about navigating a Duckiebot through an intersection to turn left, right, or go straight. When a Duckiebot comes to an intersection it goes through the following steps:

1. Stop: The Duckiebot stops at the red line of its lane.
2. Alignment: The Duckiebot might need to align itself in order to have both traffic signs in its field of view.
3. Read signs: The Duckiebot processes the traffic signs to know how to proceed.
4. Coordinate
5. Receive go signal: The Duckiebot is signaled to cross the intersection and the direction of where to go is given.
6. **Navigate**: formulate commands for the motors to guide the Duckiebot through the intersection to the next lane.
7. **Lane-following handover**: Control is handed over to ordinary lane following.

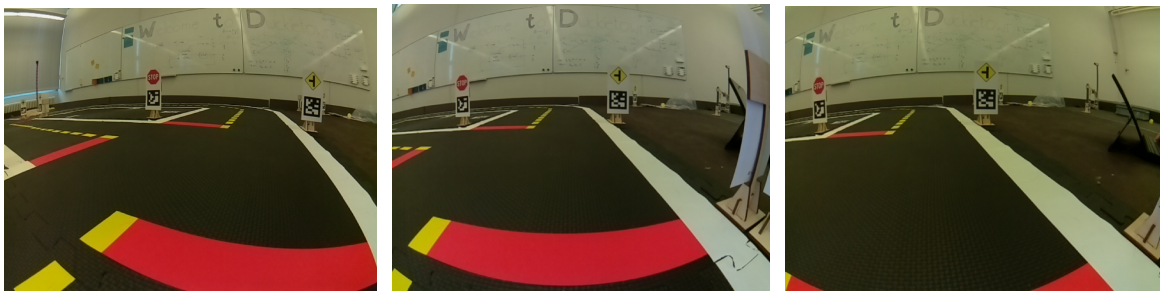
The steps in **bold** are the ones in the scope of this project.

Initial position

The Duckiebot stops at the stopline. Ideally, it would end up perfectly in the center of the lane (Figure 2 (b)), perpendicular to the stopline. Of course, this is not what happens in reality (Figure 2 (a), (c)). The Duckiebot need to be able to handle this initial condition uncertainty.

(a) Poor alignment (b) optimal alignment (c) Poor alignment

Figure 2: Duckiebot alignment relative to the initial stopline for a three-way intersection



(a) Poor alignment (b) optimal alignment (c) Poor alignment

Figure 3: Field of view of the Duckiebot camera corresponding to the alignment in Figure 2

Navigation

After alignment, there are three possible scenarios for the Duckiebot (Figure 4):

- Turning left
- Turning right
- Going straight

The Duckiebot needs to be able to handle all of these scenarios correctly by driving over the intersection and reaching the given exit.

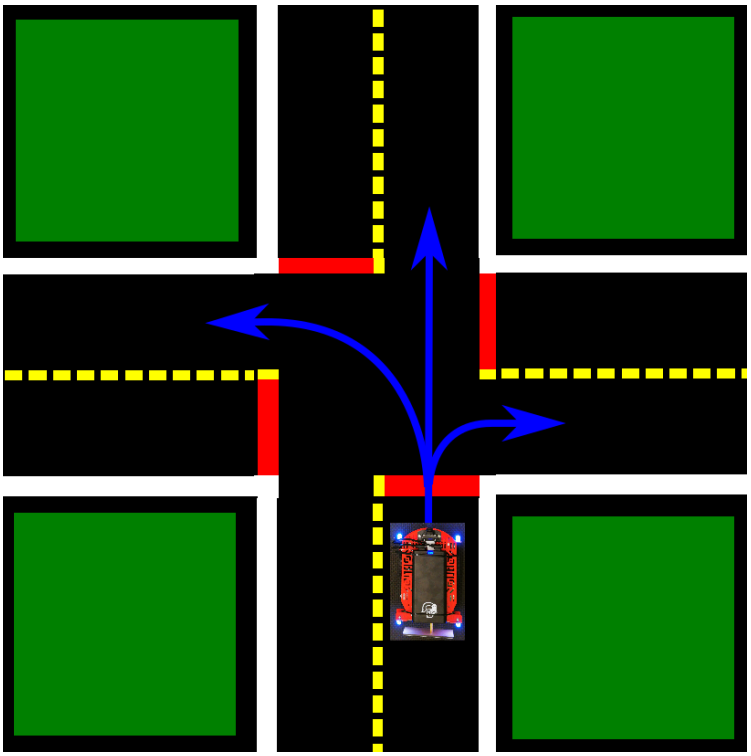


Figure 4: Navigation possibilities

Handover Back to Lane-Following

When exiting the intersection to the correct direction, control needs to be handed over to the lane-following subsystem. This needs to happen in a state (i.e. position) from which the lane-following subsystem is capable of reliably continuing navigation of the Duckiebot without further assistance of intersection navigation.

