PART A

Montreal IFT6757 2020 Project Reports: Final Reports

Contents

UNIT A-1

Group name: Project report

Before starting, you should have a look at some tips on how to write beautiful Duckiebook pages.

The objective of this report is to bring justice to your hard work during the semester and make so that future generations of Duckietown students may take full advantage of it. Some of the sections of this report are repetitions from the preliminary design document (PDD) and intermediate report you have given.

1.1. The final result

Let's start from a teaser.

- Post a video of your best results (e.g., your demo video): remember to have duckies on the robots or something terrible might happen!

You might want to add as a caption a link to your instructions to reproduce to reproduce these results. Moreover, add a link to the readme.txt of your code.

1.2. Mission and Scope

Now tell your story:

Define what is your mission here.

1) Motivation

Now step back and tell us how you got to that mission.

- What are we talking about? [Brief introduction / problem in general terms]
- Why is it important? [Relevance]

2) Existing solution

- Describe the "prior work"

3) Opportunity

- What was wrong with the baseline / prior work / existing solution? Why did it need improvement?

Examples: - there wasn't a previous implementation - the previous performance, evaluated according to some specific metrics, was not satisfactory - it was not robust / reliable - somebody told me to do so (/s) (this is a terrible motivation. In general, never ever ever say "somebody told me to do it" or "everybody does like this")

- How did you go about improving the existing solution / approaching the problem? [contribution]

Examples: - We used method / algorithm xyz to fix the gap in knowledge (don't go

in the details here) - Make sure to reference papers you used / took inspiration from, lessons, textbooks, third party projects and any other resource you took advantage of (check here how to add citations in this document). Even in your code, make sure you are giving credit in the comments to original authors if you are reusing some components.

1.3. Background and Preliminaries

• Is there some particular theorem / "mathy" thing you require your readers to know before delving in the actual problem? Briefly explain it and links for more detailed explanations here.

Definition of link: - could be the reference to a paper / textbook - (bonus points) it is best if it is a link to Duckiebook chapter (in the dedicated "Preliminaries" section)

1.4. Definition of the problem

Up to now it was all fun and giggles. This is the most important part of your report: a crisp, possibly mathematical, definition of the problem you tackled. You can use part of the preliminary design document to fill this section.

Make sure you include your: - final objective / goal - assumptions made - quantitative performance metrics to judge the achievement of the goal

1.5. Contribution / Added functionality

Describe here, in technical detail, what you have done. Make sure you include: - a theoretical description of the algorithm(s) you implemented - logical architecture - software architecture - details on the actual implementation where relevant (how does the implementation differ from the theory?) - any infrastructure you had to develop in order to implement your algorithm - If you have collected a number of logs, add link to where you stored them

Feel free to create subsections when useful to ease the flow

1.6. Formal performance evaluation / Results

Be rigorous!

• For each of the tasks you defined in you problem formulation, provide quantitative results (i.e., the evaluation of the previously introduced performance metrics)

• Compare your results to the success targets. Explain successes or failures.

• Compare your results to the "state of the art" / previous implementation where relevant. Explain failure / success.

• Include an explanation / discussion of the results. Where things (as / better than / worst than) you expected? What were the biggest challenges?

1.7. Future avenues of development

Is there something you think still needs to be done or could be improved? List it here, and be specific!

UNIT A-2

Instructions template

Before starting, you should have a look at some tips on how to write beautiful Duckiebook pages.

This is the template for the description of a what we should do to reproduce the results you got in your project. The spirit of this document is to be an operation manual, i.e., a straightforward, unambiguous recipe for reproducing the results of a specific behavior or set of behaviors.

It starts with the "knowledge box" that provides a crisp description of the border conditions needed:

- Duckiebot hardware configuration (see Duckiebot configurations)
- Duckietown hardware configuration (loops, intersections, robotarium, etc.)
- Number of Duckiebots
- Duckiebot setup steps

For example:

KNOWLEDGE AND ACTIVITY GRAPH

> **Requires:** Duckiebot in configuration `DB19`
>
> **Requires:** Duckietown without intersections
>
> **Requires:** Camera calibration completed

2.1. Video of expected results

First, we show a video of the expected behavior (if the demo is successful).

Make sure the video is compliant with Duckietown, i.e. : the city meets the appearance specifications and the Duckiebots have duckies on board.

2.2. Laptop setup notes

Does the user need to do anything to modify their local laptop configuration?

2.3. Duckietown setup notes

Here, describe the assumptions about the Duckietown, including:

- Layout (tiles types)
- Infrastructure (traffic lights, WiFi networks, ...) required
- Weather (lights, ...)

Do not write instructions on how to build the city here, unless you are doing something very particular that is not in the Duckietown operation manual (unknown ref opmanu-

al_duckietown/duckietowns)

warning next (1 of 7) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckietown/duckietowns'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

. Here, merely point to them.

2.4. Duckiebot setup notes

Write here any special setup for the Duckiebot, if needed.

Do not repeat instructions here that are already included in the Duckiebot operation manual (unknown ref opmanual_duckiebot/opmanual_duckiebot)

previous warning next (2 of 7) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/opmanual_duckiebot'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

2.5. Pre-flight checklist

The pre-flight checklist describes the steps that are sufficient to ensure that the demo will be correct:

Check: operation 1 done

Check: operation 2 done

2.6. Instructions

Here, give step by step instructions to reproduce the demo.

Step 1: XXX

Step 2: XXX

Make sure you are specifying where to write each line of code that needs to be executed, and what should the expected outcome be. If there are typical pitfalls / errors you experienced, point to the next section for troubleshooting.

2.7. Troubleshooting

Add here any troubleshooting / tips and tricks required, in the form:

> **Symptom:** The Duckiebot flies

Resolution: Unplug the battery and send an email to info@duckietown.org

> **Symptom:** I run `this elegant snippet of code` and get this error: `a nasty line of gibberish`

Resolution: Power cycle until it works.

2.8. Demo failure demonstration

Finally, put here video of how the demo can fail, when the assumptions are not respected.

You can upload the videos to the Duckietown Vimeo account and link them here.

UNIT A-3

Cross Pro-Duck: Project report

3.1. The final result

The final best results of the overall algorithm are shown in the videos below. As can be seen, it is possible to achieve a visual-servoing feedback loop in intersection crossing using this algorithm. Unfortunately, this is currently limited to right turns, crossing straight, and red line approaching/lane following. This approach is also currently not very robust.

Please refer to the instructions to reproduce to reproduce these results.

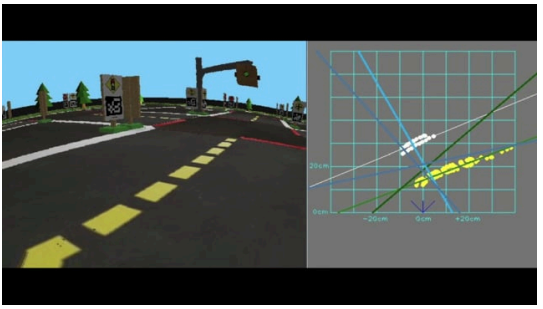


Figure 3.1. Successful Right Turns.

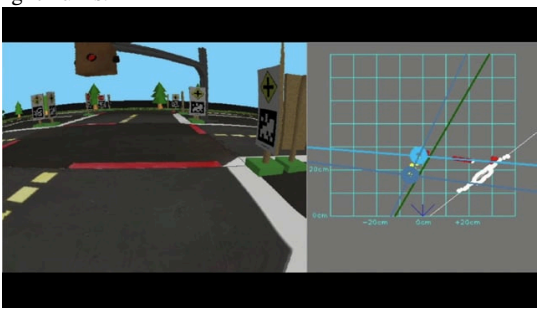


Figure 3.2. Successful Lane Following with Red Line Approach.

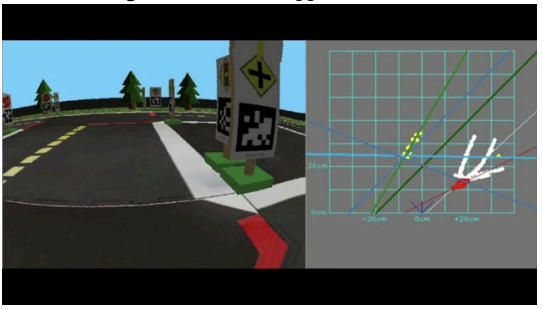


Figure 3.3. Successful Straight Crossing.

3.2. Project Goals

Overall Goal: Apply visual servoing techniques in order to construct a feedback control loop for duckietown tasks such as intersection crossing and lane following.

In this project, we apply computer vision techniques to extract useful visual information to control the motion of the duckiebots. Specifically, we want to guide the duckbiebot based on the mutual information captured in the reference and current images. The mutual information is usually some aligned visual features between the two images. Typically, the salient features are identified from the reference and target separately by a feature extractor and they are matched by a feature matcher. The matched features would be exploited to compute helpful information to assist the duckiebots in the intersection crossing tasks.

3.3. Background and Preliminaries

1) Visual Servoing

Visual servo control is a field of study focused on controlling robots based on the given visual data. The goal of visual servo control is to minimize the differences between a set of reference features and a set of observed features5:

$$e(t) = \mathbf{F}(m(t), \theta) - \mathbf{F}^*$$

where $m(t)$ is the observed image at time t, $\mathbf{F}$ represents the feature extractor and θ is a set of parameters for the feature extractor. On the other hand, $\mathbf{F}^*$ is a set of the desired (reference) features. In our project, $\mathbf{F}^*$ is defined as $\mathbf{F}(m^*, \theta)$ where m^* is our checkpoint image. Visual servoing encompasses both approaches where the features are present in the image space, and those that are present in the projected space as in this project.

2) Image Alignment in Computer Vision

Visual features are defined differently across visual servoing tasks. Most work relies on the image points to conduct visual servoing. For example, 3 and 4 use image coordinates to define the image features. Other types of visual features (such as lines, image moments, moment invariants) are also used in visual servoing tasks 6. In this project, we explore the possibility of combining different types of visual features to obtain feasible input for the control model. Specifically, we utilize the point and line features from the observed and reference images to construct an affine matrix as input for the control model. Our objective functions for feature alignment is:

$$e(t) = \big(\mathbf{P}(m(t), \theta) - \mathbf{P}(m^*, \theta)\big) + \big(\mathbf{L}(m(t), \theta) - \mathbf{L}(m^*, \theta)\big)$$

where $\mathbf{P}$ is the point feature extractor and $\mathbf{L}$ is the line feature extractor.

3) Visual Servo Control

The controller used in visual servoing applications depends greatly on the type of features extracted by the vision algorithm. In image-based (IBVS) approaches, where features are matched in the image space, the controller revolves around transforming image space dynamics into real world dynamics. This involves building a so called *inter-*

action matrix, which relates the velocity of features in image space to the velocity of the camera in the real world. This task can be challenging depending on the system at hand and is highly dependent on the quality and consistency of features detected.

An alternative is the pose-based (PBVS) approach, which uses computer vision to directly estimate the current pose of the robot in the real world. Traditional controls approaches can then be taken to provide inputs.

The approach that we take in our algorithm is a hybrid approach which incorporates features of both IBVS and PBVS. The output of the vision algorithm is a homography between the current and target images projected in the ground plane. Although we are using image features directly, the fact that they are projected onto the ground plane means that we can treat the homography between them as an $SE(2)$ target for the duckie to reach. Therefore, the required controller becomes significantly more simple as it deals with the vision output as a direct target pose.

3.4. Overview

1) Visual Path Following

Our primary contribution toward Duckietown is providing its Duckizens the ability to follow trajectories given to them in the form of sequence of images. Currently, duckies can only perform certain predefined operations well such as following a specific lane. They rely on specific hard-coded rules such as always being to the right of the yellow line and to the left of the white line in order to achieve those operations. Because of this, they especially struggle to cross intersections where there are no generalizable line patterns. Our goal is to give them the power to follow any kind of path they want to with ease by just providing them a set of path images. These images act as a sample for the actions to take and our approach is expected to be robust to different environments. For this proof-of-concept project, we restrict our scope to evaluating our approach on lane following and crossing intersections via going straight and turning right.

Existing Implementation for Intersection Navigation:

Currently, intersection crossing is done using an open-loop control. When a duckie arrives at a crossing, identified by red lines, it is fed a series of open-loop inputs corresponding to the approximately ideal inputs needed to accomplish a turn in a desired location. The goal of these open loop controls is to move the duckie to a close enough location to the desired lane entrance that the standard lane controller will be able to pick up the correct white and yellow line to continue lane following. This approach, like most open-loop approaches, is not robust and can easily lead to duckies entering incorrect lanes or flying out of the intersection altogether. As such, the visual servoing approach was chosen as a way to include closed-loop controls in the intersection crossing task in order to improve the robustness and safety for all duckies going through an intersections.

2) Architecture

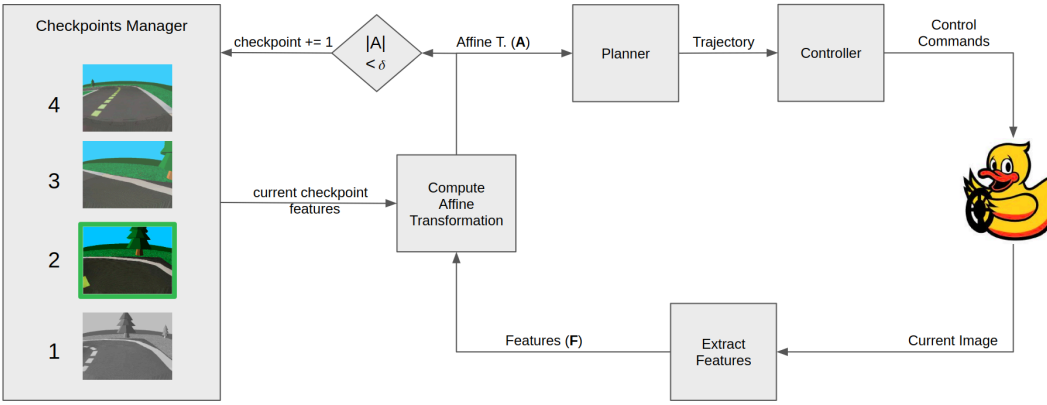


Figure 3.4. Overview of Cross Pro-Duck.

Our algorithm's role is to help a duckie reach its destination by iteratively guiding it through a set of checkpoints. To achieve this, our architecture consists of three major components: Vision-based Feature Extractor, Duckie Controller, and Checkpoint Manager. Navigation starts with the checkpoint manager loading a set of pre-recorded checkpoint images and setting the first checkpoint image as the initial target. The checkpoint image and the duckie's current image are then sent to the feature extractor which uses algorithms such as line detection, intersecting points detection, pairwise-matching etc. (explained in more detail in the next section) to compute an affine transformation matrix. This matrix estimates the direction and distance for the duckie to move in order to match its view with the checkpoint. The planner then takes this information to estimate a trajectory for the duckie to reach the checkpoint and the controller is tasked with following this trajectory. Throughout the duckie's journey, the checkpoint manager keeps track of its progress and switches checkpoints as the duckie passes through them.

3.5. Algorithms

1) Infrastructure

We take advantage of the existing ROS-based Duckietown infrastructure to support various algorithms in our approach. We modify the default `LineDetectorNode`, `LaneFilterNode`, and `LaneControllerNode` ROS nodes to seamlessly integrate our solution in the Duckietown stack. We also implement various visualizations in ROS which help us keep track of the state of duckie's motion. We list down the algorithms implemented within the major components of our solution below:

2) Computer Vision

- Detecting color coordinates
- Ground projection
- Distance-based filtering
- Clustering
- Regression

- Lines parallelization
- Intersection computation
- Line matching
- Point matching
- Affine (SE2) Transformation

In the vision component of this project, we compute the SE2 transformation matrix based on three lines (the red, yellow, white lanes) and two points (the intersection of the red+yellow lanes and the intersection of the red+white lanes).

In order to compute a feasible affine matrix for the control model, we rely on several assumptions about the ground projected lanes:

- the ground projected white lane and the ground projected yellow lane are parallel
- the ground projected white lane and the ground projected yellow lane are perpendicular to the red lane
- the distance between the ground projected white lane and the ground projected yellow lane are always constant

These assumptions are crucial because they enforce the transformation matrix between the reference image and the observed image contains purely rotation and translation information (no scaling and shearing information).

Knowing these assumptions, the computer vision component of this project breaks down into two major parts: obtain the ground projected lanes following these assumptions and compute an affine matrix by matching lines and points.

To compute the ground projected lines, we first obtain the lane coordinates for each color, project them onto the ground plane and filter out the points that are too far from the duckiebot. The ground projected points are grouped based on their proximity to the other points using agglomerative clustering technique with single linkage between the clusters. Huber regressor is used to render line regression on the lane coordinates for each color and for each cluster. In theory, the three lines computed should adhere to the assumption we made about the ground projected lanes. However, the camera used on the duckiebots does not provide the depth information. Thus, we can alter the slopes and intercepts of the lines to make these assumptions hold. (Note: In the following model for computing affine transformation matrix, it does not require us to execute the line parallelization algorithm since it takes only one line and one point as inputs. However, the following model is faultless only when the assumptions listed above hold.)

The affine matrix is computed by a pair of matched lines and a pair of matched points. Let $\boldsymbol{l}_{\text{ref}}$ and $\boldsymbol{l}_{\text{t}}$ be the matched reference and observed lines respectively and $\boldsymbol{p}_{\text{ref}}$ and $\boldsymbol{p}_{\text{t}}$ be the matched points respectively. Lines and points are defined by their homogeneous coordinates (i.e. $\boldsymbol{l} = (a, b, c)$ and $\boldsymbol{p} = (x, y, 1)$; for every point $\boldsymbol{p}$ on line $\boldsymbol{l}$, we have $\boldsymbol{l} \cdot \boldsymbol{p} = 0$). The slope of line $\boldsymbol{l}$ can be computed as $m(\boldsymbol{l}) = -\frac{a}{b}$. The affine matrix without scaling and shearing components is defined as

$$\mathbf{A} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & t_x \\ \sin(\theta) & \cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