Assumptions

Intersection navigation will build upon the following assumptions:

- The intersections are built as specified in the appearance specification.
- In the initial position of the Duckiebot there is at least one stopline in view.
- There is an existing lane controller that takes a lane pose and moves the Duckiebot forward while correcting its pose.
- There is only one Duckiebot in the intersection at once.
- There are no obstacles on the intersection.
- Good light conditions (e.g. no illumination problems, ...)

Additional Requirements

There are a few additional requirements for the solution:

- Be able to handle both three-way and four-way intersections.
- Drive a full arc (no stop-turn).
- Don’t drive too slow (ideally constant speed).
- Don’t invade other lanes, especially when exiting an intersection.

Performance Metrics

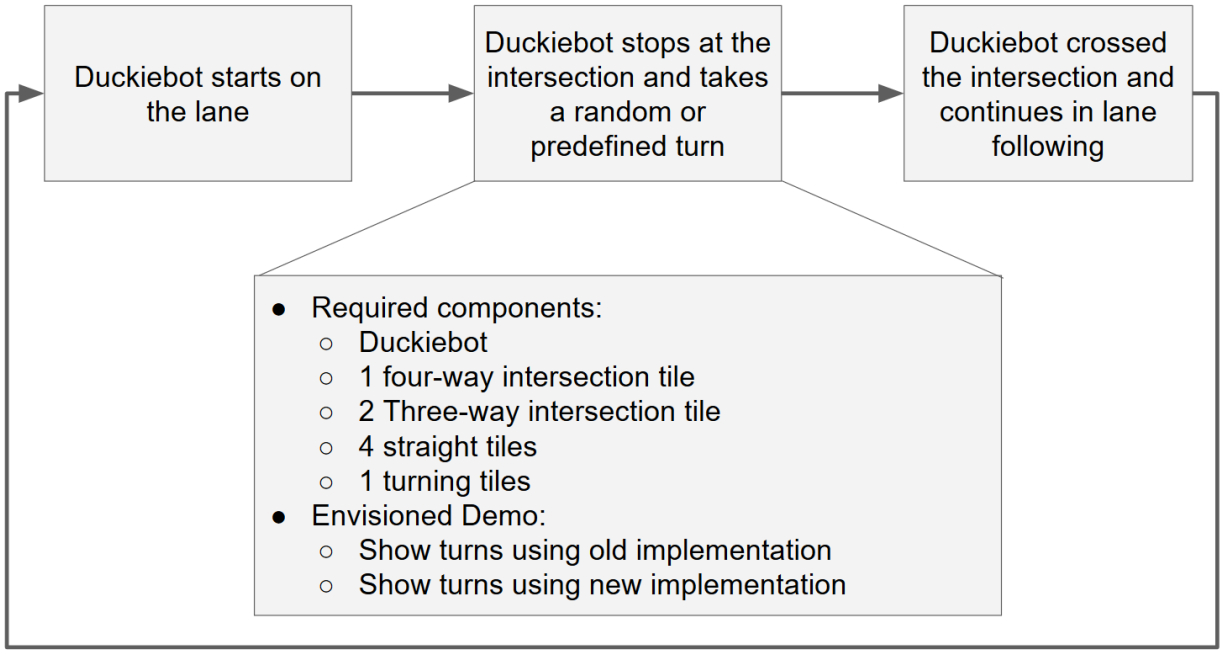


Figure 5: Procedure performance testing

- **Success rate**: The percentage of trials for which the Duckiebot ends up in the desired lane and successfully hands over the control to the lane following controller. A trial is considered to be successful if the Duckiebot is completely inside the desired lane without touching any lane markings. The success rate is evaluated by performing the setup loop (Figure 6) until the robot fails. The loop contains every possible turn type and thus represents a kind of “real-world” test as a Duckietown also usually exhibits all possible turn types. The loop could be adapted to test specific turns in isolation. A similar test for crossing intersections straight can be conducted on a loop where intersections are placed as straight tiles.
- **Duration**: The average time required for the Duckiebot to cross an intersection and an upper limit (worst-case) on the time required. This metric is not really relevant as long as the Duckiebot is not moving significantly slower than during lane-following. This is the case for our implementation, which is why we didn’t measure it.

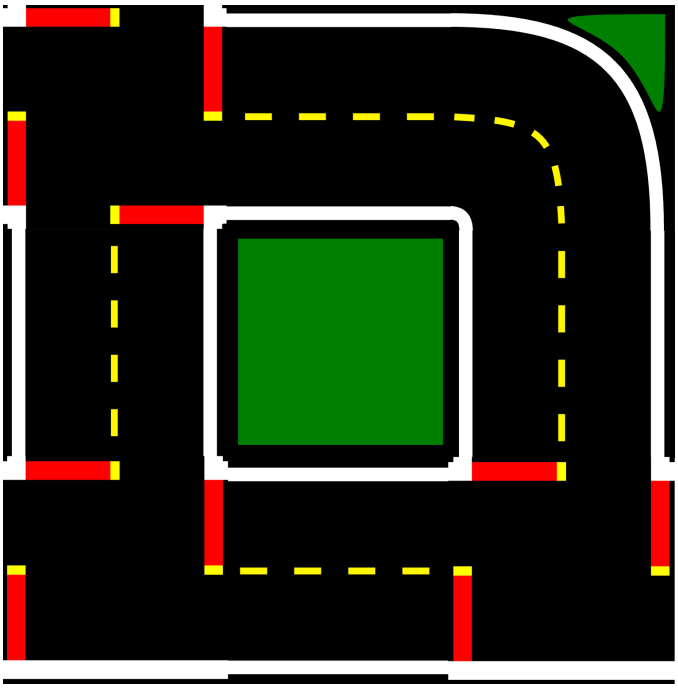


Figure 6: Performance testing setup

Contribution / Added Functionality

Logical Architecture

At the moment of its activation our subsystem assumes that the Duckiebot is located in front of an intersection. As soon as it is activated it localizes the Duckiebot on the intersection and calculates a lane pose to forward to the lane controller. This section will explain in detail how these tasks were implemented.
Our subsystem assumes the availability of color-normalized images. Furthermore, it generates lane poses to feed to the existing lane controller in order to avoid having to reimplement a controller.

Localization

The localization uses the red stoplines on intersections as absolute reference points. If the Duckiebot knows which stopline on the intersection it sees it can calculate the pose (position and orientation) of that stopline, and from that the Duckiebot will be able to calculate its absolute position on the intersection.
Initial tests showed that at least one stopline is nicely visible over the duration of most turns. Furthermore, we found that even if the implementation assumes 4 stoplines it generalizes fairly well to a 3-way-intersection (except some turns). To bridge parts on the path where there is no stopline visible to a satisfactory degree a relative pose estimate is calculated by integrating the commands sent to the motors.
The following sections assume that there is always a “last state” of intersection poses available. In the beginning the subsystem is initialized with a pose of about 20 cm in front of the intersection.
We define several coordinate frames for the intersection (Int) and the stoplines (s_0, s_1, s_2, s_3). A complete illustration can be seen in figure 7.

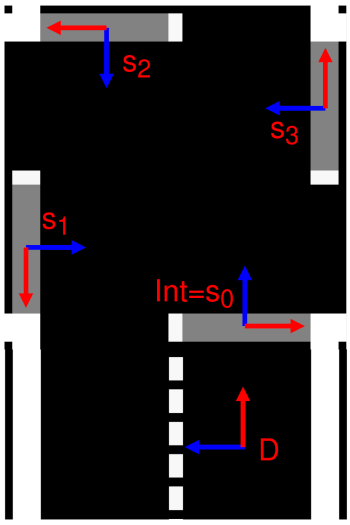


Figure 7: Defined coordinate frames. Fixed intersection coordinate frame (Int), stopline coordinate frames (s_0, s_1, s_2, s_3), and Duckiebot coordinate frame in initially assumed position (D).

Image pre-processing

The color-normalized images need some processing before they can be used for localization. Firstly, the image is rectified. The rectified image is then transformed into a "birdseye"-perspective using the homography from the extrinsic calibration. The transformation is visualized in figure 8.

(a) Original camera view (b) rectified camera view (c) birdseye view

Figure 8: Camera image transformation of the birdseye_view_node

Stopline detection

The localization system is working with the image in birdseye perspective. As the entire localization is based on detecting the stoplines on the intersection, the first step is to filter out red pixels. This is done using the same technique as the line_detector, which means the image is converted to HSV space and fixed color ranges are filtered.
These red pixels are clustered using the DBSCAN algorithm. Each detected cluster is considered a potential stopline that is in view of the Duckiebot.
Calculating the stopline pose is rather simple since we are working with an image in birdseye perspective. The position of the stopline is simply the mean of the pixels of its cluster. For

calculating the orientation a line can be fitted to the clustered pixels, which will naturally be parallel to its x-axis, because of the rectangular shape of the stoplines. This is still ambiguous, however, since the direction of the x-axis cannot be inferred from the pixels alone. For now, any one of the two orientations is chosen. We now have a pose for each cluster of pixels in the axle frame of the Duckiebot.
As previously mentioned it is important to know which cluster corresponds to which stopline on the intersection. This is why the pose for every of the (up to) four stoplines of an intersection is predicted based on the current assumed position of the Duckiebot. Each pose calculated from the clusters is then assigned to one of the predicted stoplines using a nearest-neighbor classifier that matches the stoplines based on their positions. The classification is visualized in figure 9.
Now the previously ambiguous orientation can be determined by checking which of the two possibilities is closer to the orientation of the predicted stopline.
From the complete stopline pose in the axle coordinate frame, a pose estimate of the Duckiebot in the intersection coordinate frame can be calculated with a coordinate frame transformation.
Lastly, to indicate how reliable a pose estimate calculated from a specific pixel cluster is, a quality value is calculated based on the number of pixels that are part of the cluster. This will be important for the next section.