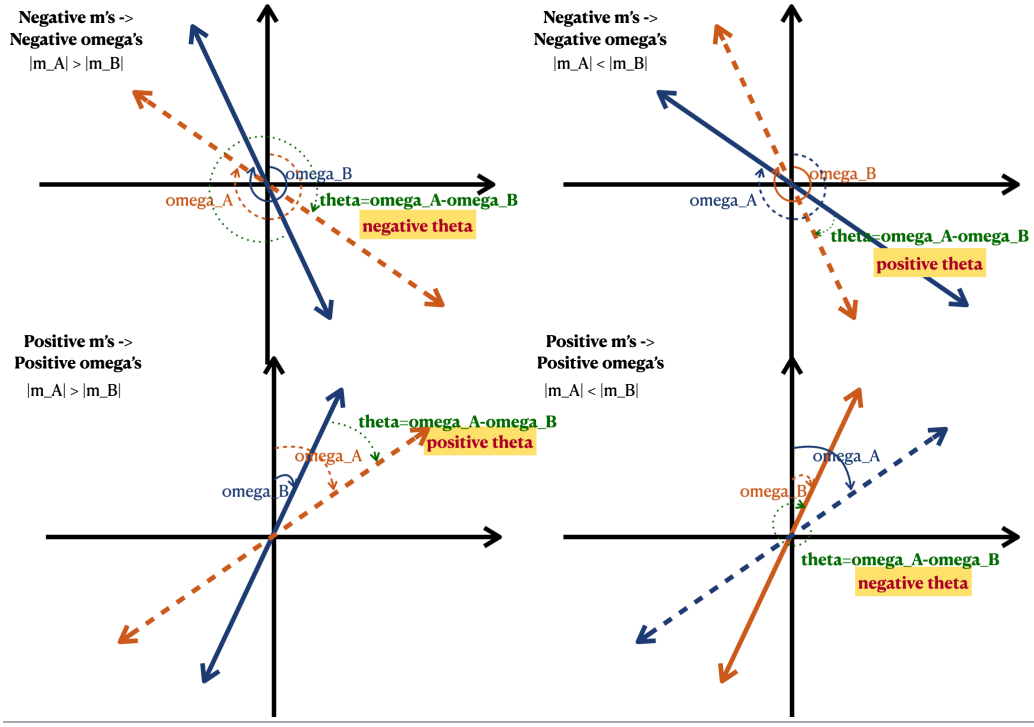
Given l_{ref}, l_{t} and p_{ref}, p_{t}, the transformation matrix from the ground projected reference plane to the ground projected current plane can be computed by defining θ as the angle of rotation between l_{ref} and l_{t}

$$\theta = \arctan(m(l_{\text{t}})) - \arctan(m(l_{\text{ref}}))$$

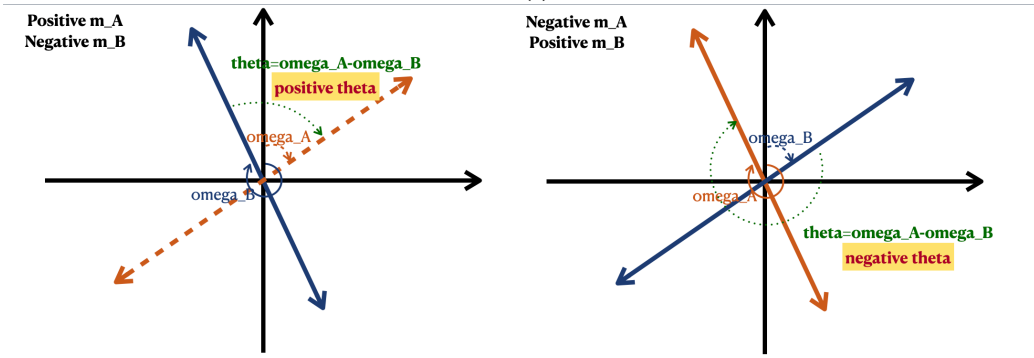
and t defined as the displace between p_{ref} and p_{t}

$$t = p_{\text{t}} - p_{\text{ref}}$$

The following plots illustrate that the relationship between θ and the lines' slopes.



(a)



(b)

Figure 3.5. Overview of theta computation.

How to compute (l_{ref}, l_{t})? The matched lines are determined according to the line regression results. $l_{\text{ref}}, l_{\text{t}}$ are the yellow lines' homogeneous coordinates if the yellow lines are detected. Otherwise, we use the red lines coordinates. Lastly, we use the white lines coordinates if the red lines are also not detected.

How to compute (p_{ref}, p_{t})? Let line $l_1 = (a_1, b_1, c_1)$ and line $l_2 = (a_2, b_2, c_2)$. Let the intersection of l_1 and l_2 be $(x_{\text{intercept}}, y_{\text{intercept}})$. We have

$$a_1 x_{\text{intercept}} + b_1 y_{\text{intercept}} + c_1 = 0 \; a_2 x_{\text{intercept}} + b_2 y_{\text{intercept}} + c_2 = 0$$

We can solve for $(x_{\text{intercept}}, y_{\text{intercept}})$ using linear system:

$$\begin{vmatrix} -c_1 \\ -c_2 \end{vmatrix} = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \begin{vmatrix} x_{\text{intercept}} \\ y_{\text{intercept}} \end{vmatrix}$$

Assume we have a function named `get_intersections` that computes the x- and y-intercept of two homogeneous lines. The following pseudo-code depicts the algorithm for computing the matched points (p_{ref}, p_{t}):

```
function compute_matching_points(ref_lines, curr_lines, ref_white_im,
curr_white_im):
    if matched red and yellow lines are detected:
        ## return their intersections
        return get_intersections(ref_red_l, ref_yellow_l),
               get_intersections(curr_red_l, curr_yellow_l)
    else if matched red and white lines are detected:
        ## return their intersections
        return get_intersections(ref_red_l, ref_white_l),
               get_intersections(curr_red_l, curr_white_l)
    else if only matched white lines detected:
        # note that white_im is the masked image of the white lane
        # note that good_feature_to_track returns a list of with de-
creasing feature quality

        ref_corners, curr_corners = good_feature_to_track(ref_white_im,
curr_white_im)
        iterate over corners and compute the corners' distances from
white_l:
            if a pair of corners whose distance from white_l is close
to 0:
                project the corners to ground plane and return them

    return NoMatchedFound
```

Note that the distance between a point and a line is the dot product of their homogeneous coordinates, and the `good_feature_to_track` function is the implementation of the published work 8. The image below shows an example of computing yellow/red intersection point given yellow and red lines.

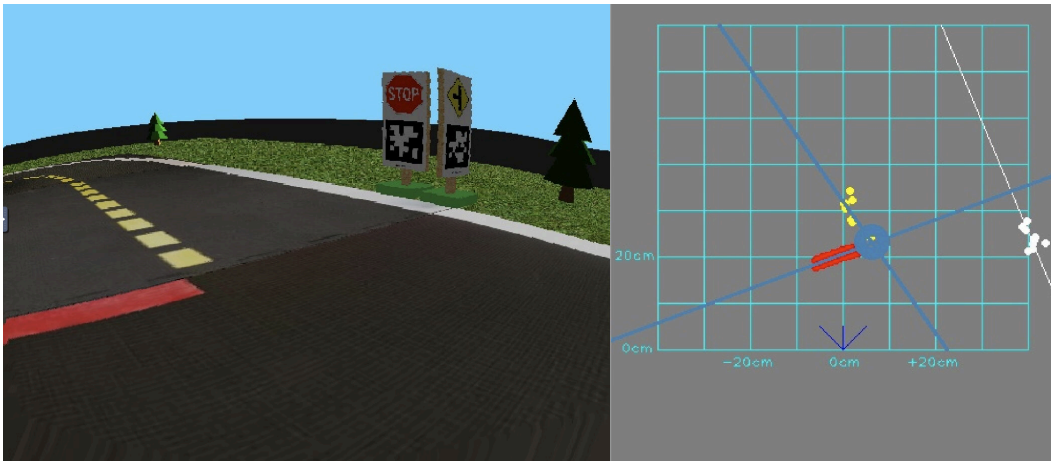


Figure 3.6. Example of line and point matching.

If only one pair of matched line is detected (no matched points detected), then the rotation matrix would be returned. If no matched point nor matched line is detected, then nothing would be returned from the vision module and the control model would handle the missing detection scenario.

3) Planning

The planner only knows about and only receives the $\boldsymbol{SE(2)}$ transformation in the form

$$\mathbf{X} = \begin{bmatrix} \mathbf{C}_{tc} & \mathbf{r}_c^{oq} \\ \mathbf{0} & 1 \end{bmatrix} \in \boldsymbol{SE(2)},$$

where $\boldsymbol{t}$ denotes the target frame of reference, $\boldsymbol{c}$ the current frame of reference, $\boldsymbol{q}$ the origin in the target frame and $\boldsymbol{o}$ the origin in the current frame. $\mathbf{C}$ is the rotation matrix between the current and target frame and $\mathbf{r}$ is the translational component between the current origin and the target origin. With this definition, the planner takes the desired transformation matrix and simply uses it as a target state, with the current pose is always taken to be "0", with $\mathbf{C} = \mathbf{1}$ and $\mathbf{r} = \mathbf{0}$. A toy example with the received target pose is shown below. Note that here $\boldsymbol{\theta} = \mathbf{0}$ is pointing up.