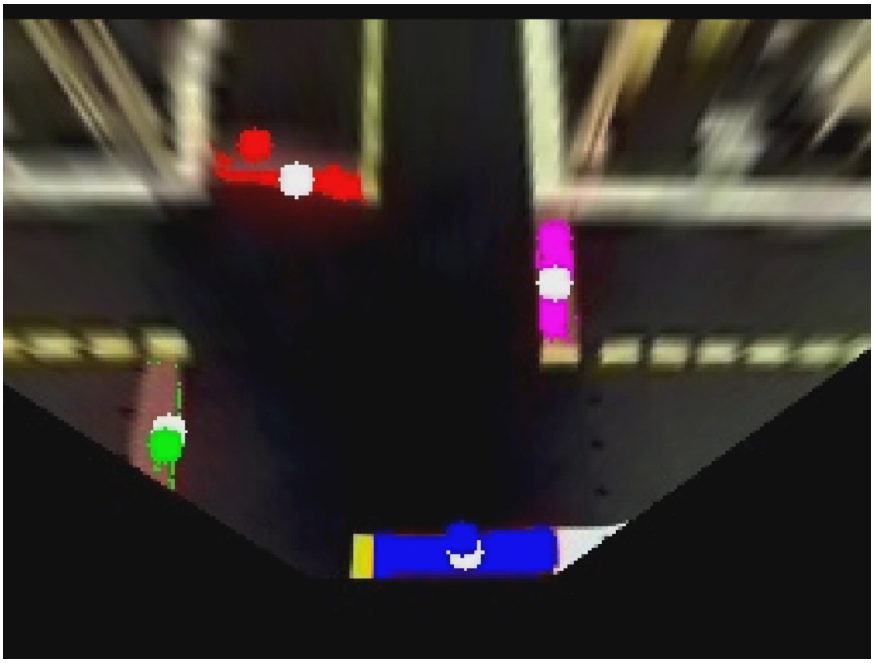


Figure 9: Stopline clustering and classification. The colored dots are the predicted stopline positions. The white dots are the stopline position calculated from the clusters. The pixels of each cluster are colored based on the nearest predicted stopline.

Stopline filtering

The pose estimates generated from the pixel clusters vary in quality. When a stopline is right in front of the Duckiebot, the estimate will be more accurate than when the stopline is far away or

at the very edge of the image, where it will be more distorted and blurred. For this reason the pose estimates should be combined somehow to create the most likely pose of the Duckiebot on the intersection.
We implemented a simple stateless filter with two available policies after removing estimates that are below a certain quality threshold:

- Maximum quality: The estimate with the highest quality is selected.
- Weighted average: The estimates are weighted proportionally to their quality and the average is calculated.

Compared to maximum quality, the weighted average approach produces less “jumpy” estimates, which is why we opted for that policy in the end. Figure 10 visualizes an example of an averaged pose estimate.
Although there are much more sophisticated techniques for this step (e.g. EKF), this simple implementation proved to provide pretty solid results already, which is why we decided to stick with it.

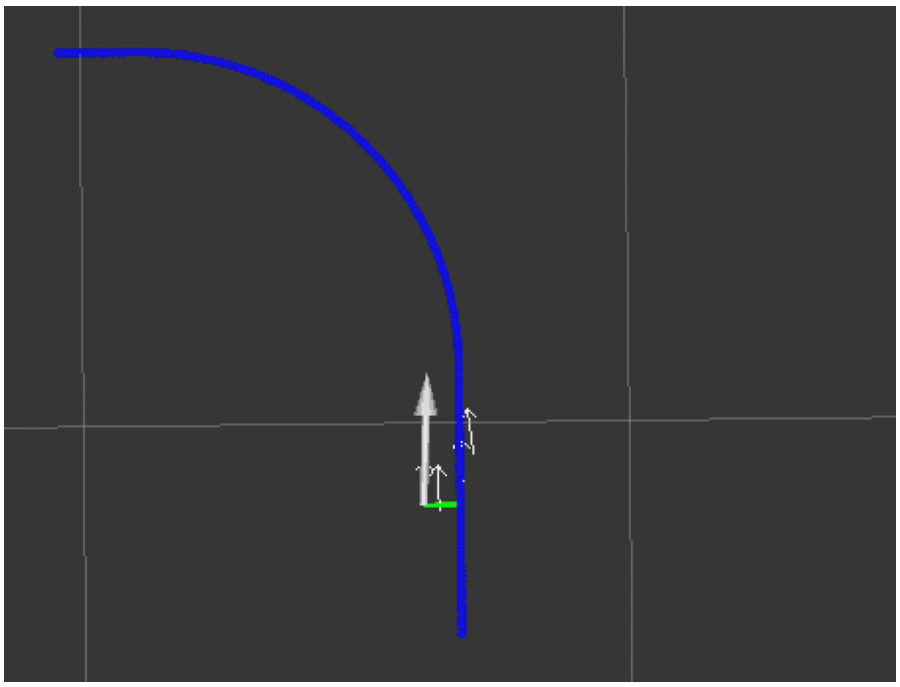


Figure 10: Stopline filtering and pose estimation. Each small arrow represents one pose estimate from a cluster of red pixels. The big arrow is the weighted average of the pose estimates. The blue line visualizes the virtual lane (see next section).

Lane projection

To convert an intersection pose to a lane pose (lateral offset d and angle offset φ) a lane is necessary. Since there is no lane on an intersection, we use something we call a “virtual lane”, which is a precomputed reference path laid over the intersection tile. The reference path is made up of discrete points and includes the corresponding angle of the tangents and the curvature at these points. To calculate the lane pose, the closest point on the trajectory to the current pose is considered. The lateral offset is the (Euclidean) distance to that point. The sign

of the lateral offset depends on which side of the path the Duckiebot is. The angle offset is the angle between the tangent of the closest point and the Duckiebots x-axis. The resulting lane pose can be used by the existing lane controller.

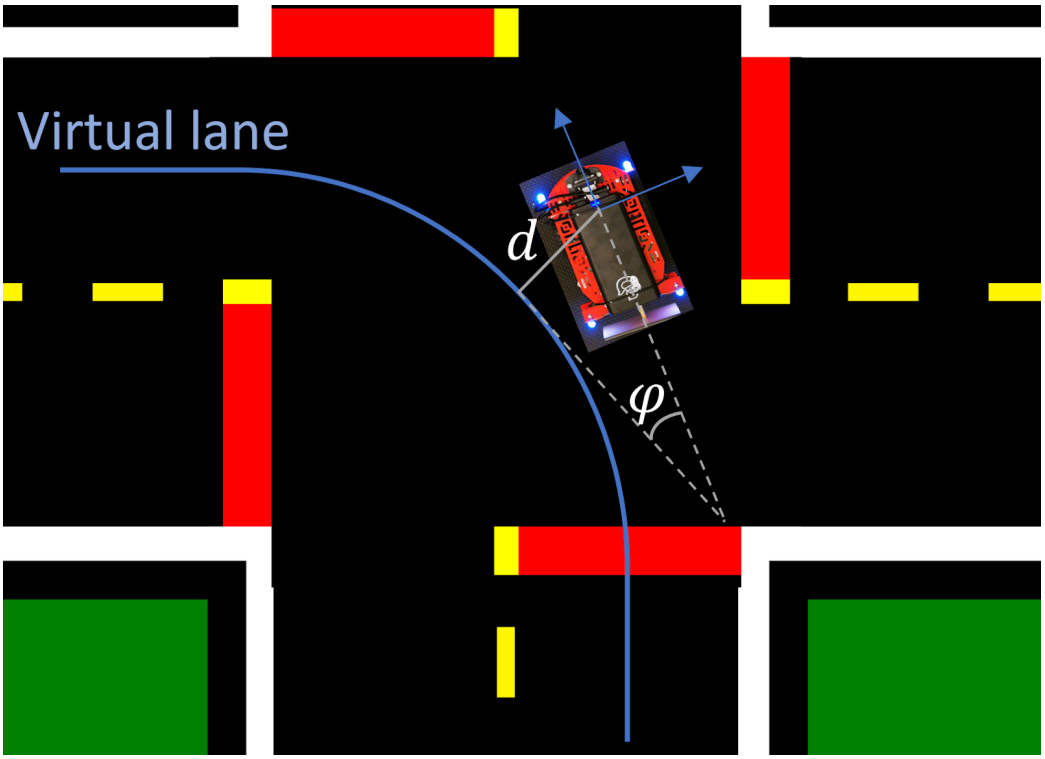


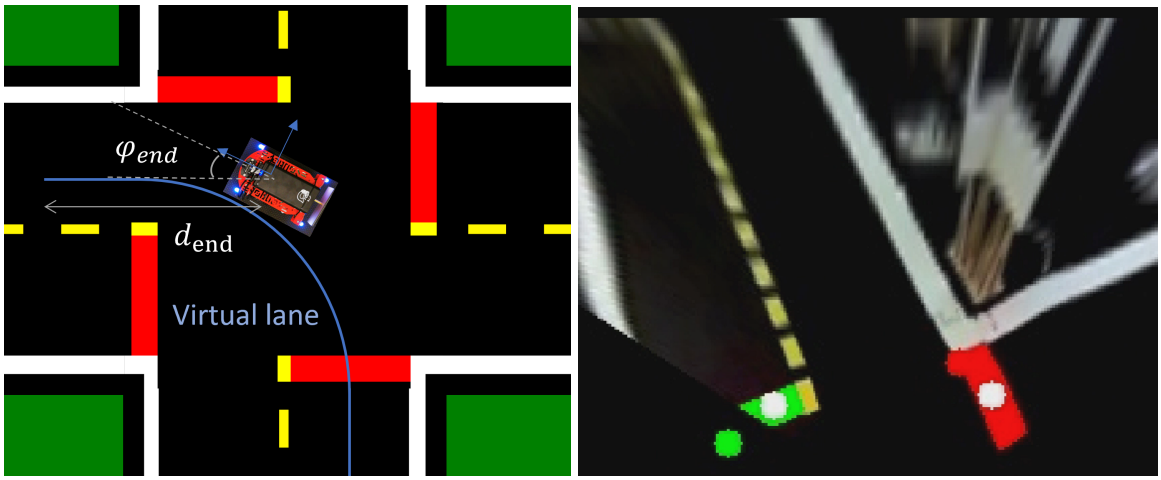
Figure 11: Illustration of the virtual lane and its relationship with the lane pose for a left turn.