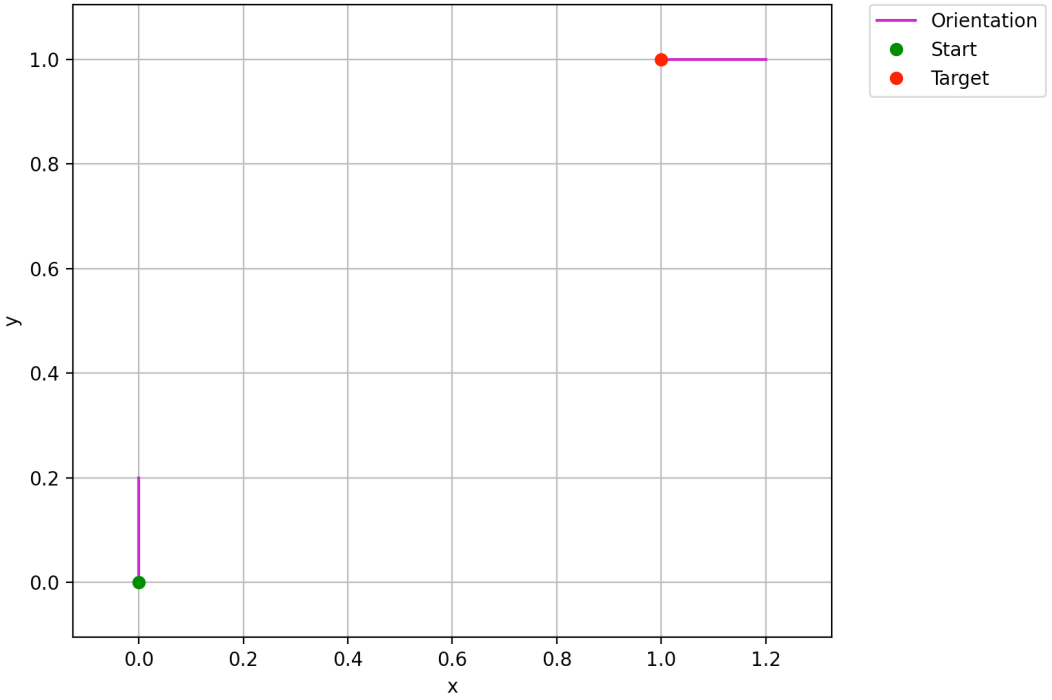


Figure 3.7. Sample current and target pose received by the planner.

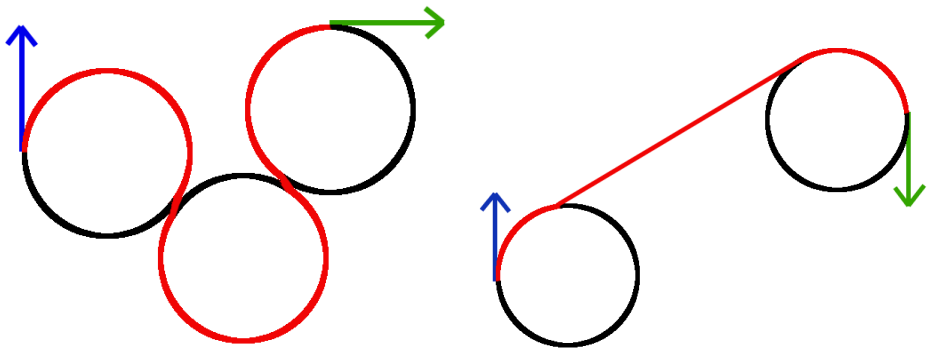
Dubins Path: Having a current and target state, the planner is built upon a dubins path algorithm. The duckiebot is modeled as Dubins Car, meaning it can only

- Go **S**traight at constant speed v,
- Turn **L**eft at maximum angular velocity ω, or
- Turn **R**ight at maximum angular velocity $-\omega$.

It was shown in 2 that the shortest path between two points for such a car would one of the combinations

$$\{\mathbf{LRL}, \mathbf{RLR}, \mathbf{LSL}, \mathbf{RSR}, \mathbf{LSR}, \mathbf{RSL}\}.$$

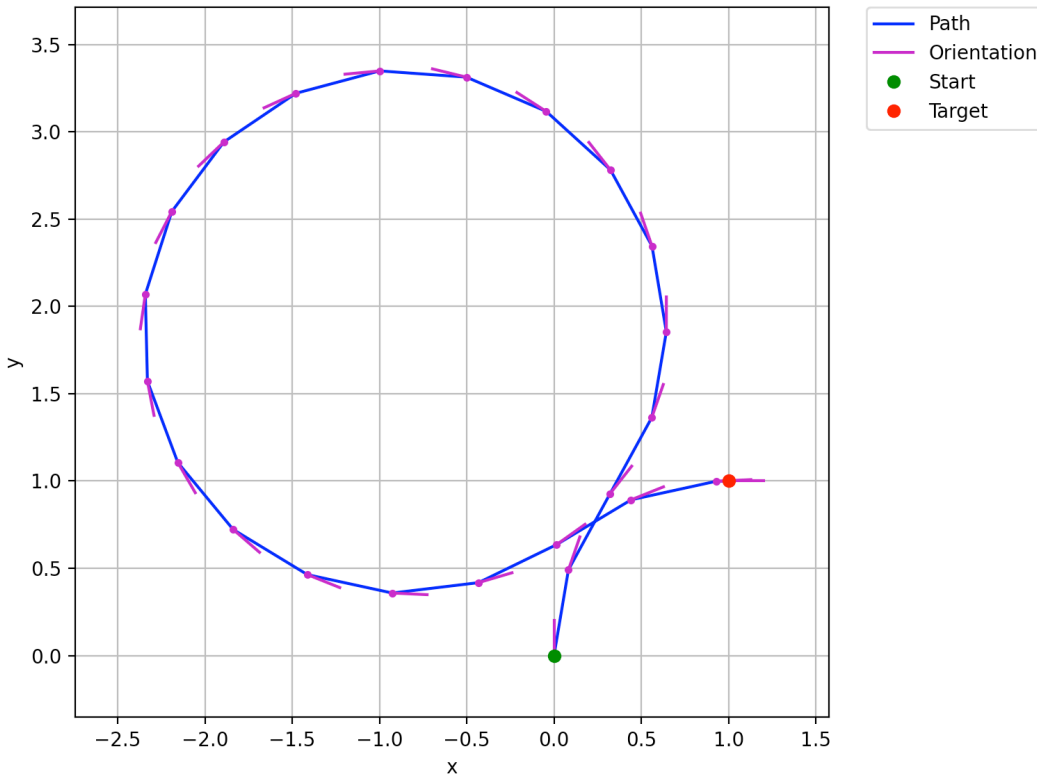
Example paths are shown in the images below, which are taken from 1.



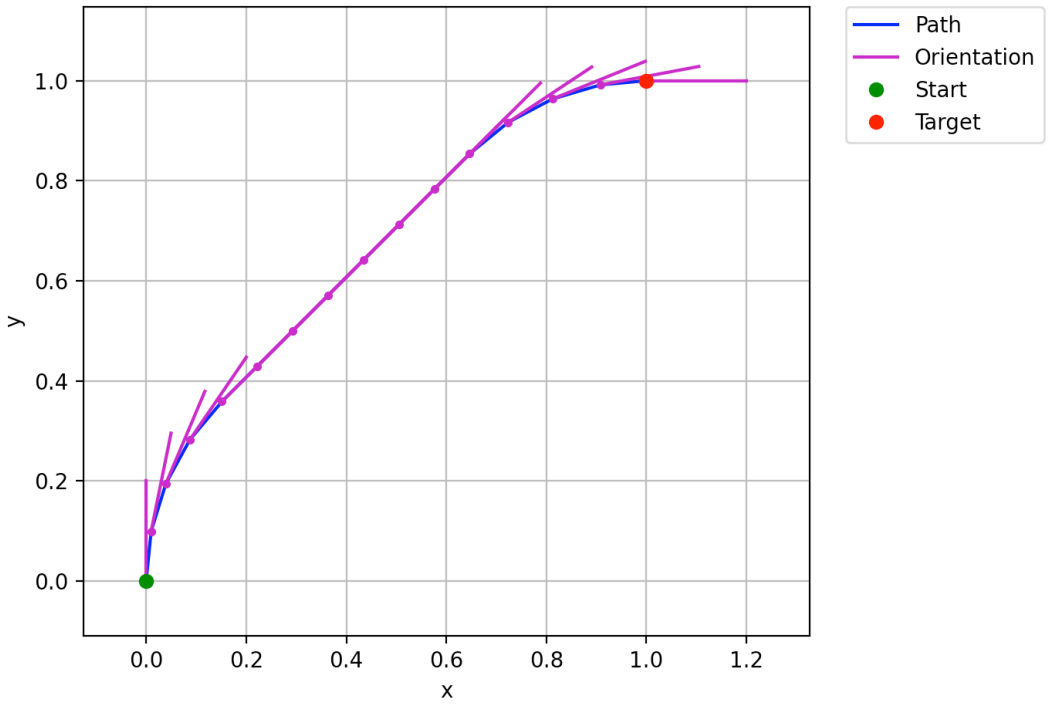
(a) (b)

Figure 3.8. Sample RLR and RSR Dubin curves.

- The path is dependent on the minimum curvature producible by the car.
- The duckiebot can “cheat” at being a Dubins Car since it can turn on the spot. This is key to avoiding generated paths that involve huge loops such as the one shown in the image below.
- This allows for paths that always point the camera towards the next checkpoint.



(a)



(b)

Figure 3.9. A Dubin path between the current and target pose with a minimum curvature of 2 units on the top and 0.5 on the bottom.

- Dubins path planning becomes a simple way to compute a trajectory where the camera is always pointing towards the next checkpoint.
- The set of forward velocities v and angular velocity ω at each time step is computed via forward Euler discretization.

4) Control

The planner produces a set of

- poses $\mathbf{x}$,
- linear velocities $\mathbf{v}$, and
- angular velocities $\boldsymbol{\omega}$,

for some set of time steps k. The controller then simply executes these generated inputs until a new measurement is available. Since new measurements arrive very quickly, it is rare for more than the first inputs to be executed. However, the robustness is there in case it is needed.

A number of additional checks and balances are also included to handle situations where only orientation information is available and for fringe situations. The controller is ultimately very simple on the account that the distances in the intersection are very short and new measurements arrive quickly. The reader is referred to the failure section for additional information about considered control options.

5) Checkpoints Manager

- The checkpoints manager supports easy collection, saving, and loading of checkpoints with the click of a button
- It keeps track of the current checkpoint and switches the checkpoint when the affine transformation is almost equal to an identity matrix

3.6. Results

No numerical evaluation criteria are defined at this stage of the project. A possible criteria could be the percent of successful turns, however the algorithm is not currently robust enough to produce good results frequently enough for this to be a worthwhile metric.

Qualitatively, the presented algorithm can serve as a proof of concept that visual servoing could in fact be used for intersection crossing and lane following. As can be seen in the videos presented in the results overview section, when the algorithm works it works very well.

Unfortunately, the algorithm does not currently perform very robustly. A likely reason is the inconsistency of the line detection and mismatches happening due to a lack of checks and balances. For example, the algorithm will sometimes match a corner that is off the visible screen, which is clearly not a good match. These checks and balances are only found during testing, and we unfortunately ran out of time to implement all of them.

Nonetheless, the feasibility of this approach has been proven and it is hoped that subsequent work will occur to implement the next steps listed below. Please refer to this instructions section for some fun failure videos!

3.7. Future avenues of development

Here we list down the aspects of our solution which can be tuned to greatly improve the outcome given more time.

- Line detection: Solving a vision driven task, our solution heavily relies on correctly estimating image features such as lines and their colors. An accurate line detection module is even more important for navigating intersections since the bot needs to rely on several distant lines in all directions to estimate its state. We found our solution to mostly be limited by line detection's accuracy and we see a potentially better solution as a direct result of improving this module with more experimentation.
- Algorithm start: The algorithm currently relies on the bot being in front of a red line marking the end of one line. As such, it would be necessary to implement a "pre-roll" in order to move into a better location for visual servoing to start.
- User interface: The fully envisioned algorithm would be capable of traversing a duckietown fully autonomously, only requiring user input for decisions as to where to turn. With additional time, it would be great to implement loading in all 3 types of turns, as well as line following. The procedure could then be automated so that it would automatically enter lane following up until it reaches a red line, at which point the user

could click an arrow to indicate which way the bot should turn. After executing the turn, the bot would automatically enter lane following again and repeat.

- PID controller overlay: In order to improve the controller it would be good to overlay a PID or simply a D control law on top of the planned dubins path to ensure smoother accelerations.
- Filtered target: Due to the "jumpiness" of the line detection, it would be good to have a filter or even just an averager for the desired pose which could smooth the measurements and decrease the sudden shift in input values.

3.8. Contribution / Added functionality

The exact contributions of our work can be summarized as follows.

- An algorithm capable of reaching a reference/checkpoint duckietown lane image from a nearby position where at least some similar lines are visible.
- A checkpoint management system capable of recording checkpoints, saving them for later use, and automatically switching from checkpoint to checkpoint during a control sequence.
- A computer vision module capable of detecting and projecting all visible road lines. The module is additionally capable of determining corners where a red line meets either a yellow or a white line. Finally, this module can also determine matching lines and points between two images close to each other in physical space.
- A dubins path planning algorithm capable of reaching a target $\boldsymbol{SE(2)}$ pose.

3.9. Recorded Failures and Changes

This section is meant to list out some of the approaches that were considered or implemented and why they were not used in the final submission. The goal is for this to serve as a list of ideas and warnings for anyone interested in continuing this work.

1) Vision

- Initially, we employed SIFT algorithm as our vision model. It succeeded in capturing image features in the background but failed in capturing image features on the lanes due to the reasons: 1. the background is more complex and the magnitudes of the image gradients are high in the background; 2. a lot of mismatched features on the lanes due to the scale, rotation and photometric invariant properties of SIFT (the image patches are similar and repetitive on the lanes).
- To avoid the problems with SIFT, we experimented with several homography computation approaches that allow us to compute the homography matrix using line and point information interchangeably. These models yield decent homography matrices. However, they rely heavily on perfect detection of the lanes and the homography computed is infeasible for the control model.
- Additionally, we intended to replace the homography matrices with affine matrices. We implemented an algorithm which computes affine matrices with 6 DOF. However, we realized that the affine matrix computed exhibits scaling and shearing effects on translation. Thus, we added restrictions to our line detections model and this led to our

final model.

2) Planning/Control

- Originally, an optimal Model Predictive Control (MPC) strategy was considered. However, because the trajectory was only ever planned for short distances given the nature of the problem, the added complexity was deemed unnecessary and the inputs from the planner were fed in directly.
- In hindsight, perhaps a dubins algorithm is not necessary and a simple PID loop would work. Due to the shortness of the trajectory and the frequency of updates, the dubins path is frequently only a couple of steps long and is rarely followed through to the second input. In theory however, it will nonetheless be more robust than a PID controller.

3.10. Reference

1. A. Giese, “A Comprehensive , Step-by-Step Tutorial to Computing Dubin’s Paths,”Andy G’s Blog, pp. 1–20, 2012. [Online]. Available:https://gieseanw.wordpress.com/2012/10/21/a-comprehensive-step-by-step-tutorial-to-computing-dubins-paths/. ↩

2. L. E. Dubins, “On Curves of Minimal Length with a Constraint on AverageCurvature, and with Prescribed Initial and Terminal Positions and Tangents,”American Journal of Mathematics, vol. 79, no. 3, p. 497, 1957. ↩

3. S. Hutchinson, G. D. Hager and P. I. Corke, “A tutorial on visual servo control,” in IEEE Transactions on Robotics and Automation, vol. 12, no. 5, pp. 651-670, Oct. 1996, doi: 10.1109/70.538972. ↩

4. J. T. Feddema and O. R. Mitchell, “Vision-guided servoing with feature-based trajectory generation (for robots),” in IEEE Transactions on Robotics and Automation, vol. 5, no. 5, pp. 691-700, Oct. 1989, doi: 10.1109/70.88086. ↩

5. F. Chaumette and S. Hutchinson, “Visual servo control. I. Basic approaches,” in IEEE Robotics and Automation Magazine, vol. 13, no. 4, pp. 82-90, Dec. 2006, doi: 10.1109/MRA.2006.250573. ↩

6. Andreff N, Espiau B, Horaud R. Visual Servoing from Lines. The International Journal of Robotics Research. 2002;21(8):679-699. doi:10.1177/027836402761412430 ↩

7. Ackermann, M.R., Blömer, J., Kuntze, D. et al. Analysis of Agglomerative Clustering. Algorithmica 69, 184–215 (2014). doi:10.1007/s00453-012-9717-4 ↩ (unknown ref 10.1007/s00453-012-9717-4)

previous **warning** next (3 of 7) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#fn-
ref:10.1007/s00453-012-9717-4'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

8. Jianbo Shi and Tomasi, “Good features to track,” 1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, Seattle, WA, USA, 1994, pp. 593-600, doi: 10.1109/CVPR.1994.323794. ↩

UNIT A-4

Instructions Cross Pro-Duck

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Duckietown Simulator without obstacles. Project Code: GitHub repo

4.1. Video of expected results

Below is a compilation of some of our best runs with the duckiebot performing right turns, crossing straight through an intersection, and doing lane following up to a red line!

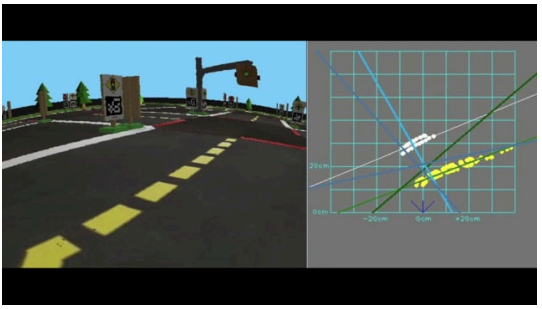


Figure 4.1. Successful Right Turns.

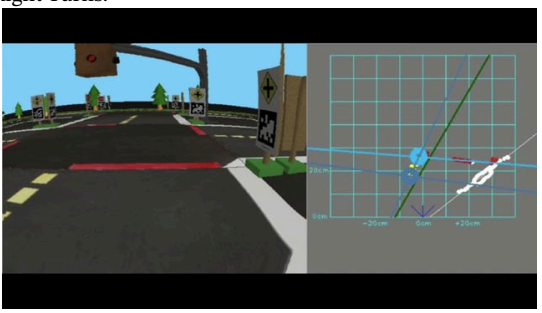


Figure 4.2. Successful Lane Following with Red Line Approach.

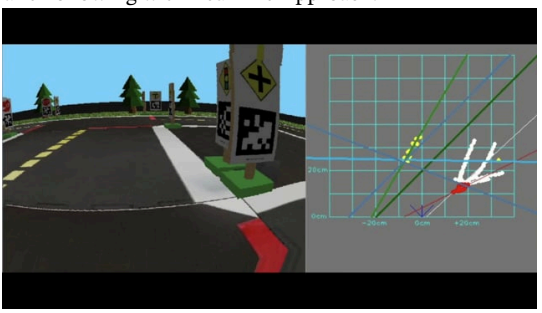


Figure 4.3. Successful Straight Crossing.