Handover to lane following

As soon as the following two conditions are met, control is handed back to the lane following controller:

- Distance between the Duckiebot and the end of the virtual lane is less than a predefined distance.
- Angle between the Duckiebot and the end of the virtual lane is less than a predefined angle.

The conditions are visualized in figure 12.



(a)Handover condition (b) Possible perspective when handing over

Figure 12: Switching back to lane following

Relative pose from motor commands

By integrating the commands sent to the motor a relative pose estimate can be calculated. This pose would diverge from the real pose due to integral drift so it is reset every time a pose is successfully estimated via vision. The integrated pose is used to predict the movement of the stoplines between frames, reducing the likelihood of misclassification.
The raw integrated pose is only used to generate the next command when the vision algorithm fails to produce a reliable pose estimate.

Software Architecture

This section goes into detail about the implementation of everything introduced in the previous section.

Integration with the existing system

The added subsystem is activated during the INTERSECTION_CONTROL state. It is subscribing to color-normalized image generated by the anti_instagram_node. To cross the intersection it continuously publishes lane poses to the lane_controller_node. To receive the relative pose estimate it subscribes to the velocity_to_pose_node. As soon as the handover conditions are met, the INTERSECTION_DONE signal is published. Figure 13 illustrates the relationship.

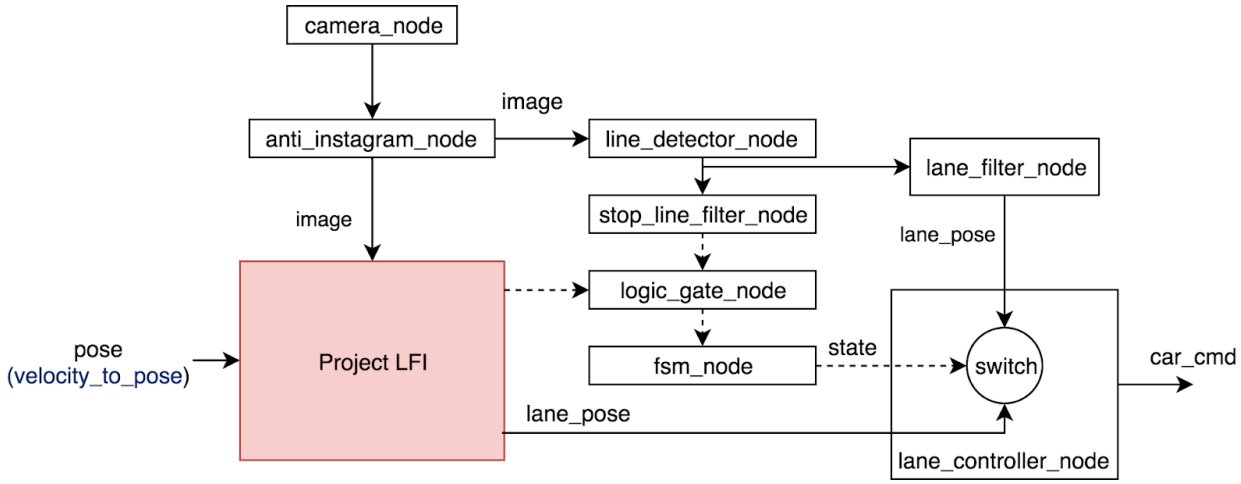


Figure 13: Integration with the existing system

ROS Nodes

This section lists for each added ROS-node the tasks it implements and the topics it subscribes and publishes to. Furthermore, it mentions noteworthy implementation details. Note that each node has verbose publication topics that are not actually needed for the pipeline to work. The “most important” publications are bold.

Birdseye_view_node

Tasks	• Image pre-processing
Subscriptions	• camera_node/camera_info • anti_instagram_node/corrected_image/compressed • birdseye_node_node/switch
Publications	• birdseye_node_node/verbose/rectified/compressed • **birdseye_node_node/image_out/compressed**

Table 1: Tasks, subscriptions and publications of the birdseye_view_node

The birdseye node takes images from the anti-instagram node and transforms them into birdseye perspective. It makes heavy use of the scaled_homography module, which initially takes the homography from extrinsic calibration, but modifies it to perform a transformation into birdseye perspective. It also recalculates the homographies when the resolution changes.

Localization_node

Tasks	• Stopline detection • Stopline filtering
Subscriptions	• camera_node/camera_info

	● birdseye_node/image_out/compressed ● velocity_to_pose_node/pose ● localization_node/switch ● localization_node/reset ● fsm_node/mode
Publications	● localization_node/verbose/clustering/compressed ● localization_node/verbose/red_filter/compressed ● localization_node/stoplines_measured ● localization_node/stoplines_predicted ● localization_node/pose_estimates ● **localization_node/best_pose_estimate** ● /[veh_name]/coordinator_node/intersection_go

Table 2: Tasks, subscriptions and publications of the localization_node

The localization node takes images in birdseye perspective and tries to infer the Duckiebot's position in the intersection frame. To keep the localization node readable several modules were created:

- Intersection_model: This module provides a model of the duckietown intersection. In essence, given a position of a Duckiebot in the intersection frame, it can calculate where the stopline poses should be in the axle frame.
- Stopline_detector: Contains all functionality for stopline detection.
- Stopline_filter: Contains all functionality for stopline filtering.