4.2. Laptop setup notes

The user is expected to have the latest version of `duckietown-shell` installed. It is also expected to have the latest Duckietown Docker images. In addition, the user needs to install the Python packages that we list down in the `requirements.txt` file within our project directory. The main requirement of interest is the python `dubins`1 library, which is required for the path planning. Although not the most long term solution, this package was included by simply adding `pip3 install dubins` in the `launchers/run_interface.sh` file.

4.3. Duckietown setup notes

Although we only test our solution in the `LFI-norm-4way-000` map in the simulator, we expect it to work similarly in any other map that does not involve obstacles such as other duckiebots, duckies, cones, etc. This means that any map can be chosen from here provided it has `LF` or `LFI` as prefix.

4.4. Pre-flight checklist

Check 1: The Duckietown shell is working by running `dts`. It is expected to see the shell's version and a prompt waiting for a command.

Check 2: You have cloned the project source code from here and you are in the root of the project. You should see `exercise_ws` as one of the directories on running `ls`.

Check 3: The `exercise_ws` directory has all of the custom ROS nodes that we provide: `custom_msgs`, `ground_projection`, `lane_control`, `lane_filter`, `line_detector`, and `vision_opencv`.

Check 4: The `exercise_ws/checkpoints` directory has at least one directory containing a set of images referring to the kind of path the duckiebot is expected to follow.

4.5. Instructions

Here, give step by step instructions to reproduce the demos shown in [the videos]{#demo-cross-pro-duck-expected}.

Step 1: In the `line_detector/config/default.yaml` file, uncomment the desired `checkpoints_path` as described by the ending of the path. Note that `turn_left` does not currently work.

Step 2: Build the exercise by running `dts exercises build`. You should see messages like "All 9 packages succeeded!" and "Build complete" at the end.

Step 3: Run `dts exercises test --sim --pull` command. You should initially see the Duckietown shell pulling the latest Docker images. Once that is done, the shell will launch the ROS interface and the agent.

Step 4: The agent can be controlled via the noVNC2 web app hosted by default at localhost:8087. Launch the virtual joystick app by running `dt-launcher-joystick` in the noVNC2 command prompt to control the robot. The robot's current view can be visualized in `RQT Image View` on topic `/agent/camera_node/image/compressed`. The detected lines can be viewed on topic `/agent/line_detector_node/debug/vs_lines_all/`

`compressed` as shown in the demo videos above.

Step 5: For the purpose of this demonstration, the user is expected to drive the Duckiebot to a point from where the checkpoint image is visible. For an intersection crossing, this means driving the bot right in front of the red line until the red line to the right of you is visible. For lane following/red line approach the bot simply needs to be in a lane. This is to ensure that the target image is visible from the current view which is a requirement of the Visual Servoing algorithm.

Step 6: To make the robot drive autonomously through the checkpoints, the user can run `docker exec -it $(docker ps -aqf "name=^agent$") /bin/bash /code/exercise_ws/src/servo.sh` command in their home terminal or simply press `e` and then the `up` arrow in the virtual joystick. Then, press `a` in the virtual joystick. The robot should then start moving, follow the reference trajectory, stop once it has reached the end, without any intervention.

Step 7: To manually drive the robot through the checkpoints, simply follow the same procedure as in Step 6, however do not press `a`. You can simply use the arrow keys in the virtual joystick to drive the robot with the checkpoints being active. It is possible to switch into autonomous mode at any point by pressing `a` and to switch back to manual mode by pressing `s`. If you wish to stop checkpoints from being used, simply press `e` and then the `down` arrow in the virtual joystick.

Step 8: Interested users are again referred to the topic `/agent/line_detector_node/debug/vs_lines_all/compressed` to monitor the state of the robot. The visualization displays all the detected yellow, red, and white lines. It also displays in green the lines or intersection points in blue that are being used to compute the transformation matrix.

Step 9: Once finished, users can drive the robot back to the starting position to try again or load a different set of checkpoints by specifying its path in parameter of `LineDetectorNode`'s config and restarting the agent.

Collecting Checkpoints:

In addition to the default set of checkpoints that we provide, it is easy to collect custom checkpoints.

Step 1: Set the `checkpoints_path` parameter of `LineDetectorNode`'s config to `"None"`.

Step 2: Perform Steps 2 - 4 of the previous section

Step 3: Using the joystick, drive to the locations that you want to save as checkpoints. At each intended location, run `docker exec -it $(docker ps -aqf "name=^agent$") /bin/bash /code/exercise_ws/src/checkpoint.sh` or press `e` and then the `left` arrow in the virtual joystick to save the current image as a checkpoint. Repeat this step until all the checkpoints have been captured. At each save, you should see an stdout message like "Saved checkpoint with id xxxxxxxx. Total checkpoints: 3."

Step 4: Run `docker exec -it $(docker ps -aqf "name=^agent$") /bin/bash /code/exercise_ws/src/checkpoints_to_disk.sh` or `e` and then the `right` arrow in the virtual joystick to save all the collected checkpoints to disk. To load the checkpoints in future, simply set the `checkpoints_path` parameter to the path printed in the output of the command.

4.6. Troubleshooting

Symptom: The Duckiebot did not follow the reference trajectory/checkpoints.

Resolution: Unfortunately, our solution is not very robust right now and it is expected that the robot will occasionally irreversibly deviate from its intended path. If this happens, stop the robot by pressing the key s and drive back to the initial position to try again or try different checkpoints/maps/turns. It is strongly advised that users try to manually run through checkpoints before engaging autonomous mode. Most of our successful experiments involve checkpoints for lane following, turning right or going straight at an intersection.

Symptom: The visual output gets stuck in the simulator.

Resolution: Follow the regular steps of reloading the agent, simulator, and your machine in that order. Sometimes problems would get fixed only after a good old fashioned restart of the entire machine, although the reasoning behind this is unclear...

Symptom: Some error gets thrown but the code continues running afterwards.

Resolution: Some fringe detection failures can throw errors at the agent start or even during runtime if something very strange is going on. If the code continues running then no restart is required.

Symptom: My saved checkpoints are not being loaded.

Resolution: Make sure you specify the full path `/code/exercise_ws/checkpoints/YOUR_FOLDER_NAME`.

Symptom: My checkpoints are not being added or saved.

Resolution: Make sure your `checkpoints_path` is set to `None`.

4.7. Demo failure demonstration

Below is a video showing some of the common failures that we came across. Note that these occured with various settings, some of which have been better tuned since then. However, if a reader is to progress this work then they will inevitably change the parameters again and may observe similar issues.

In no particular order, some of the failures are:

- Overly aggressive turning, forward velocity.
- Under aggressive turning, forward velocity.
- Line detector gets confused and starts trying to align itself with an incorrect line.
- Poorly set homography tolerance leading to the checkpoint getting stuck.
- Poor checkpoint choices leading to the robot being completely lost. This is especially difficult for left turns.

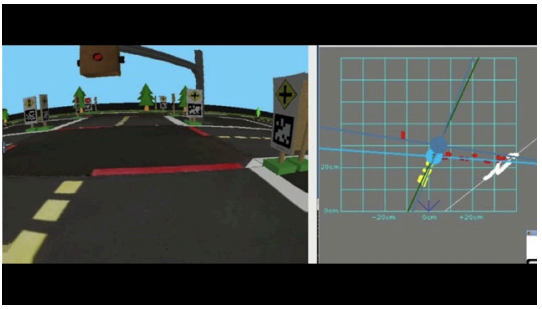


Figure 4.4. Various types of failed crossings and lane followings.

4.8. Reference

1. https://pypi.org/project/dubins/ ↩

PART B
Caffe and Tensorflow

Contents

UNIT B-1

How to install PyTorch on the Duckiebot

PyTorch is a Python deep learning library that's currently gaining a lot of traction, because it's a lot easier to debug and prototype (compared to TensorFlow / Theano).

To install PyTorch on the Duckietbot you have to compile it from source, because there is no pro-compiled binary for ARMv7 / ARMhf available. This guide will walk you through the required steps.

1.1. Step 1: install dependencies and clone repository

First you need to install some additional packages. You might already have installed. If you do, that's not a problem.

```
sudo apt-get install libopenblas-dev cython libatlas-dev m4 libblas-dev
```

In your current shell add two flags for the compiler

```
export NO_CUDA=1 # this will disable CUDA components of PyTorch, be-
cause the little RaspberriPi doesn't have a GPU that supports CUDA
export NO_DISTRIBUTED=1 # for distributed computing
```

Then `cd` into a directory of your choice, like `cd ~/Downloads` or something like that and clone the PyTorch library.

```
git clone --recursive https://github.com/pytorch/pytorch
```

1.2. Step 2: Change swap size

When I was compiling the library I ran out of SWAP space (which is 500MB by default). I was successful in compiling it with 2GB of SWAP space. Here is how you can increase the SWAP (only for compilation - later we will switch back to 500MB).

Create the swap file of 2GB

```
sudo dd if=/dev/zero of=/swap1 bs=1M count=2048
```

Make this empty file into a swap-compatible file

```
sudo mkswap /swap1
```

Then disable the old swap space and enable the new one

```
sudo nano /etc/fstab
```

This above command will open a text editor on your `/etc/fstab` file. The file should have this as the last line: `/swap0 swap swap`. In this line, please change the `/swap0` to `/swap1`. Then save the file with CTRL+o and ENTER. Close the editor with CTRL+x.