For coordinate frame transformations the ROS tf package is used.
The integration of motor commands is already implemented by the node "velocity_to_pose_node". Unfortunately this node assumes that the Duckiebot has no inertia and as a consequence reports more rapid orientation changes than present in reality. To mitigate this we added a damping factor to our node which simply scales down the change of the orientation. This is by no means an ideal solution, but makes the output from velocity_to_pose_node usable. Using the pose estimates from the integrated motor commands for the stopline prediction can be enabled with the parameter "/localization_node/integration_assisted_clustering".

Virtual_lane_node

Tasks	● Lane projection ● Handover to lane following
Subscriptions	● localization_node/best_pose_estimate
Publications	● **lane_controller_node/intersection_navigation_pose** ● logic_gate_node/intersection_done_and_deep_lane_off ● virtual_lane_node/verbose/trajectory ● virtual_lane_node/verbose/closest_point

Table: 3: Tasks, subscriptions and publications of the virtual_lane_node

The virtual_lane_node uses the best_pose_estimate from the localization node as the Duckiebot's pose. The precomputed virtual lanes are loaded as ROS parameters from a yaml file.

Virtual Lane Generation

There are three different virtual lanes:

1. Going straight
2. Turning left
3. Turning right

The file simple_track_generator.py was used to generate a yaml file for each one of these virtual lanes, which include discretized x- and y-coordinates as well as the tangent angle and the curvature at each point. The yaml files are then loaded with the launch file.

Performance Evaluation

Results

Success Rate

The success rate is calculated as follows:

$$rate_{success} = \frac{\sum successful\ crossing}{\sum successful\ crossing + \sum failed\ crossing}$$

As can be seen in Table 4, for every intersection the rate of successfully completed crossings is above 75%. This indicates that our approach generalizes quite well. However, it still lacks robustness. Except for the left turn, all the other values are below 90%, which is not as high as we had hoped.

	left turn	**going straight**	**right turn**
4-way intersection	91.9 %	89.6 %	79.2 %
3-way intersection		78.0 %	

Table 4: Success rate of each turn and intersection

Crossing Time

We decided that the crossing time is not worth measuring, because the Duckiebot always drives with a constant velocity through the intersection and should therefore also have a constant crossing time. The velocity is set to a slightly lower value than for normal lane following, in order for the delay to have smaller effect on the driven trajectory (see sectionWeaknesses).

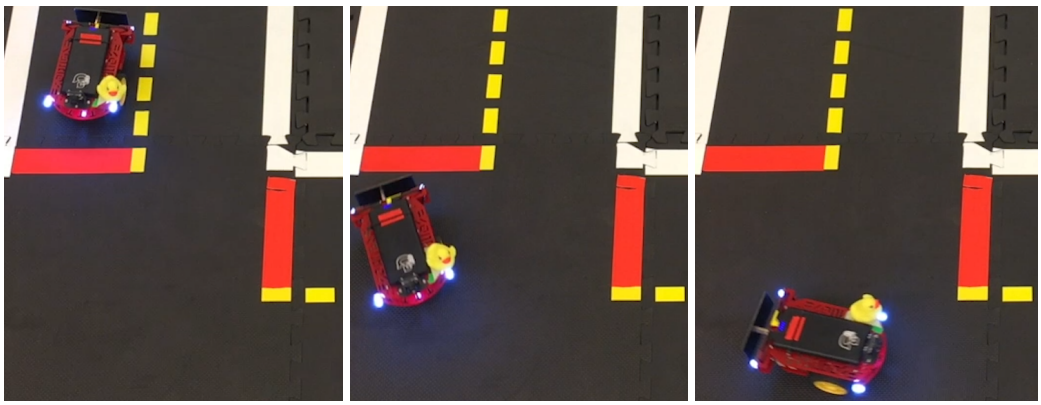
Strengths

We found some clear strengths in our approach. Firstly, the absolute pose estimate based on stoplines proved to be very accurate at a refresh rate of about 10 hz as long as the stopline is not too far away and most of it is visible. Specifically, the localization is pretty robust to initial pose variation. If a good chunk of the first stopline is visible it basically never fails. If the Duckiebot is so far into the intersection that the first stopline is not visible anymore the system fails if, in addition, it has a large orientation offset or no other stoplines are sufficiently visible. In our tests we did not observe the initial pose estimate to ever be a problem.