Now your system knows about the new swap space, and it will change it upon reboot, but if you want to use it right now, without reboot, you can manually turn off and empty the old swap space and enable the new one:

```
sudo swapoff /swap0
sudo swapon /swap1
```

1.3. Step 3: compile PyTorch

`cd` into the main directory, that you clones PyTorch into, in my case `cd ~/Downloads/pytorch` and start the compilation process:

```
python setup.py build
```

This shouldn't create any errors but it took me about an hour. If it does throw some exceptions, please let me know.

When it's done, you can install the pytorch package system-wide with

```
sudo -E python setup.py install # the -E is important
```

For some reason on my machine this caused recompilation of a few packages. So this might again take some time (but should be significantly less).

1.4. Step 4: try it out

If all of the above went through without any issues, congratulations. :) You should now have a working PyTorch installation. You can try it out like this.

First you need to change out of the installation directory (**this is important - otherwise you get a really weird error**):

```
cd ~
```

Then run Python:

```
python
```

And in the Python interpreter try this:

```
>>> import torch
>>> x = torch.rand(5, 3)
>>> print(x)
```

1.5. (Step 5, optional: unswap the swap)

Now if you like having 2GB of SWAP space (additional RAM basically, but a lot slower than your built-in RAM), then you are done. The downside is that you might run out of space later on. If you want to revert back to your old 500MB swap file then do the following:

Open the `/etc/fstab` file in the editor:

```
sudo nano /etc/fstab
```

please change the `/swap0` to `/swap1`. Then save the file with CTRL+o and ENTER. Close the editor with CTRL+x.

UNIT B-2

How to install Caffe and Tensorflow on the Duckiebot

Caffe and TensorFlow are popular deep learning libraries, and are supported by the Intel Neural Computing Stick (NCS).

2.1. Caffe

1) Step 1: install dependencies and clone repository

Install some of the dependencies first. The last command "sudo pip install ....." will cause some time.

```
sudo apt-get install -y gfortran cython
sudo apt-get install -y libprotobuf-dev libleveldb-dev libsnappy-dev li-
bopencv-dev libhdf5-serial-dev protobuf-compiler git
sudo apt-get install --no-install-recommends libboost-all-dev
sudo apt-get install -y python-dev libgflags-dev libgoogle-glog-dev li-
blmdb-dev libatlas-base-dev python-skimage
sudo pip install pyzmq jsonschema pillow numpy scipy ipython jupyter
pyyaml
```

Then, you need to clone the repo of caffe

```
cd
git clone https://github.com/BVLC/caffe
```

2) Step 2: compile Caffe

Before compile Caffe, you have to modify Makefile.config

```
cd caffe
cp Makefile.config.example Makefile.config
sudo vim Makefile.config
```

Then, change four lines from

```
'#'CPU_ONLY := 1
/usr/lib/python2.7/dist-packages/numpy/core/include
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib
```

to

```
CPU_ONLY := 1
/usr/local/lib/python2.7/dist-packages/numpy/core/include
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include /usr/include/hdf5/
serial/
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib /usr/lib/arm-lin-
ux-gnueabihf/hdf5/serial/
```

Next, you can start to compile caffe

```
make all
make test
make runtest
make pycaffe
```

If you did't get any error above, congratulation on your success. Finally, please export pythonpath

```
sudo vim ~/.bashrc
export PYTHONPATH=/home/"$USER"/caffe/python:$PYTHONPATH
```

3) Step 3: try it out

Now, we can confirm whether the installation is successful. Download AlexNet and run caffe time

```
cd ~/caffe/
python scripts/download_model_binary.py models/bvlc_alexnet
./build/tools/caffe time -model models/bvlc_alexnet/deploy.prototxt
-weights  models/bvlc_alexnet/bvlc_alexnet.caffemodel   -iterations 10
```

And you can see the benchmark of AlexNet on Pi3 caffe.

2.2. Tensorflow

1) Step 1: install dependencies and clone repository

First, update apt-get:

```
$ sudo apt-get update
```

For Bazel:

```
$ sudo apt-get install pkg-config zip g++ zlib1g-dev unzip
```

For Tensorflow: (NCSDK only support python 3+. I didn't use mvNC on rpi3, so here I

choose python 2.7)
(For Python 2.7)

```
$ sudo apt-get install python-pip python-numpy swig python-dev
$ sudo pip install wheel
```

(For Python 3.3+)

```
$ sudo apt-get install python3-pip python3-numpy swig python3-dev
$ sudo pip3 install wheel
```

To be able to take advantage of certain optimization flags:

```
$ sudo apt-get install gcc-4.8 g++-4.8
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8
100
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8
100
```

Make a directory that hold all the thing you need

```
$ mkdir tf
```

2) Step 2: build Bazel

Download and extract bazel (here I choose 0.7.0):

```
$ cd ~/tf
$ wget https://github.com/bazelbuild/bazel/releases/download/0.7.0/
bazel-0.7.0-dist.zip
$ unzip -d bazel bazel-0.7.0-dist.zip
```

Modify some file:

```
$ cd bazel
$ sudo chmod u+w ./* -R

$ nano scripts/bootstrap/compile.sh
```

To line 117, add “-J-Xmx500M”:

```
run "${JAVAC}" -classpath "${classpath}" -sourcepath "${sourcepath}" \
      -d "${output}/classes" -source "$JAVA_VERSION" -target "$JAVA_VERSION" \
      -encoding UTF-8 "@${paramfile}" -J-Xmx500M
```

Figure 2.1

```
$ nano tools/cpp/cc_configure.bzl
```

Place the line return “arm” around line 133 (beginning of the _get_cpu_value function):

```
...
"""Compute the cpu_value based on the OS name."""
return "arm"
...
```

Figure 2.2

Build Bazel (it will take a while, about 1 hour):

```
$ ./compile.sh
```

When the build finishes:

```
$ sudo cp output/bazel /usr/local/bin/bazel
```

Run bazel check if it’s working:

```
$ bazel
```

```
Usage: bazel <command> <options> ...

Available commands:
  analyze-profile     Analyzes build profile data.
  build               Builds the specified targets.
  canonicalize-flags  Canonicalizes a list of bazel options.
  clean               Removes output files and optionally stops the server.
  dump                Dumps the internal state of the bazel server process.
  fetch               Fetches external repositories that are prerequisites to the targets.
  help                Prints help for commands, or the index.
  info                Displays runtime info about the bazel server.
  mobile-install      Installs targets to mobile devices.
  query               Executes a dependency graph query.
  run                 Runs the specified target.
  shutdown            Stops the bazel server.
  test                Builds and runs the specified test targets.
  version             Prints version information for bazel.

Getting more help:
  bazel help <command>
                  Prints help and options for <command>.
  bazel help startup_options
                  Options for the JVM hosting bazel.
  bazel help target-syntax
                  Explains the syntax for specifying targets.
  bazel help info-keys
                  Displays a list of keys used by the info command.
```

Figure 2.3

3) Step 3: compile Tensorflow

Clone tensorflow repo (here I choose 1.4.0):

```
$ cd ~/tf
$ git clone -b r1.4 https://github.com/tensorflow/tensorflow.git
$ cd tensorflow
```

(Incredibly important) Changes references of 64-bit program implementations (which we don't have access to) to 32-bit implementations.

```
$ grep -Rl 'lib64' | xargs sed -i 's/lib64/lib/g'
```

Modify the file platform.h:

```
$ sed -i "s|#define IS_MOBILE_PLATFORM|//#define IS_MOBILE_PLATFORM|g"
tensorflow/core/platform/platform.h
```

Configure the build: (important) if you want to build for Python 3, specify /usr/bin/python3 for Python's location and /usr/local/lib/python3.x/dist-packages for the Python library path.

```
$ ./configure
Please specify the location of python. [Default is /usr/bin/python]: /usr/bin/python
Please specify optimization flags to use during compilation when bazel option "--config=opt"
Do you wish to use jemalloc as the malloc implementation? [Y/n] Y
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] N
Do you wish to build TensorFlow with Hadoop File System support? [y/N] N
Do you wish to build TensorFlow with the XLA just-in-time compiler (experimental)? [y/N] N
Please input the desired Python library path to use. Default is [/usr/local/lib/python2.7/di
Do you wish to build TensorFlow with OpenCL support? [y/N] N
Do you wish to build TensorFlow with CUDA support? [y/N] N
```

Figure 2.4

Build the Tensorflow (this will take a LOOOONG time, about 7 hrs):

```
$ bazel build -c opt --copt="-mfpu=neon-vfpv4" --copt="-funsafe-math-op-
timizations" --copt="-ftree-vectorize" --copt="-fomit-frame-pointer" --
local_resources 1024,1.0,1.0 --verbose_failures tensorflow/tools/
pip_package:build_pip_package
```

After finished compiling, install python wheel:

```
$ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensor-
flow_pkg
$ sudo pip install /tmp/tensorflow_pkg/tensorflow-1.4.0-cp27-none-lin-
ux_armv7l.whl
```

Check version:

```
$ python -c 'import tensorflow as tf; print(tf.__version__)'
```

And you're done! You deserve a break.

4) Step 3: try it out

Suppose you already have inception-v3 model (with inception-v3.meta and inception-v3.ckpt)

Create a testing python file

```
$ vim test.py
```

Write the following code:

```
1  import tensorflow as tf
2  import numpy as np
3  import cv2
4  import sys
5  import time
6
7  def run(input_image):
8      tf.reset_default_graph()
9      with tf.Session() as sess:
10         saver = tf.train.import_meta_graph('./output/inception-v3.meta')
11         saver.restore(sess, 'inception_v3.ckpt')
12
13         softmax_tensor = sess.graph.get_tensor_by_name('Softmax:0')
14         feed_dict = {'input:0': input_image}
15         classification = sess.run(softmax_tensor, {'input:0': input_image})    #first run fo warm-up
16
17         start_time = time.time()
18         classification = sess.run(softmax_tensor, {'input:0': input_image})
19         print 'predict label:', np.argmax(classification[0])
20         print 'predict time:', time.time() - start_time, 's'
21
22 if __name__=="__main__":
23     args = sys.argv
24     if len(args) != 2:
25         print 'Usage: python %s filename'%args[0]
26         quit()
27     image_data =tf.gfile.FastGFile(args[1], 'rb').read()
28     image = cv2.imread(args[1])
29     image = cv2.resize(image, (299,299))
30     image = np.array(image)/255.0
31     image = np.asarray(image).reshape((1, 299, 299, 3))
32     run(image)
```

Figure 2.5

Save, and excute it

```
$ python test.py cat.jpg
```

Then it will show the predict label and predict time.

UNIT B-3

Movidius Neural Compute Stick Install

3.1. Laptop Installation

install based on ncsdk website

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk
make install
make examples
```

test installation

```
cd ~/ncsdk/examples/app/hello_ncs_py/
make run
```

3.2. Duckiebot Installation

you only need to install the NCSDK but there is also the option of installing Caffe and/or Tensorflow as well, in order to perhaps speed up the development cycle. I would recommend against it, as it can be a bigger problem than it solves.

1) Barebones Install (recommended)

you don't need tensorflow, caffe, or any tools in order to run the compiled networks and not installing them will save you a lot of hassle

on duckiebot:

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk/api/src
make
sudo make install
```

2) Caffe/Tensorflow Install

> Note: if you want to be able to compile your models on the duckiebot itself, install tensorflow or caffe beforehand and remember to install for python 3 (`pip3`)

follow directions here

make sure caffe and tensorflow as installed

```
python3 -c 'import tensorflow as tf; import caffe'
```

install sdk:

```
git clone http://github.com/Movidius/ncsdk
cd ~/ncsdk
make install
make examples
```

UNIT B-4

How To Use Neural Compute Stick

4.1. Workflow

create and train model in tensorflow or caffe (brief note on configuration)
save tensorflow model as a `.meta` (or caffe model in `.prototxt`)

```
saver = tf.train.Saver()
...
saver.save(sess, 'model')
```

compile the model into NC format (documentation here)

```
mvNCCompile model.meta -o model.graph
```

move model onto duckiebot

```
scp model.meta user@robot name:~/path_to_networks/
```

run the compiled model

```
with open(path_to_networks + model.meta, mode='rb') as f:
    graphfile = f.read()
graph = device.AllocateGraph(graphfile)
graph.LoadTensor(input_image.astype(numpy.float16), 'user object')
output, userobj = graph.GetResult()
```

4.2. Benchmarking

get benchmarking (frames per second) from their app zoo

```
git clone https://github.com/movidius/ncappzoo
cd ncappzoo/apps/benchmarkncs
./mobilenets_benchmark.sh | grep FPSk
```

PART C

ETH Autonomous mobility on Demand 2019: Final Reports

Contents

UNIT C-1

Group name: final report

Before starting, you should have a look at some tips on how to write beautiful Duckiebook pages (unknown ref duckumentation/contribute)

previous **warning** next (4 of 7) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#ducku-
mentation/contribute'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

The objective of this report is to bring justice to your extraordinarily hard work during the semester and make so that future generations of Duckietown students may take full advantage of it. Some of the sections of this report are repetitions from the preliminary design document (PDD) and intermediate report you have given.

1.1. The final result

Let's start from a teaser.

- Post a video of your best results (e.g., your demo video): remember to have duckies on the robots or something terrible might happen!

Add as a caption: see the operation manual to reproduce these results. Moreover, add a link to the readme.txt of your code.

1.2. Mission and Scope

Now tell your story:

Define what is your mission here.

1) Motivation

Now step back and tell us how you got to that mission.

- What are we talking about? [Brief introduction / problem in general terms]
- Why is it important? [Relevance]

2) Existing solution

- Was there a baseline implementation in Duckietown which you improved upon, or did you implemented from scratch? Describe the "prior work"

3) Opportunity

- What was wrong with the baseline / prior work / existing solution? Why did it need improvement?

Examples: - there wasn't a previous implementation - the previous performance, evaluated according to some specific metrics, was not satisfactory - it was not robust / reliable - somebody told me to do so (/s) (this is a terrible motivation. In general, never ever ever say "somebody told me to do it" or "everybody does like this")

- How did you go about improving the existing solution / approaching the problem? [contribution]

Examples: - We used method / algorithm xyz to fix the gap in knowledge (don't go in the details here) - Make sure to reference papers you used / took inspiration from, lessons, textbooks, third party projects and any other resource you took advantage of (check here how to add citations in this document). Even in your code, make sure you are giving credit in the comments to original authors if you are reusing some components.

4) Preliminaries

- Is there some particular theorem / "mathy" thing you require your readers to know before delving in the actual problem? Briefly explain it and links for more detailed explanations here.

Definition of link: - could be the reference to a paper / textbook - (bonus points) it is best if it is a link to Duckiebook chapter (in the dedicated "Preliminaries" section)

1.3. Definition of the problem

Up to now it was all fun and giggles. This is the most important part of your report: a crisp, possibly mathematical, definition of the problem you tackled. You can use part of the preliminary design document to fill this section.

Make sure you include your: - final objective / goal - assumptions made - quantitative performance metrics to judge the achievement of the goal

1.4. Contribution / Added functionality

Describe here, in technical detail, what you have done. Make sure you include: - a theoretical description of the algorithm(s) you implemented - logical architecture - software architecture - details on the actual implementation where relevant (how does the implementation differ from the theory?) - any infrastructure you had to develop in order to implement your algorithm - If you have collected a number of logs, add link to where you stored them

Feel free to create subsections when useful to ease the flow

1.5. Formal performance evaluation / Results

Be rigorous!

- For each of the tasks you defined in you problem formulation, provide quantitative results (i.e., the evaluation of the previously introduced performance metrics)
- Compare your results to the success targets. Explain successes or failures.
- Compare your results to the "state of the art" / previous implementation where relevant. Explain failure / success.
- Include an explanation / discussion of the results. Where things (as / better than / worst than) you expected? What were the biggest challenges?

1.6. Future avenues of development

Is there something you think still needs to be done or could be improved? List it here, and be specific!

PART D

ETH Autonomous mobility on Demand 2019: Final Reports

Contents

UNIT D-1

Demo template

Before starting, you should have a look at some tips on how to write beautiful Duckiebook pages (unknown ref duckumentation/contribute)

previous **warning** next (5 of 7) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#ducku-
mentation/contribute'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

This is the template for the description of a demo. The spirit of this document is to be an operation manual, i.e., a straightforward, unambiguous recipe for reproducing the results of a specific behavior or set of behaviors.

It starts with the "knowledge box" that provides a crisp description of the border conditions needed:

- Duckiebot hardware configuration (see Duckiebot configurations)
- Duckietown hardware configuration (loops, intersections, robotarium, etc.)
- Number of Duckiebots
- Duckiebot setup steps

For example:

KNOWLEDGE AND ACTIVITY GRAPH

Requires: Duckiebot in configuration `DB18`

Requires: Duckietown without intersections

Requires: Camera calibration completed

1.1. Video of expected results

First, we show a video of the expected behavior (if the demo is successful).

Make sure the video is compliant with Duckietown, i.e. : the city meets the appearance specifications and the Duckiebots have duckies on board.

1.2. Duckietown setup notes

Here, describe the assumptions about the Duckietown, including:

- Layout (tiles types)
- Infrastructure (traffic lights, WiFi networks, ...) required
- Weather (lights, ...)

Do not write instructions on how to build the city here, unless you are doing something very particular that is not in the Duckietown operation manual (unknown ref opmanual_duckietown/duckietowns)

previous **warning** next (6 of 7) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckietown/duckietowns'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

. Here, merely point to them.

1.3. Duckiebot setup notes

Write here any special setup for the Duckiebot, if needed.

Do not repeat instructions here that are already included in the Duckiebot operation manual (unknown ref opmanual_duckiebot/opmanual_duckiebot)

previous **warning** (7 of 7) index

warning

```
I will ignore this because it is an external link.

 > I do not know what is indicated by the link '#opman-
ual_duckiebot/opmanual_duckiebot'.
```

Location not known more precisely.

Created by function `n/a` in module `n/a`.

.

1.4. Pre-flight checklist

The pre-flight checklist describes the steps that are sufficient to ensure that the demo will be correct:

Check: operation 1 done

Check: operation 2 done

1.5. Demo instructions

Here, give step by step instructions to reproduce the demo.

Step 1: XXX

Step 2: XXX

Make sure you are specifying where to write each line of code that needs to be executed, and what should the expected outcome be. If there are typical pitfalls / errors you experienced, point to the next section for troubleshooting.

1.6. Troubleshooting

Add here any troubleshooting / tips and tricks required, in the form:

Symptom: The Duckiebot flies

Resolution: Unplug the battery and send an email to info@duckietown.org

Symptom: I run `this elegant snippet of code` and get this error: `a nasty line of gibberish`

Resolution: Power cycle until it works.

1.7. Demo failure demonstration

Finally, put here video of how the demo can fail, when the assumptions are not respected.

You can upload the videos to the Duckietown Vimeo account and link them here.

Other learning modules

Logo