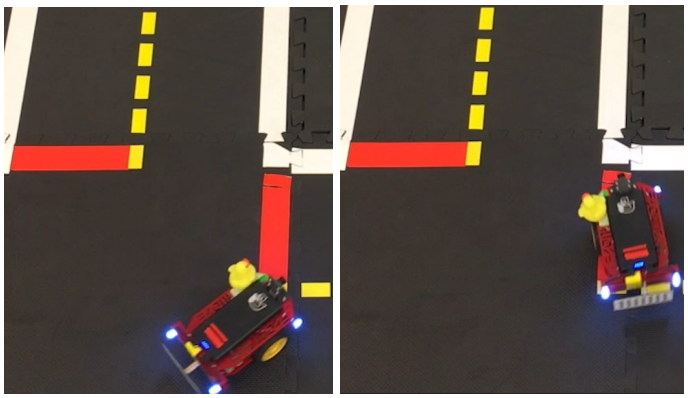
Weaknesses

The most often encountered causes of failure are:

1. The visibility of stoplines is sometimes not good enough. This can be problematic in the following scenarios:
    a. Right after the first stopline is out of view
    b. Last quarter of trajectories
2. The integrator node is sometimes not accurate enough to compensate for the missing absolute estimates.
3. We tried to find an approach as general as possible. In some specific scenarios, this may cause problems: In three-way intersections one out of four clustering measurements is constantly missing. This becomes an issue if the closest, and therefore most important, stopline is missing, as can be seen in figure 14.
4. Due to the computational complexity, especially of the DBSCAN clustering algorithm, a significant delay between 100 and 180 ms (depending on the amount of red pixels in the image) is introduced. This can lead to overshoot and a crossing of the opposite line, especially during sharp turns like the right turn, as can be seen in figure 15.
5. During our demonstration we seemed to have problems with other Duckiebots waiting at the intersection. Even though we tested this case and did not seem to encounter any problems during our tests, we have to concede that we might have missed something and that these cases require further investigation.

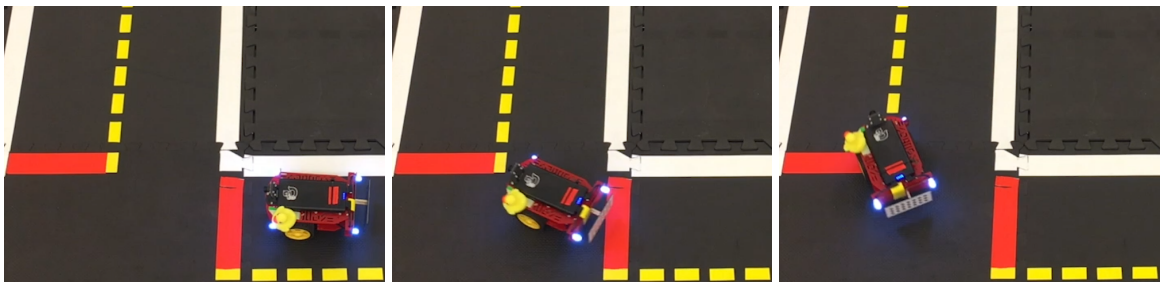


(a) First timestep (b) Second timestep (c) Third timestep

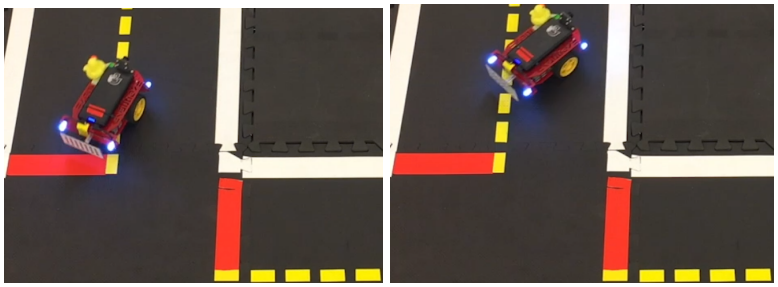


(d) Forth timestep (e) Fifth timestep

Figure 14: Failed left turn due to a missing stopline in a three-way intersection



(a) First timestep (b) Second timestep (c) Third timestep



(d) Forth timestep (e) Fifth timestep

Figure 15: Overshoot (with a successful recovery) in a right turn caused by delay

Conclusion and Future Work

The approach for localization on an intersection and generating commands for the lane controller introduced in this work seems promising, even though our implementation does not handle all turn types with satisfactory quality. Some ideas for improvement for future intersection navigation projects are:

- Use the knowledge of the april tags to customize different navigation approaches, based on what kind of intersection (three- or four-way) and what kind of turn (left, right, straight) the Duckiebot is facing.
- Add proper inertia to the wheel command integration to improve its accuracy and, in turn, get a pose estimate at higher frequency.
- Add a filter (e.g. EKF) to avoid noisy measurements and increase the general robustness.
- Avoid doing applying the homography to the full image to save computational resources: e.g. clustering red line segments and doing a bayes filter like for lane filter.
- Try different clustering algorithms.
- Generalize the localization to consider more features than stoplines, e.g. yellow lines or white lines just for one dimension, to improve the localization.

Appendix

- Our code and instructions on how to reproduce our demo can be found on out Github repository: https://github.com/duckietown-ethz/proj-lfi
- Videos of our demo, as well as other Notes or Presentations can be found on our Google Drive folder. Please contact an author or one of the supervisors to get access.